International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

SINGLETON PATTERN IMPLEMENTATIONS AND THREAD SAFTEY CONSIDERATIONS IN MULTITHREADED ENVIRONMENT

Arun Neelan
Independent Researcher
PA, USA
arunneelan@yahoo.co.in

Abstract

The Singleton Pattern remains one of the most widely recognized design patterns in object-oriented programming, offering controlled access to a single instance of a class. While its intent is conceptually simple, the practical implementation of the pattern—particularly in multi-threaded, distributed, and modular environments—introduces a range of technical challenges. This review paper provides a systematic examination of the various strategies for implementing the Singleton Pattern, including classic lazy initialization, thread-safe techniques such as synchronized accessors and double-checked locking, the initialization-on-demand holder idiom, and Enum-based singletons. Each approach is analysed with respect to its benefits, limitations, and performance implications. Additionally, the paper explores advanced considerations such as class loading issues, reflection-based vulnerabilities, serialization concerns, and the use of Singleton within Dependency Injection frameworks. Through this comprehensive analysis, the paper aims to assist software professionals in selecting the most appropriate Singleton implementation tailored to the specific requirements and constraints of their systems. The paper also highlights the pattern's limitations in modern architectures and discusses future trends, such as better integration with Dependency Injection and potential alternatives in distributed or functional contexts, guiding future use and adaptation of the Singleton pattern.

Keywords: Creational Design Patterns, Software Design Patterns, Singleton Pattern, Object-Oriented Design, Software Engineering.

I. INTRODUCTION

In software engineering, design patterns are established solutions to common problems that developers encounter during the development process. These patterns promote best practices, leading to more efficient, maintainable, and scalable code. The Singleton pattern is a well-known Creational design pattern that ensures a class has only one instance and provides a global point of access to it. This is particularly useful in scenarios where coordinated access to a shared resource is essential, such as configuration managers, logging services, caching systems, or thread pools.

Although the Singleton pattern is conceptually simple, it presents many nuances—especially

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

regarding thread safety, performance, testability, and language-specific implementation details. In Java, several approaches to implementing the Singleton pattern have evolved over time, each with its own set of advantages and drawbacks.

II. SINGLETON PATTERN – DEFINITION AND CLASS DIAGRAM

The Singleton Pattern guarantees that a class has only a single instance and offers a global point of access to that instance [1].

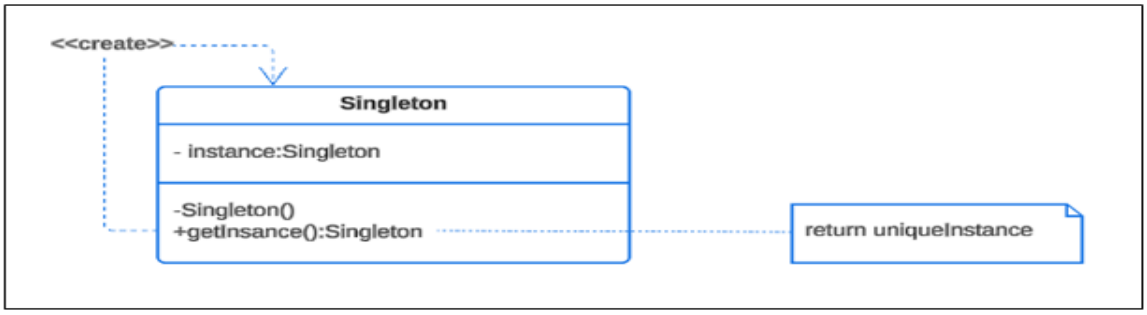

Fig. 1. Singleton Pattern – Class Diagram

III. IMPLEMENTATIONS – CHARACTERISTICS AND EVALUATION

The following sections discuss in detail the different ways of creating Singleton instance along with pros and cons of each in detail.

A. Classic Implementation

The section below illustrates a custom implementation of the Singleton pattern. This implementation ensures that only one instance of the class is created, and it is created only when needed (Lazy Initialization). Lazy Initialization is especially beneficial when the class performs resource-intensive operations during its initialization, as it delays the creation of the instance until it is required.

The drawback of the classic Singleton implementation is that it creates a single instance under normal circumstances, but in a multi-threaded environment, it doesn't ensure thread safety. If multiple threads invoke the getInstance method simultaneously, they may both bypass the null check and create separate instances, thereby violating the Singleton pattern. This leads to inconsistent behavior. Additionally, creating multiple instances can result in unnecessary resource consumption or even errors, especially if the Singleton manages critical resources.

