**Singleton Pattern**

\* Restricts the instantiation of a class and ensures that only one instance of the class exists in the JVM

\* Must provide a global access point to get the instance of the class

\* used for Logging, drivers objects, caching and thread pool

\* This pattern is also used in other pattersn (Abstract Factory, Builder, Prototype, Facade etc)

\* This pattern is also used in core Java classes e.g. java.lang.Runtime, java.awt.Desktop

**Implementations**

\* Private constructor to restrict instantiation of the class form other classes

\* Private Static varibale of the class

\* Public Static method that returns the instance of the class (global access point)

**Different Approaches of Implementations**

**1. Eager initialization**

\* The instance of singleton class is created at the time of class loading

\* Easiest way to create a singleton class (drawback: instance is created even when client applicantion might not be using it)

\* No options for Exception handling

\* Best approach when: your SC is not using a lot of resources

\* SC is used mostly for resources (File System, DB connections, etc) so it is best to avoid the instantiation unless required by the client

**2. Statis Block Initialization**

\* Similar to Eager (except that instance if class is created in the static block that provided exception handling)

\* Same as Eager (creates instance even before it is really required)

**3. Lazy Initialization**

\* Creates instance (in the global access method) when required (lazy)

\* Works file in single threaded environment

\* Chances of failure in multi threaded environment (multi threads might be in the loop at the same time and destroy the singleton pattern and all threads will get the different instances)

**4. Thread Safe Singleton**

\* The easiest way to create thread safe SC: make the global access method **synchronized** (only one thread can execute this method at a time)

\* Works fine and provides thread safety but **reduces** the performance because of cost of **synchronized** method

\* To avoid this extra overhead every time, **double checked locking** principle is used (synchronized block is used inside the if condition with an additonal check to ensure that only one instance is created)

**5. Bill Pugh Singleton Implementation**

\* Prior to JAVA 5, Java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance simultaneously.

\* Bill Pugh ( Ph.D. / Computer Science / Cornell University)



came up with a different approach (using a **Inner Static helper class**)

\* Private inner static helper class contains the instance of SC

\* When the SC is loaded, helper class isn’t loaded into memory but onl when someone class the getInstance() method, this (helper) class gets loaded and creates the SC instance.

\* Most widley used approach for SC (no Synchronization = no overload)

\* Easy to understand and implement

**\* Using reflection to destroy Singleton**

\* Reflection can be used to destroy all the above SC implementations

\* See example: (hash codes are different : two instances created : destroyed the SC)

\* Reflection is powerful and used in many frameworks (Spring, Hibernate etc)

**6. Enum Singleton**

\* To overcome this situation with reflection, Joshua Bloch suggested the use to Enum



\* As Java ensures that enum value is intantiated only once in a java program

\* Also, Java enum values are globally accessible (so is the SC)

\* Drawback: Enum type is little inflexible (e.g. it does not allow lazy initialization)

**7. Serialization and Singleton**

\* Sonetimes in distributed systems, we need to implement Serializable interface in SC so that we can store its state in the file system and retrieve it later.

\* See example: Problem: Whenever we deserialize it, it will create a new instance (destroys the SC).

\* To overcome this scenario, we need to provide the implementation of readResolve() method [see example]