MSC07-J. Prevent multiple instantiations of singleton objects

The singleton design pattern's intent is succinctly described by the seminal work of Gamma and colleagues [Gamma 1995]:

> *Ensure a class only has one instance, and provide a global point of access to it.*

Because there is only one singleton instance, "any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources such as database connections or sockets" [Fox 2001]. Other applications of singletons involve maintaining performance statistics, monitoring and logging system activity, implementing printer spoolers, and even tasks such as ensuring that only one audio file plays at a time. Classes that contain only static methods are good candidates for the Singleton pattern.

The Singleton pattern typically uses a single instance of a class that encloses a private static class field. The instance can be created using *lazy initialization*, which means that the instance is not created when the class loads but when it is first used.

A class that implements the singleton design pattern must prevent multiple instantiations. Relevant techniques include the following:

- Making its constructor private
- Employing lock mechanisms to prevent an initialization routine from being run simultaneously by multiple threads
- Ensuring the class is not serializable
- Ensuring the class cannot be cloned
- Preventing the class from being garbage-collected if it was loaded by a custom class loader

Noncompliant Code Example (Nonprivate Constructor)

This noncompliant code example uses a nonprivate constructor for instantiating a singleton:

```
class MySingleton {
  private static MySingleton instance;

  protected MySingleton() {
    instance = new MySingleton();
  }

  public static synchronized MySingleton getInstance() {
    return instance;
  }
}
```

A malicious subclass may extend the accessibility of the constructor from protected to public, allowing untrusted code to create multiple instances of the singleton. Also, the class field `Instance` has not been declared final.

Compliant Solution (Private Constructor)

This compliant solution reduces the accessibility of the constructor to private and immediately initializes the field `Instance`, allowing it to be declared final. Singleton constructors must be private.

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

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  public static synchronized MySingleton getInstance() {
    return instance;
  }
}
```

The `MySingleton` class need not be declared final because it has a private constructor.

Noncompliant Code Example (Visibility across Threads)

Multiple instances of the `Singleton` class can be created when the getter method is tasked with initializing the singleton when necessary, and the getter method is invoked by two or more threads simultaneously.

```
class MySingleton {
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Lazy initialization
  public static MySingleton getInstance() { // Not synchronized
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

A singleton initializer method in a multithreaded program must employ some form of locking to prevent construction of multiple singleton objects.

Noncompliant Code Example (Inappropriate Synchronization)

Multiple instances can be created even when the singleton construction is encapsulated in a synchronized block, as in this noncompliant code example:

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

The reason multiple instances can be created in this case is that two or more threads may simultaneously see the field `instance` as `null` in the `if` condition and enter the synchronized block one at a time.

Compliant Solution (Synchronized Method)

To address the issue of multiple threads creating more than one instance of the singleton, make `getInstance()` a synchronized method:

```
class MySingleton {
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Lazy initialization
  public static synchronized MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

Compliant Solution (Double-Checked Locking)

Another compliant solution for implementing thread-safe singletons is the correct use of the double-checked locking idiom:

```
class MySingleton {
  private static volatile MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Double-checked locking
  public static MySingleton getInstance() {
    if (instance == null) {
      synchronized (MySingleton.class) {
        if (instance == null) {
          instance = new MySingleton();
        }
      }
    }
    return instance;
  }
}
```

This design pattern is often implemented incorrectly (see LCK10-J. Use a correct form of the double-checked locking idiom for more details on the correct use of the double-checked locking idiom).

Compliant Solution (Initialize-on-Demand Holder Class Idiom)

This compliant solution uses a static inner class to create the singleton instance:

```
class MySingleton {
  static class SingletonHolder {
    static MySingleton instance = new MySingleton();
  }

  public static MySingleton getInstance() {
    return SingletonHolder.instance;
  }
}
```

This approach is known as the *initialize-on-demand holder class idiom* (see LCK10-J. Use a correct form of the double-checked locking idiom for more information).

Noncompliant Code Example (Serializable)

This noncompliant code example implements the `java.io.Serializable` interface, which allows the class to be serialized. Deserialization of the class implies that multiple instances of the singleton can be created.

