**Exclusion 🡪 Synchronization 🡪 Statics and Singleton**

**2.2.4 Statics and Singletons**

As described in the *Design Patterns* book, a Singleton class intentionally supports only one instance.

It is convenient to declare that single instance as a static, in which case both class and instance

methods may use the same lock.

Here is one way to define a fully synchronized singleton class that postpones construction of the

instance until it is first accessed via the instance method. This class represents a counter that

could be used to assign global sequence numbers to objects, transactions, messages, etc., across

different classes in an application. (Just to illustrate computation during initialization, the initial value

is set to a randomly chosen number with at least 262 positive successors.)

class LazySingletonCounter {

private final long initial;

private long count;

private LazySingletonCounter() {

initial = Math.abs(new java.util.Random().nextLong() / 2);

count = initial;

}

private static LazySingletonCounter s = null;

private static final Object classLock =

LazySingletonCounter.class;

public static LazySingletonCounter instance() {

synchronized(classLock) {

if (s == null)

s = new LazySingletonCounter();

return s;

}

}

public long next() {

synchronized(classLock) { return count++; }

}

public void reset() {

synchronized(classLock) { count = initial; }

}

}

The locking mechanics seen here (or any of several minor variants) prevent situations in which two

different threads invoke the instance method at about the same time, causing two instances to be

created. Only one of these instances would be bound to s and returned the next time instance is

invoked. As discussed in § 2.4.1, in a few cases this and other intentional semantic weakenings might

be acceptable; in most cases, however, this would be a serious error.

An easier way to avoid this kind of error is to avoid lazy initialization. Because JVMs perform

dynamic loading of classes, there is usually no need to support lazy initialization of singletons. A

static field is not initialized until the class is loaded at runtime. While there are no guarantees

about exactly when a class will be loaded (beyond that it will be loaded by the time it is accessed by

executing code), full initialization of statics is less likely to impose significant start-up overhead than

in most other languages. So, unless initialization is both very expensive and rarely needed, it is usually

preferable to take the simpler approach of declaring a singleton as a static final field. For

example:

class EagerSingletonCounter {

private final long initial;

private long count;

private EagerSingletonCounter() {

initial = Math.abs(new java.util.Random().nextLong() / 2);

count = initial;

}

private static final EagerSingletonCounter s =

new EagerSingletonCounter();

public static EagerSingletonCounter instance() { return s; }

public synchronized long next() { return count++; }

public synchronized void reset() { count = initial; }

}

Simpler yet, if there is no compelling reason to rely on instances, you can instead define and use a

version with all static methods, as in:

class StaticCounter {

private static final long initial =

Math.abs(new java.util.Random().nextLong() / 2);

private static long count = initial;

private StaticCounter() { } // disable instance construction

public static synchronized long next() { return count++; }

public static synchronized void reset() { count = initial; }

}

Also, consider using ThreadLocal (see § 2.3.2) rather than a Singleton in situations where it is

more appropriate to create one instance of a class per thread than one instance per program.