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

```
class Singleton {
    // Step 1: static variable to hold that one instance
    // of this class.
    private static Singleton instance;

    // Step 2: Make the constructor private to prevent instantiation from outside.
    private Singleton() {
        // Private Constructor.
    }

    // Step 3: Global point of access to the instance.
    // Instantiates the class and returns an instance of it.
    // Creates only if it hasn't been created earlier. Returns the previously created one if available. Creates only
when this method is called -- Lazy Initialization.
    public static Singleton getInstance() {
        if (instance == null) {
            // Instantiates by calling its private constructor.
            // The constructor is defined so that it's called onlywithin this class only.
            instance = new Singleton();
        }
        return instance
    }
}

//Class to test the above implementation.
public class TestSingletonClassicImpl {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();

        // Output: true.
        System.out.println(singleton1 == singleton2);
    }
}
```

Listing 1. Classic Singleton Pattern Implementation in Java.

B. Thread Safe Implementations

1. Synchronized Method Approach: This is achieved by making the getInstance method synchronized.

```
class Singleton {
    private static Singleton instance;

    private Singleton() {
    }

    // The synchronized keyword ensures that only one thread can execute the getInstance() method at a time,
providing thread safety by preventing multiple threads from creating separate instances of the Singleton.
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Listing 2. Singleton – Synchronized Method in Java.

While the implementation ensures thread safety, there are several limitations associated with using synchronization in the getInstance method:

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

- Performance Overhead: Thread safety is applied to the entire method, including sections that don't require synchronization. This unnecessary synchronization creates a performance bottleneck, leading to low performance, reduced throughput, and scalability concerns, particularly when the method is called frequently.
- Unnecessary Synchronization Post-Initialization: Thread safety remains enforced even after the instance has been created. However, synchronization is only necessary during the instance's initialization phase. Once the instance is created, synchronization can be avoided, as subsequent access does not pose a thread safety risk. This redundant synchronization adds unneeded overhead after the instance is initialized.

Avoiding method synchronization is recommended, even in applications where performance isn't the primary concern, by selecting a simpler and more efficient alternative solution from the available options. Leaving inefficient code in place can lead to problems in the future, especially when changes occur, such as shifts in the customer base or traffic patterns.

2. Double Checked Locking Approach: This approach aims to optimize performance by reducing synchronization overhead. The key idea is to check if the instance has already been initialized (first check), and only synchronize when necessary (second check inside the synchronized block), to minimize the time spent inside the synchronized section [2].

```
public class Singleton {
   // The volatile keyword ensures thread-safe visibility of a variable, preventing caching issues. Its behavior is JVM-
specific, relying on the Java Memory Model (JMM) for visibility guarantees.
   private static volatile Singleton instance;

   private Singleton() {
   }

   public static Singleton getInstance() {
      // This null check avoids locking once the instance is initialized, reducing synchronization overhead when the
instance is //already created.
      if (instance == null) {
         // Control will enter this block only once for the first thread that attempts to create the instance.
         // After the instance is created, subsequent threads will bypass this block, as the instance is already initialized.
         synchronized (Singleton.class) {
            // This ensures that only one thread can initialize the instance, even if multiple threads are blocked on the
synchronized block.
            if (instance == null) {
               instance = new Singleton();
            }
         }
      }
      return instance;
   }
}
```

Listing 3. Singleton– Double Checking Locking Method in Java

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

3. Initialization-on-demand Holder Idiom (Bill Pugh): This approach leverages a static nested class to encapsulate and initialize the Singleton instance. The inner class is only loaded when it's accessed, and the classloader ensures thread safety when loading the class, so no explicit synchronization is needed.

```
public class Singleton {
    private Singleton() {
    }

    // Static inner class that holds the Singleton instance.
    // The class is only loaded when accessed, ensuring lazy initialization.
    // The Singleton instance is created once and is thread-safe due to JVM's class loading mechanism.
    private static class SingletonHelper {
        // This holds the Singleton instance.
        // It is initialized when the SingletonHelper class is loaded.
        private static final Singleton INSTANCE = new Singleton();
    }

    // Public method to provide access to the Singleton instance.
    // The first time this method is called, the SingletonHelper class is loaded, and the instance of Singleton is
//created.Subsequent calls to this method return the same instance, ensuring that only one instance of Singleton exists.
    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Listing 4. Singleton– Bill Pugh in Java