```
class MySingleton implements Serializable {
  private static final long serialVersionUID = 6825273283542226860L;
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Lazy initialization
  public static synchronized MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

A singleton's constructor cannot install checks to enforce the requirement that the class is instantiated only once because deserialization can bypass the object's constructor.

Noncompliant Code Example (`readResolve()` Method)

Adding a `readResolve()` method that returns the original instance is insufficient to enforce the singleton property. This technique is insecure even when all the fields are declared transient or static.

```
class MySingleton implements Serializable {
  private static final long serialVersionUID = 6825273283542226860L;
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Lazy initialization
  public static synchronized MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }

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

At runtime, an attacker can add a class that reads in a crafted serialized stream:

```
public class Untrusted implements Serializable {
  public static MySingleton captured;
  public MySingleton capture;

  public Untrusted(MySingleton capture) {
    this.capture = capture;
  }

  private void readObject(java.io.ObjectInputStream in)
                         throws Exception {
    in.defaultReadObject();
    captured = capture;
  }
}
```

The crafted stream can be generated by serializing the following class:

```
public final class MySingleton
                   implements java.io.Serializable {
  private static final long serialVersionUID =
      6825273283542226860L;
  public Untrusted untrusted =
      new Untrusted(this); // Additional serial field

  public MySingleton() { }
}
```

Upon deserialization, the field `MySingleton.untrusted` is reconstructed before `MySingleton.readResolve()` is called. Consequently, `Untrusted.captured` is assigned the deserialized instance of the crafted stream instead of `MySingleton.instance`. This issue is pernicious when an attacker can add classes to exploit the singleton guarantee of an existing serializable class.

Noncompliant Code Example (Nontransient Instance Fields)

This serializable noncompliant code example uses a nontransient instance field `str`:

```
class MySingleton implements Serializable {
  private static final long serialVersionUID =
      2787342337386756967L;
  private static MySingleton instance;

  // Nontransient instance field
  private String[] str = {"one", "two", "three"};

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  public void displayStr() {
    System.out.println(Arrays.toString(str));
  }

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

"If a singleton contains a nontransient object reference field, the contents of this field will be deserialized before the singleton's `readResolve` method is run. This allows a carefully crafted stream to 'steal' a reference to the originally deserialized singleton at the time the contents of the object reference field are deserialized" [Bloch 2008].

Compliant Solution (Enumeration Types)

Stateful singleton classes must be nonserializable. As a precautionary measure, classes that are serializable must not save a reference to a singleton object in their nontransient or nonstatic instance variables. This precaution prevents the singleton from being indirectly serialized.

Bloch [Bloch 2008] suggests the use of an enumeration type as a replacement for traditional implementations when serializable singletons are indispensable.

```
public enum MySingleton {
  ; // Empty list of enum values

  private static MySingleton instance;

  // Nontransient field
  private String[] str = {"one", "two", "three"};

  public void displayStr() {
    System.out.println(Arrays.toString(str));
  }
}
```

This approach is functionally equivalent to, but much safer than, commonplace implementations. It both ensures that only one instance of the object exists at any instant and provides the serialization property (because `java.lang.Enum<E>` extends `java.io.Serializable`).

Noncompliant Code Example (Cloneable Singleton)

When the singleton class implements `java.lang.Cloneable` directly or through inheritance, it is possible to create a copy of the singleton by cloning it using the object's `clone()` method. This noncompliant code example shows a singleton that implements the `java.lang.Cloneable` interface.

```
class MySingleton implements Cloneable {
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents
    // instantiation by untrusted callers
  }

  // Lazy initialization
  public static synchronized MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

Compliant Solution (Override `clone()` Method)

To avoid making the singleton class cloneable, do not implement the `Cloneable` interface and do not derive from a class that already implements it.

When the singleton class must indirectly implement the `Cloneable` interface through inheritance, the object's `clone()` method must be overridden with one that throws a `CloneNotSupportedException` exception [Daconta 2003].

```
class MySingleton implements Cloneable {
  private static MySingleton instance;

  private MySingleton() {
    // Private constructor prevents instantiation by untrusted callers
  }

  // Lazy initialization
  public static synchronized MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }

  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

See OBJ07-J. Sensitive classes must not let themselves be copied for more details about preventing misuse of the `clone()` method.

