Singleton Design Pattern

The Singleton design pattern ensures that a class has *only one instance* and provides *a global point of access* to that instance. It involves a single class responsible for creating an object and ensuring that only a single instance ever gets created. Singletons are often used to *share state* or to avoid the cost of setting up multiple objects. This pattern is widely used in scenarios where only one instance of a class should exist, such as logging, caching, database connections, or thread pools.

Implementation Guidelines:

To implement the Singleton pattern, the following guidelines are typically followed:

1. **Static Member:** The class should contain a *static member variable* to hold the single instance of the class.

2. **Private Constructor:** The constructor of the class should be made private to prevent external instantiation of the class.

3. **Static Factory Method:** A static factory method is provided to access the single instance of the class. This method is typically named getInstance().

Here's an example implementation:

```java
/*package whatever //do not write package name here */
import java.io.*;
class Singleton {
    // static class
    private static Singleton instance;
    private Singleton()
    {
        System.out.println("Singleton is Instantiated.");
    }
    public static Singleton getInstance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
    public static void doSomething()
    {
        System.out.println("Something is Done.");
    }
}
```

Initialization Methods:

Singleton classes can be instantiated using different initialization methods:

1. **Early Initialization:** In this method, the *class is initialized whether it is to be used or not*. It simplifies the implementation but consumes memory even if the instance is not needed immediately.

```
// Eager Instantiation
class SingletonEager {
    private static SingletonEager instance = new SingletonEager();

    private SingletonEager() {}

    public static SingletonEager getInstance() {
        return instance;
    }
}
```

2. **Lazy Initialization:** In this method, *the class is initialized only when it is required*, saving memory by avoiding unnecessary instantiation. Lazy initialization is commonly used in scenarios where the creation of the instance is resource-intensive.

```
// Lazy Initialization
class SingletonLazy {
    private static SingletonLazy instance;

    private SingletonLazy() {}

    public static SingletonLazy getInstance() {
        if (instance == null)
            instance = new SingletonLazy();
        return instance;
    }
}
```

There are different methods to implement the Singleton design pattern in Java:

Method 1: Classic Implementation

```
class Singleton {
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

Method 2: Synchronized getInstance()

```
// Thread Synchronized Java implementation of singleton design
pattern
class Singleton {
    private static Singleton obj;
    private Singleton() {}
    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

Method 3: Eager Instantiation

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton {
    private static Singleton obj = new Singleton();
    private Singleton() {}

    public static Singleton getInstance() { return obj; }
}
```

Method 4: Double Checked Locking (Most Efficient)

```
// Double Checked Locking based Java implementation of singleton
design pattern
class Singleton {
    private static volatile Singleton obj = null;
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null) {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj == null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

This method drastically reduces the overhead of calling *the synchronized method every time by using double-checked locking* and the volatile keyword to ensure thread safety.

Singleton Design Pattern in Spring Framework
In the Spring framework, the Singleton design pattern is used extensively. The default scope for a bean in Spring is singleton, meaning that the Spring IoC container creates exactly one instance of the object per container.

The Singleton pattern is also used in the implementation of various Spring components, such as the ApplicationContext and the BeanFactory.

Here's an example of how the Singleton pattern is used in Spring:

```
// Singleton bean in Spring XML configuration
<bean id="userService"
class="com.example.UserService" scope="singleton"/>

// Singleton bean in Java-based configuration
@Configuration
public class AppConfiguration {

    @Bean
    @Scope("singleton") // default scope
    public UserService userService(){
        return new UserService();
    }
}
```

In the above examples, the UserService bean will be created as a singleton instance within the Spring IoC container.

Benefits:

- **Memory Efficiency:** As only one instance of the singleton class is created, it saves memory by reusing the same instance.

- **Global Access:** Provides a global point of access to the single instance, allowing easy access from any part of the application.

- **Reusability:** Ensures reusability as the same singleton object is used repeatedly throughout the application.

Overall, the Singleton design pattern is widely used in Java and Spring applications to ensure that only one instance of a class exists, which can improve performance, save memory, and provide a global point of access to the instance.

References:

1. **Singleton Design Pattern**
   https://www.baeldung.com/spring-boot-singleton-vs-beans
   https://www.javatpoint.com/singleton-design-pattern-in-java
   https://www.softwaretestinghelp.com/design-patterns-in-java/
   https://www.geeksforgeeks.org/singleton-design-pattern/
2. **Singleton Design Pattern in Spring Framework**
   https://www.linkedin.com/pulse/design-patterns-used-spring-framework-abid-anjum/