CON03-C. Ensure visibility when accessing shared variables

Reading a shared primitive variable in one thread may not yield the value of the most recent write to the variable from another thread. Consequently, the thread may observe a stale value of the shared variable. To ensure the visibility of the most recent update, the write to the variable must *happen before* the read (C Standard, subclause 5.1.2.4, paragraph 18 [ISO/IEC 9899:2011]). Atomic operations—other than relaxed atomic operations—trivially satisfy the happens before relationship. Where atomic operations are inappropriate, protecting both reads and writes with a mutex also satisfies the happens before relationship.

********** Text below this note not yet converted from Java to C! **********

Noncompliant Code Example (Non-volatile Flag)

This noncompliant code example uses a `shutdown()` method to set the non-volatile `done` flag that is checked in the `run()` method.

```
final class ControlledStop implements Runnable {
  private boolean done = false;

  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

If one thread invokes the `shutdown()` method to set the flag, a second thread might not observe that change. Consequently, the second thread might observe that `done` is still `false` and incorrectly invoke the `sleep()` method. Compilers and just-in-time compilers (JITs) are allowed to optimize the code when they determine that the value of `done` is never modified by the same thread, resulting in an infinite loop.

Compliant Solution (`Volatile`)

In this compliant solution, the `done` flag is declared volatile to ensure that writes are visible to other threads.

```
final class ControlledStop implements Runnable {
  private volatile boolean done = false;

  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

Compliant Solution (`AtomicBoolean`)

In this compliant solution, the `done` flag is declared to be of type `java.util.concurrent.atomic.AtomicBoolean`. Atomic types also guarantee that writes are visible to other threads.

```
final class ControlledStop implements Runnable {
  private final AtomicBoolean done = new AtomicBoolean(false);

  @Override public void run() {
    while (!done.get()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public void shutdown() {
    done.set(true);
  }
}
```

Compliant Solution (`synchronized`)

This compliant solution uses the intrinsic lock of the `Class` object to ensure that updates are visible to other threads.

```
final class ControlledStop implements Runnable {
  private boolean done = false;

  @Override public void run() {
    while (!isDone()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); // Do something
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); // Reset interrupted status
      }
    }
  }

  public synchronized boolean isDone() {
    return done;
  }

  public synchronized void shutdown() {
    done = true;
  }
}
```

While this is an acceptable compliant solution, intrinsic locks cause threads to block and may introduce contention. On the other hand, volatile-qualified shared variables do not block. Excessive synchronization can also make the program prone to deadlock.

Synchronization is a more secure alternative in situations where the `volatile` keyword or a `java.util.concurrent.atomic.Atomic*` field is inappropriate, such as when a variable's new value depends on its current value. See rule VNA02-J. Ensure that compound operations on shared variables are atomic for more information.

Compliance with rule LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code can reduce the likelihood of misuse by ensuring that untrusted callers cannot access the lock object.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON03-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04		Supported, but no explicit checker