JAVA Design Patterns

Most Contents are from below websites for learning purpose.

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

<https://github.com/iluwatar/java-design-patterns>

<https://java-design-patterns.com/patterns>

A design pattern is a well-described solution to a common software problem.

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.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code.
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:

1. Creational
2. Structural
3. Behavioral

Following are different patterns for each categories:

1. Creational Design Patterns
2. Singleton
3. Factory
4. Abstract Factory
5. Builder
6. Prototype
7. Structural Design Patterns
8. Adapter
9. Composite
10. Proxy
11. Flyweight
12. Facade
13. Bridge
14. Decorator
15. Behavioral Design Patterns
16. Template Method
17. Mediator
18. Chain of Responsibility
19. Observer
20. Strategy
21. Command
22. State
23. Visitor
24. Interpreter
25. Iterator
26. Memento

**A. Creational Design Patterns**

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

**1. 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.
* 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.
* 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.
* Following are different approaches of Singleton pattern implementation and design concerns:

a. Eager Initialization

b. Static Block Initialization

c. Lazy Initialization

d. Thread Safe Singleton

e. Bill Pugh Singleton Implementation

f. Using Reflection to destroy Singleton Pattern

g. Enum Singleton

h. Serialization and Singleton

**a.** **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.

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.

**b. Static Block Initialization**

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

**c. Lazy Initialization**

Lazy initialization method to implement Singleton pattern creates the instance in the global access method.

**d. Thread Safe Singleton**

The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time.

**e. 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.

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.

**f. Using Reflection To Destroy Singleton Pattern**

Reflection can be used to destroy all the above singleton implementation approaches.

**g. 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 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.

**h. 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.

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.

**2. 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.

Super class in factory design pattern can be an interface, abstract class or a normal java class.

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.

**3. Abstract Factory Pattern**

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

A single Factory class that returns the different sub-classes based on the input provided 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.

Advantages ::

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.

**4. Builder 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, 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.

we can implement builder design pattern in java as follows:

1. First of all you need to create a static nested 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 Computer then 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.

**5. Prototype 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 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 it’s a design decision.

**B. 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 pattern is used so that two unrelated interfaces can work together. The object that joins these unrelated interfaces is called an Adapter.

There are two approaches to implement adapter pattern **class adapter and object adapter**, both these approaches produce same result.

Class Adapter **–** This form uses java inheritance and extends the source interface, in our case Socket class.

Object Adapter **–** This form uses Java Composition and adapter contains the source object.

**2. Composite Pattern**

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.

Composite Pattern consists of following objects.

**Base Component –** Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.

**Leaf –** Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn’t have references to other Components.

**Composite –** It consists of leaf elements and implements the operations in base component.

Composite pattern base component defines the common methods for leaf and composites.

Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.

Composite pattern should be applied only when the group of objects should behave as the single object.

Composite design pattern can be used to create a tree like structure.

**3. Proxy Pattern**

Proxy design pattern intent according to GoF is:

Provide a surrogate or placeholder for another object to control access to it.

proxy design pattern is used when we want to provide controlled access of a functionality.

Proxy design pattern common uses are to control access or to provide a wrapper implementation for better performance.

**4. Flyweight Pattern**

According to GoF, flyweight design pattern intent is:

Use sharing to support large numbers of fine-grained objects efficiently.

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.

Before we apply flyweight design pattern, we need to consider following factors:

1. The number of Objects to be created by application should be huge.

2. The object creation is heavy on memory and it can be time consuming too.

3. The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an

Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into intrinsic and extrinsic properties.

Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations.

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map of Objects in the factory that should not be accessible by client application.

Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

The client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but it’s a complete design decision for particular application.

Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.

Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

**5. Facade Pattern**

According to GoF Facade design pattern is:

Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

Facade pattern interface is a lot easier and cleaner way to avoid having a lot of logic at client side.

Facade design pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.

Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.

Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.

Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.

We can use Factory pattern with Facade to provide better interface to client systems.

**6. Bridge Pattern**

**7. Decorator Pattern**

**C. Behavioral Design Patterns :**