4. Eager Initialization Approach: In this approach, the instance is created at the time of class loading, rather than when it is needed. This is appropriate when the application always creates and uses an instance, or the overhead of creation and runtime aspects of the Singleton isn't onerous [2, p. 181].

```
public class Singleton {
    private Singleton() {
    }

    // Static inner class that holds the Singleton instance.
    // The class is only loaded when accessed, ensuring lazy initialization.
    // The Singleton instance is created once and is thread-safe due to JVM's class loading mechanism.
    private static class SingletonHelper {
        // This holds the Singleton instance.
        // It is initialized when the SingletonHelper class is loaded.
        private static final Singleton INSTANCE = new Singleton();
    }

    // Public method to provide access to the Singleton instance.
    // The first time this method is called, the SingletonHelper class is loaded, and the instance of Singleton is created.
//Subsequent calls to this method return the same instance, ensuring that only one instance of Singleton exists.
    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Listing 5. Singleton– Eager Initialization Method in Java

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

5. Static Block Initialization: This approach creates the Singleton instance at the time of class loading, but the initialization occurs inside a static block. It is recommended when additional logic or exception handling is required during the initialization process.

```
public class Singleton {
   private static Singleton INSTANCE;

   private Singleton() {
   }

   // Static block to initialize the Singleton instance.
   static {
      try {
         // Instance creation and additional logic can be added here in the block.
         INSTANCE = new Singleton();
      } catch (Exception e) {
         // Handle any initialization exceptions.
         throw new RuntimeException("Error initializing Singleton.", e);
      }
   }

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

Listing 6. Singleton – Static Block Initialization Method in Java

6. Challenges with Custom Thread-Safe Implementation Approaches and Mitigation: Although the thread-safe implementation approaches discussed ensure that only one instance is created in a multi-threaded environment, there are still other ways through which more than one instance can be created.
   a) Class Loading and Multiple Class Loaders: When an app uses multiple class loaders, multiple instances of a Singleton class can be created. To prevent this, ensure the Singleton is loaded by just one class loader, which can be managed through proper control or a shared parent class loader.
   b) Reflection-Based Attacks: The private constructor of the Singleton class can be accessed through Reflection APIs, leading to the creation of multiple instances and violating the pattern. To avoid this issue, an exception can be thrown from the private constructor if an instance already exists. Additionally, if the constructor is invoked before the getInstance method initializes the instance, ensure that it is properly constructed with all necessary state.
   c) Serialization Issues: Serialization can compromise the Singleton pattern by creating a new instance during the deserialization process. To preserve the Singleton property, the readResolve method should be implemented to return the existing instance. Additionally, marking instance variables as transient can prevent them from being

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

serialized. For a deeper understanding of the serialization and deserialization process, including customization options, refer to the Java Specification for the relevant version (e.g., Java SE 8, Java SE 11, etc.) [3] [4], or other official documentation corresponding to the specific release.

```
public class Singleton implements Serializable {
   private static final Singleton INSTANCE = new Singleton();

   private Singleton() {
   }

   public static Singleton getInstance() {
      return INSTANCE;
   }

   // Overriding readResolve to return the existing instance during deserialization.
   // Default behavior will return a new instance.
   private Object readResolve() {
      return INSTANCE; // Ensure the same instance is returned.
   }
}
```

Listing 7. Singleton – Handling Serialization in Java

7. Enum Singleton: This approach is considered the best way to implement a Singleton [5] because in Java, enums are designed with built-in protections against common issues such as reflection attacks, serialization problems, and multi-threading concerns. However, this method cannot be used if the Singleton needs to extend a class other than Enum. Additionally, an Enum in Java can have methods, just like any ordinary class. These methods can define behaviors or provide utility functions specific to the enum constants. For more details on how to define methods within an enum, as well as other features of enums, refer to [6] [7] or the relevant version of the Java specification.

```
public enum Singleton {
   INSTANCE;
}

class TestEnumSingleton {
   public static void main(String[] args) {
      Singleton singleton1 = Singleton.INSTANCE;
      Singleton singleton2 = Singleton.INSTANCE;

      // Output: true
      System.out.println(singleton1 == singleton2);
   }
}
```

Listing 8. Singleton – Enum Method in Java

International Journal of Core Engineering & Management
Volume-8, Issue-02, 2025 ISSN No: 2348-9510

The above sections have detailed the available options for creating a Singleton instance, along with their pros, cons, and mitigation strategies.

