**Singleton Pattern**

**Use Case:**

At times, we would like to have only one instance of the class in the entire software program and the reason for this requirement could be due to limited resources (Such as the DB access, and file system) which can cause performance issues in the program, or maybe at times because of the business logic. The resource can be used across a program, so it should be accessed globally

**Singleton Pattern**

Singleton Pattern is the simplest design Pattern and comes under

creational Patterns which address the above design requirements. It restricts the instantiation of a class and ensures that only one instance of the class exists.

**Implementation**

To implement a singleton pattern, Class should have the below properties

* 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 the outer world to get the instance of the singleton class.

Example:

public class Singleton  
{  
 private Singleton() {}  
 private static Singleton instance = new Singleton();  
 public static Singleton instance() {return instance;}  
}

**Various approaches to implementing Singleton Pattern are**

1. **Eager initialization**

The instance of a singleton is created at the time of class loading. An example of Eager initialization is shown below. From the example, we can see we have a static Member, so the method is created during initialization and resources are allocated even though the client application is not using it. This method does not provide error handling

package com.example.singleton;

public class EagerInitializedSingleton {

private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

// private constructor to avoid client applications using the constructor

private EagerInitializedSingleton(){}

public static EagerInitializedSingleton getInstance() {

return instance;

}

}

1. **Static block initialization**

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

package com.example.singleton;

public class StaticBlockSingleton {

private static StaticBlockSingleton instance;

private StaticBlockSingleton(){}

// static block initialization for exception handling

static {

try {

instance = new StaticBlockSingleton();

} catch (Exception e) {

throw new RuntimeException("Exception occurred in creating singleton instance");

}

}

public static StaticBlockSingleton getInstance() {

return instance;

}

}

1. **Lazy Initialization**

An instance of a singleton class is initiated when a client application tries to use the class. By doing this, there is no resources wastage. This approach works well in a single thread however in a multithread application, two instances of a class will get created if two threads try to create a class at the same time.

package com.example.singleton;

public class LazyInitializedSingleton {

private static LazyInitializedSingleton instance;

private LazyInitializedSingleton(){}

public static LazyInitializedSingleton getInstance() {

if (instance == null) {

instance = new LazyInitializedSingleton();

}

return instance;

}

}

1. **Thread Safe Singleton**

To create a thread-safe singleton class, we use synchronized before the getInstance() method. By this, only one thread can invoke this method at any given time. But this will have a performance issue due to synchronized operation.

package com.example.singleton;

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

public static synchronized ThreadSafeSingleton getInstance() {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

return instance;

}

}

To avoid this extra overhead every time, the double-checked locking principle is used.

package com.example.singleton;

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

public static ThreadSafeSingleton getInstance() {

if (instance == null) {

synchronized (ThreadSafeSingleton.class){

if (instance == null) {

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

1. **Bill Pugh Singleton**

In the above approach, a synchronized is invoked when someone tries to access a singleton class. To overcome this, Bill Pugh comes up with a new idea of creating the singleton class using an inner static helper class.

package com.journaldev.singleton;

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper {

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance() {

return SingletonHelper.INSTANCE;

}

}