Mastering Singleton Pattern - Ensuring a Single Instance

The Singleton Pattern ensures that a class has **only one instance** throughout its lifecycle and provides a **global access point** to that instance. It is widely used for managing shared resources, global states, and configurations.

---

1. Eager Initialization

In **Eager Initialization**, the singleton instance is created at the time the class is loaded.

How It Works

1. Create a **static final instance** of the class.
2. Use a **private constructor** to prevent external instantiation.
3. Provide a **public static method** to return the instance.

Steps to Implement

1. Define a private static instance at the class level.
2. Use a private constructor to restrict direct instantiation.
3. Return the instance using a public static method.

Code Example

```
public class EagerSingleton {
    private static final EagerSingleton instance = new
        EagerSingleton();

    private EagerSingleton() {}

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

---

2. Lazy Initialization

In **Lazy Initialization**, the singleton instance is created only when it is first requested.

How It Works

1. Start with a **null instance**.
2. Check if the instance is `null` before creating it.
3. Use a **synchronized method** to ensure thread safety in multi-threaded environments.

Steps to Implement

1. Declare a static instance initialized as `null`.
2. Use a private constructor.
3. Check and initialize the instance in a synchronized method.

Code Example

```
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

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

---

3. Double-Checked Locking

Double-Checked Locking optimizes thread-safe lazy initialization by reducing synchronization overhead.

How It Works

1. Use a **volatile static instance** to ensure visibility.
2. First, check if the instance is `null` without synchronization.
3. Acquire a lock, check again, and then initialize.

Steps to Implement

1. Declare the static instance as `volatile`.
2. Use a double `if` check with synchronized blocks.
3. Return the instance.

Code Example

```
public class DoubleCheckedSingleton {
    private static volatile DoubleCheckedSingleton instance;

    private DoubleCheckedSingleton() {}

    public static DoubleCheckedSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckedSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckedSingleton();
                }
            }
        }
        return instance;
    }
}
```

---

4. Reflection Safe Singleton

To prevent **reflection attacks**, add a guard condition in the constructor to ensure no additional instances are created.

How It Works

1. Throw an exception in the private constructor if an instance already exists.
2. This prevents creating a new instance using reflection.

Code Example

```
public class ReflectionSafeSingleton {
    private static ReflectionSafeSingleton instance;

    private ReflectionSafeSingleton() {
        if (instance != null) {
            throw new IllegalStateException("Instance already
        created!");
        }
    }

    public static ReflectionSafeSingleton getInstance() {
        if (instance == null) {
            instance = new ReflectionSafeSingleton();
```

```
        }
        return instance;
    }
}
```

---

5. Serialization Safe Singleton

Serialization can create multiple instances of a Singleton. Use `readResolve()` to ensure only one instance exists after deserialization.

How It Works

1. Use the `readResolve` method to return the same instance during deserialization.
2. Prevents the creation of a new object.

Code Example

```java
import java.io.Serializable;

public class SerializationSafeSingleton implements Serializable {
    private static final long serialVersionUID = 1L;

    private static final SerializationSafeSingleton instance =
        new SerializationSafeSingleton();

    private SerializationSafeSingleton() {}

    public static SerializationSafeSingleton getInstance() {
        return instance;
    }

    protected Object readResolve() {
        return instance;
    }
}
```

---

6. Enum Singleton

The **Enum Singleton** approach is the simplest and most robust way to implement the Singleton Pattern in Java.

Why Enum Singleton?

1. **Thread-Safe**: Enums are inherently thread-safe.
2. **Serialization Safe**: Enums prevent creating new instances during deserialization.
3. **Reflection Safe**: Enums cannot be instantiated using reflection.

Code Example

```
public enum EnumSingleton {
    INSTANCE;

    public void showMessage() {
        System.out.println("Hello from Enum Singleton!");
    }
}
```

---

7. Singleton Simulator - Testing All Implementations

This simulator tests all Singleton variations to ensure they return the **same instance**.

Code Example

```
public class SingletonSimulator {
    public static void main(String[] args) {
        System.out.println("Testing Singleton Implementations:
        \n");

        // Eager Singleton
        EagerSingleton eager1 = EagerSingleton.getInstance();
        EagerSingleton eager2 = EagerSingleton.getInstance();
        System.out.println("Eager Singleton: " + (eager1 ==
        eager2));

        // Lazy Singleton
        LazySingleton lazy1 = LazySingleton.getInstance();
        LazySingleton lazy2 = LazySingleton.getInstance();
        System.out.println("Lazy Singleton: " + (lazy1 ==
        lazy2));

        // Double-Checked Singleton
        DoubleCheckedSingleton doubleChecked1 =
        DoubleCheckedSingleton.getInstance();
```

```
        DoubleCheckedSingleton doubleChecked2 =
        DoubleCheckedSingleton.getInstance();
        System.out.println("Double-Checked Singleton: " + (doubleChecked1
        == doubleChecked2));

        // Reflection Safe Singleton
        ReflectionSafeSingleton reflection1 =
        ReflectionSafeSingleton.getInstance();
        ReflectionSafeSingleton reflection2 =
        ReflectionSafeSingleton.getInstance();
        System.out.println("Reflection Safe Singleton: " + (reflection1
        == reflection2));

        // Serialization Safe Singleton
        SerializationSafeSingleton serial1 =
        SerializationSafeSingleton.getInstance();
        SerializationSafeSingleton serial2 =
        SerializationSafeSingleton.getInstance();
        System.out.println("Serialization Safe Singleton: " + (serial1
        == serial2));

        // Enum Singleton
        EnumSingleton enum1 = EnumSingleton.INSTANCE;
        EnumSingleton enum2 = EnumSingleton.INSTANCE;
        System.out.println("Enum Singleton: " + (enum1 ==
        enum2));
        enum1.showMessage();
    }
}
```

Summary Table

Implementation	Thread Safety	Serialization Safe	Reflection Safe	Complexity
Eager Initialization	Yes	No	No	Simple
Lazy Initialization	No	No	No	Moderate

Implementation	Thread Safety	Serialization Safe	Reflection Safe	Complexity
Double-Checked Locking	Yes	No	No	Moderate
Reflection Safe Singleton	Yes	No	Yes	Complex
Serialization Safe Singleton	Yes	Yes	No	Moderate
Enum Singleton	Yes	Yes	Yes	Simple

The Singleton Pattern is powerful for resource sharing and global state management. Choose the right implementation based on your application’s needs.