Noncompliant Code Example (Garbage Collection)

A class may be garbage-collected when it is no longer reachable. This behavior can be problematic when the program must maintain the singleton property throughout the entire lifetime of the program.

A static singleton becomes eligible for garbage collection when its class loader becomes eligible for garbage collection. This usually happens when a nonstandard (custom) class loader is used to load the singleton. This noncompliant code example prints different values of the hash code of the singleton object from different scopes:

```
  {
  ClassLoader cl1 = new MyClassLoader();
  Class class1 = cl1.loadClass(MySingleton.class.getName());
  Method classMethod =
      class1.getDeclaredMethod("getInstance", new Class[] { });
  Object singleton = classMethod.invoke(null, new Object[] { });
  System.out.println(singleton.hashCode());
}

ClassLoader cl1 = new MyClassLoader();
Class class1 = cl1.loadClass(MySingleton.class.getName());
Method classMethod =
    class1.getDeclaredMethod("getInstance", new Class[] { });
Object singleton = classMethod.invoke(null, new Object[] { } );
System.out.println(singleton.hashCode());
```

Unknown macro: 'mc' Code that is outside the scope can create another instance of the singleton class even though the requirement was to use only the original instance. Unknown macro: 'mc' Because a singleton instance is associated with the class loader that is used to load it, it is possible to have multiple instances of the same class in the Java Virtual Machine. This situation typically occurs in J2EE containers and applets. Technically, these instances are different classes that are independent of each other. Failure to protect against multiple instances of the singleton may or may not be insecure depending on the specific requirements of the program.

Compliant Solution (Prevent Garbage Collection)

This compliant solution takes into account the garbage-collection issue described previously. A class cannot be garbage-collected until the `ClassLoader` object used to load it becomes eligible for garbage collection. A simple scheme to prevent garbage collection is to ensure that there is a direct or indirect reference from a live thread to the singleton object that must be preserved.

This compliant solution demonstrates this technique. It prints a consistent hash code across all scopes. It uses the `ObjectPreserver` class [Grand 2002] described in TSM02-J. Do not use background threads during class initialization.

```
 {
  ClassLoader cl1 = new MyClassLoader();
  Class class1 = cl1.loadClass(MySingleton.class.getName());
  Method classMethod =
      class1.getDeclaredMethod("getInstance", new Class[] { });
  Object singleton = classMethod.invoke(null, new Object[] { });
  ObjectPreserver.preserveObject(singleton); // Preserve the object
  System.out.println(singleton.hashCode());
}

ClassLoader cl1 = new MyClassLoader();
Class class1 = cl1.loadClass(MySingleton.class.getName());
Method classMethod =
    class1.getDeclaredMethod("getInstance", new Class[] { });
// Retrieve the preserved object
Object singleton = ObjectPreserver.getObject();
System.out.println(singleton.hashCode());
```

Risk Assessment

Using improper forms of the Singleton design pattern may lead to creation of multiple instances of the singleton and violate the expected contract of the class.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC07-J	Low	Unlikely	Medium	P2	L3

Automated Detection

Tool	Version	Checker	Description

The Checker Framework	2.1.3	**Linear Checker**	Control aliasing and prevent re-use (see Chapter 19)
Coverity	7.5	**SINGLETON_RACE** **UNSAFE_LAZY_INIT** **FB.LI_LAZY_INIT_UPDATE_STATIC** **FB.LI_LAZY_INIT_STATIC**	Implemented
Parasoft Jtest	9.5	**TRS.ILI**	Implemented

Related Guidelines

MITRE CWE	CWE-543, Use of Singleton Pattern without Synchronization in a Multithreaded Context

Bibliography

[Bloch 2008]	Item 3, "Enforce the Singleton Property with a Private Constructor or an `enum` Type" Item 77, "For Instance Control, Prefer `enum` Types to `readResolve`"
[Daconta 2003]	Item 15, "Avoiding Singleton Pitfalls"
[Darwin 2004]	Section 9.10, "Enforcing the Singleton Pattern"
[Fox 2001]	When Is a Singleton Not a Singleton?
[Gamma 1995]	Singleton
[Grand 2002]	Chapter 5, "Creational Patterns," section "Singleton"
[JLS 2015]	Chapter 17, "Threads and Locks"