VNA00-J. Ensure visibility when accessing shared primitive 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, either the variable must be declared volatile or the reads and writes must be synchronized.

Declaring a shared variable volatile guarantees visibility in a thread-safe manner only when both of the following conditions are met:

- A write to a variable is independent from its current value.
- A write to a variable is independent from the result of any nonatomic compound operations involving reads and writes of other variables (see VNA 02-J. Ensure that compound operations on shared variables are atomic for more information).

The first condition can be relaxed when you can be sure that only one thread will ever update the value of the variable [Goetz 2006]. However, code that relies on a single-thread confinement is error prone and difficult to maintain. This design approach is permitted under this rule but is discouraged.

Synchronizing the code makes it easier to reason about its behavior and is frequently more secure than simply using the volatile keyword. However, synchronization has somewhat higher performance overhead and can result in thread contention and deadlocks when used excessively.

Declaring a variable volatile or correctly synchronizing the code guarantees that 64-bit primitive long and double variables are accessed atomically. For more information on sharing those variables among multiple threads, see VNA05-J. Ensure atomicity when reading and writing 64-bit values.

Noncompliant Code Example (Non-volatile Flag)

This noncompliant code example uses a shutdown() method to set the nonvolatile done flag that is checked in the run() method:

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

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

Although this compliant solution is acceptable, 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 VNA02-J. Ensure that compound operations on shared variables are atomic for more information).

Compliance with 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.

Exceptions

VNA00-J-EX0: Class objects are created by the virtual machine; their initialization always precedes any subsequent use. Consequently, cross-thread visibility of Class objects is already assured by default.

Risk Assessment

Failing to ensure the visibility of a shared primitive variable may result in a thread observing a stale value of the variable.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>VNA00-J</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

Some static analysis tools are capable of detecting violations of this rule.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>4.2.0</td>
<td></td>
<td>Not Implemented</td>
</tr>
<tr>
<td>FindBugs</td>
<td>2.0.1</td>
<td></td>
<td>Not Implemented</td>
</tr>
<tr>
<td>Parasoft Jtest</td>
<td>10.3</td>
<td>TRS.LORD, TRS.MRAV</td>
<td>Implemented</td>
</tr>
<tr>
<td>PMD</td>
<td>5.0.0</td>
<td></td>
<td>Not Implemented</td>
</tr>
<tr>
<td>Fortify</td>
<td></td>
<td></td>
<td>Not Implemented</td>
</tr>
<tr>
<td>Coverity</td>
<td>7.5</td>
<td></td>
<td>Implemented</td>
</tr>
<tr>
<td>ThreadSafe</td>
<td>1.3</td>
<td>CCE_SL_INCONSISTENT, CCE_CC_CALLBACK_ACCESS, CCE_SL_MIXED, CCE_SL_INCONSISTENT_COL, CCE_SL_MIXED_COL, CCE_CC_UNSAFE_CONTENT, CCE_FF_VOLATILE</td>
<td>Implemented</td>
</tr>
</tbody>
</table>
Related Guidelines

<table>
<thead>
<tr>
<th>MITRE CWE</th>
<th>CWE-413, Improper Resource Locking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CWE-567, Unsynchronized Access to Shared Data in a Multithreaded Context</td>
</tr>
<tr>
<td></td>
<td>CWE-667, Improper Locking</td>
</tr>
</tbody>
</table>

Bibliography

| [Bloch 2008] | Item 66, "Synchronize Access to Shared Mutable Data" |
| [Goetz 2006] | Section 3.4.2, "Example: Using Volatile to Publish Immutable Objects" |
| [JLS 2015]  | Chapter 17, "Threads and Locks" |
|             | §17.4.3, "Programs and Program Order" |
|             | §17.4.5, "Happens-before Order" |
|             | §17.4.8, "Executions and Causality Requirements" |
| [JPL 2006]  | Section 14.10.3, "The Happens-Before Relationship" |