C. Dependency Injection Approach

While Singleton ensures shared state and controlled resource usage, it tightly couples the class to its lifecycle management, reducing flexibility and complicating unit testing. Dependency Injection (DI) offers a more robust alternative by delegating the responsibility of managing object lifecycles to an external container, thereby promoting loose coupling and enhancing testability. When integrating the singleton pattern within a DI-based architecture, the DI container is configured to provide a singleton-scoped instance, ensuring that only one object is created and reused wherever the dependency is injected. For example, in a Java application using a DI framework like Spring, a class can be annotated with @Service or @Component and scoped as a singleton (which is the default behavior) to ensure a single instance is maintained throughout the application.

```java
import org.springframework.stereotype.Service;

// This makes the class a Spring-managed singleton by default
@Service
public class MyService {
    public void performTask() {
        System.out.println("Singleton service performing task...");
    }
}


//Example usage
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    private final MyService myService;

    @Autowired
    public MyComponent(MyService myService) {
        this.myService = myService;
    }

    public void execute() {
        myService.performTask();
    }
}
```

Listing 9. Singleton – Dependency Injection in Java

In this example, MyService is registered as a singleton within the Spring container, and MyComponent receives it via constructor-based dependency injection. This method provides the benefits of singleton behavior without manual implementation, improving maintainability and supporting better testing practices.

IV. LIMITATIONS AND CHALLENGES

The Singleton pattern, while effective in ensuring a single instance throughout a system, presents several limitations and challenges. One major issue is its impact on testability, as it introduces tight coupling between classes, making it difficult to mock or replace instances during unit tests. Achieving thread safety is another recurring challenge, particularly when multiple threads concurrently access the Singleton instance. While different implementation strategies, such as eager initialization and double-checked locking, attempt to address these issues, they often come with trade-offs in terms of performance and complexity. Moreover, the global access provided by the Singleton can increase coupling and make it difficult to manage the state across large applications.

Furthermore, while the Dependency Injection (DI) approach is often seen as a solution to some of the Singleton's drawbacks, its use can sometimes cause the pattern to resemble other design patterns, such as the Service Locator or Factory, making it harder to distinguish between them in certain contexts. Despite these challenges, the Singleton pattern remains widely used, and ongoing discussions continue to focus on refining its implementation and exploring its relevance in modern software architecture.

V. FUTURE TRENDS

As software development continues to evolve, the Singleton pattern is expected to adapt to modern challenges and trends. One key trend is its increasing integration with Dependency Injection (DI) frameworks, which can help address issues like tight coupling and poor testability. Additionally, performance optimizations will likely remain a priority, especially in multi-threaded environments, where techniques for improving thread safety and reducing initialization overhead will continue to be refined. The rise of distributed systems and microservices may also push for hybrid patterns that maintain Singleton-like behavior while supporting scalability across services. Furthermore, the growing emphasis on functional programming might inspire new approaches to implementing Singleton behavior without sacrificing immutability. As software architectures become more complex, alternative patterns like Multiton or Service Locator may increasingly complement or replace the traditional Singleton in certain use cases, offering more flexibility in system design.

VI. CONCLUSION

The Singleton Pattern remains a fundamental design approach for managing shared resources in object-oriented systems. While it offers a clear solution, its correct implementation—

especially in multi-threaded environments, can be complex and requires careful consideration of thread safety, performance, and testability. This review has explored various implementations, including lazy initialization, synchronized accessors, double-checked locking, and Enum-based Singletons, each with its own strengths and drawbacks.

Mitigation strategies like proper synchronization and reflection handling are essential, and selecting an approach should depend on the application's specific requirements. As development practices evolve, trends such as integration with Dependency Injection frameworks, improved testability, and support for distributed or functional systems are shaping how the Singleton pattern is applied. By understanding its limitations and the direction it's heading, developers can make more informed, future-ready design decisions.

REFERENCES

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH, 1995.
2. E. Freeman and E. Robson, Head first design patterns: Building Extensible and Maintainable Object-Oriented Software. 2021.
3. Oracle, "Serialization Specification (Java SE 11 & JDK 11)," Oracle. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/specs/serialization/index.html.
4. Oracle, "Serializable (Java SE 11 & JDK 11)," Oracle. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Serializable.html.
5. J. Bloch, Effective java. Addison-Wesley Professional, 2018.
6. "Enum Types (The JavaTM Tutorials > Learning the Java Language > Classes and Objects)." Available: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
7. "The Java® language specification." Available: https://docs.oracle.com/javase/specs/jls/se8/html/index.html