Always invoke wait() and await() methods inside a loop

The `Object.wait()` method temporarily cedes possession of a lock so that other threads that may be requesting the lock can proceed. `Object.wait()` must always be called from a `synchronized` block or method. The waiting thread resumes execution only after it has been notified, generally as the result of the invocation of the `notify()` or `notifyAll()` method by some other thread. The `wait()` method must be invoked from a loop that checks whether a `condition predicate` holds. Note that a condition predicate is the negation of the condition expression in the loop. For example, the condition predicate for removing an element from a vector is `!isEmpty()`, whereas the condition expression for the while loop condition is `isEmpty()`. Following is the correct way to invoke the `wait()` method when the vector is empty.

```java
private Vector vector;
// ...
public void consumeElement() throws InterruptedException {
    synchronized (vector) {
        while (vector.isEmpty()) {
            vector.wait();
        }
        // Resume when condition holds
    }
}
```

The notification mechanism notifies the waiting thread and allows it to check its condition predicate. The invocation of `notify()` or `notifyAll()` in another thread cannot precisely determine which waiting thread will be resumed. Condition predicate statements allow notified threads to determine whether they should resume upon receiving the notification. Condition predicates are also useful when a thread is required to block until a condition becomes true, for example, when waiting for data to arrive on an input stream before reading the data.

Both safety and liveness are concerns when using the `wait/notify` mechanism. The safety property requires that all objects maintain consistent states in a multithreaded environment [Lea 2000]. The liveness property requires that every operation or method invocation execute to completion without interruption.

To guarantee liveness, programs must test the `while` loop condition before invoking the `wait()` method. This early test checks whether another thread has already satisfied the condition predicate and sent a notification. Invoking the `wait()` method after the notification has been sent results in indefinite blocking.

To guarantee safety, programs must test the `while` loop condition after returning from the `wait()` method. Although `wait()` is intended to block indefinitely until a notification is received, it still must be encased within a loop to prevent the following vulnerabilities [Bloch 2001]:

- **Thread in the middle**: A third thread can acquire the lock on the shared object during the interval between a notification being sent and the receiving thread resuming execution. This third thread can change the state of the object, leaving it inconsistent. This is a time-of-check, time-of-use (TOCTOU) race condition.
- **Malicious notification**: A random or malicious notification can be received when the condition predicate is false. Such a notification would cancel the `wait()` method.
- **Misdelivered notification**: The order in which threads execute after receipt of a `notifyAll()` signal is unspecified. Consequently, an unrelated thread could start executing and discover that its condition predicate is satisfied. Consequently, it could resume execution despite being required to remain dormant.
- **Spurious wakeups**: Certain Java Virtual Machine (JVM) implementations are vulnerable to spurious wakeups that result in waiting threads waking up even without a notification [API 2014].

For these reasons, programs must check the condition predicate after the `wait()` method returns. A `while` loop is the best choice for checking the condition predicate both before and after invoking `wait()`.

Similarly, the `await()` method of the `Condition` interface also must be invoked inside a loop. According to the Java API [API 2014], `Interface Condition`.

When waiting upon a `Condition`, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a `Condition` should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

New code should use the `java.util.concurrent.locks` concurrency utilities in place of the `wait/notify` mechanism. However, legacy code that complies with the other requirements of this rule is permitted to depend on the `wait/notify` mechanism.

Noncompliant Code Example

This noncompliant code example invokes the `wait()` method inside a traditional `if` block and fails to check the postcondition after the notification is received. If the notification were accidental or malicious, the thread could wake up prematurely.
synchronized (object) {
    if (!<condition does not hold>) {
        object.wait();
    }
    // Proceed when condition holds
}

Compliant Solution

This compliant solution calls the `wait()` method from within a `while` loop to check the condition both before and after the call to `wait()`:

synchronized (object) {
    while (!<condition does not hold>) {
        object.wait();
    }
    // Proceed when condition holds
}

Invocations of the `java.util.concurrent.locks.Condition.await()` method also must be enclosed in a similar loop.

Risk Assessment

Failure to encase the `wait()` or `await()` methods inside a `while` loop can lead to indefinite blocking and denial of service (DoS).

<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>THI03-J</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft Jtest</td>
<td>2023.1</td>
<td>CERT.THI03.UWIL</td>
<td>Call 'wait()' and 'await()' only inside a loop that tests the liveness condition</td>
</tr>
<tr>
<td>SonarQube</td>
<td>9.9</td>
<td>S2274</td>
<td>&quot;Object.wait(...)&quot;) and &quot;Condition.await(...)&quot;) should be called inside a &quot;while&quot; loop</td>
</tr>
</tbody>
</table>

Bibliography

| [API 2014] | Class Object Interface Condition |
| [Bloch 2001] | Item 50, "Never Invoke wait Outside a Loop" |
| [Goetz 2006] | Section 14.2, "Using Condition Queues" |
| [Lea 2000] | Section 1.3.2, "Liveness" Section 3.2.2, "Monitor Mechanics" |