LCK01-J. Do not synchronize on objects that may be reused

Misuse of synchronization primitives is a common source of concurrency issues. Synchronizing on objects that may be reused can result in deadlock and nondeterministic behavior. Consequently, programs must never synchronize on objects that may be reused.

Noncompliant Code Example (`Boolean` Lock Object)

This noncompliant code example synchronizes on a `Boolean` lock object.

```
private final Boolean initialized = Boolean.FALSE;

public void doSomething() {
  synchronized (initialized) {
    // ...
  }
}
```

The `Boolean` type is unsuitable for locking purposes because it allows only two values: true and false. Boolean literals containing the same value share unique instances of the `Boolean` class in the Java Virtual Machine (JVM). In this example, `initialized` refers to the instance corresponding to the value `Boolean.FALSE`. If any other code were to inadvertently synchronize on a `Boolean` literal with this value, the lock instance would be reused and the system could become unresponsive or could deadlock.

Noncompliant Code Example (Boxed Primitive)

This noncompliant code example locks on a boxed `Integer` object.

```
private int count = 0;
private final Integer Lock = count; // Boxed primitive Lock is shared

public void doSomething() {
  synchronized (Lock) {
    count++;
    // ...
  }
}
```

Boxed types may use the same instance for a range of integer values; consequently, they suffer from the same reuse problem as `Boolean` constants. The wrapper object are reused when the value can be represented as a byte; JVM implementations are also permitted to reuse wrapper objects for larger ranges of values. While use of the intrinsic lock associated with the boxed `Integer` wrapper object is insecure; instances of the `Integer` object constructed using the `new` operator (`new Integer(value)`) are unique and not reused. In general, locks on any data type that contains a boxed value are insecure.

Compliant Solution (Integer)

This compliant solution locks on a nonboxed `Integer`, using a variant of the private lock object idiom. The `doSomething()` method synchronizes using the intrinsic lock of the `Integer` instance, `Lock`.

```
private int count = 0;
private final Integer Lock = new Integer(count);

public void doSomething() {
  synchronized (Lock) {
    count++;
    // ...
  }
}
```

When explicitly constructed, an `Integer` object has a unique reference and its own intrinsic lock that is distinct not only from other `Integer` objects, but also from boxed integers that have the same value. While this is an acceptable solution, it can cause maintenance problems because developers can incorrectly assume that boxed integers are also appropriate lock objects. A more appropriate solution is to synchronize on a private final lock object as described in the final compliant solution for this rule.

Noncompliant Code Example (Interned `String` Object)

This noncompliant code example locks on an interned `String` object.

```
private final String lock = new String("LOCK").intern();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

According to the Java API class `java.lang.String` documentation [API 2006]:

> *When the `intern()` method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.*

Consequently, an interned `String` object behaves like a global variable in the JVM. As demonstrated in this noncompliant code example, even when every instance of an object maintains its own `lock` field, the fields all refer to a common `String` constant. Locking on `String` constants has the same reuse problem as locking on `Boolean` constants.

Additionally, hostile code from any other package can exploit this vulnerability, if the class is accessible. See rule LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code for more information.

Noncompliant Code Example (`String` Literal)

This noncompliant code example locks on a final `String` literal.

```
// This bug was found in jetty-6.1.3 BoundedThreadPool
private final String lock = "LOCK";

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

`String` literals are constant and are automatically interned. Consequently, this example suffers from the same pitfalls as the preceding noncompliant code example.

Compliant Solution (`String` Instance)

This compliant solution locks on a noninterned `String` instance.

```
private final String lock = new String("LOCK");

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

A `String` instance differs from a `String` literal. The instance has a unique reference and its own intrinsic lock that is distinct from other `String` object instances or literals. Nevertheless, a better approach is to synchronize on a private final lock object, as shown in the following compliant solution.

Compliant Solution (Private Final Lock `Object`)

This compliant solution synchronizes on a private final lock object. This is one of the few cases in which a `java.lang.Object` instance is useful.

```
private final Object lock = new Object();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

For more information on using an `Object` as a lock, see rule LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code.

Risk Assessment

A significant number of concurrency vulnerabilities arise from locking on the wrong kind of object. It is important to consider the properties of the lock object rather than simply scavenging for objects on which to synchronize.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
LCK01-J	medium	probable	medium	**P8**	**L2**

Automated Detection

Some static analysis tools can detect violations of this rule.

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Lock Checker**	Concurrency and lock errors (see Chapter 6)
CodeSonar	5.1p0	**FB.MT_CORRECTNESS.DL_SYNCHRONIZATION_ON_BOOLEAN** **FB.MT_CORRECTNESS.DL_SYNCHRONIZATION_ON_BOXED_PRIMITIVE** **FB.MT_CORRECTNESS.DL_SYNCHRONIZATION_ON_SHARED_CONSTANT**	Synchronization on Boolean Synchronization on boxed primitive Synchronization on interned String
Parasoft Jtest	10.3	TRS.SCS	Implemented
SonarQube	6.7	**S1860**	
ThreadSafe	1.3	CCE_CC_REUSEDOBJ_SYNC	Implemented

Bibliography

[API 2006]	Class String, Collections
[Findbugs 2008]	
[Miller 2009]	Locking
[Pugh 2008]	Synchronization
[Tutorials 2008]	Wrapper Implementations