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**

**3. Abstract Factory Pattern**

**4. Builder Pattern**

**5. Prototype Pattern**

B. Structural Design Patterns

C. Behavioral Design Patterns