

Item 69: Prefer concurrency utilities to wait and notify

The first edition of this book devoted an item to the correct use of `wait` and `notify` (Bloch01, Item 50). Its advice is still valid and is summarized at end of this item, but this advice is far less important than it once was. This is because there is far less reason to use `wait` and `notify`. As of release 1.5, the Java platform provides higher-level concurrency utilities that do the sorts of things you formerly had to hand-code atop `wait` and `notify`. **Given the difficulty of using `wait` and `notify` correctly, you should use the higher-level concurrency utilities instead.**

The higher-level utilities in `java.util.concurrent` fall into three categories: the Executor Framework, which was covered only briefly in Item 68; concurrent collections; and synchronizers. Concurrent collections and synchronizers are covered briefly in this item.

The concurrent collections provide high-performance concurrent implementations of standard collection interfaces such as `List`, `Queue`, and `Map`. To provide high concurrency, these implementations manage their own synchronization internally (Item 67). Therefore, **it is impossible to exclude concurrent activity from a concurrent collection; locking it will have no effect** but to slow the program.

This means that clients can't atomically compose method invocations on concurrent collections. Some of the collection interfaces have therefore been extended with *state-dependent modify operations*, which combine several primitives into a single atomic operation. For example, `ConcurrentMap` extends `Map` and adds several methods, including `putIfAbsent(key, value)`, which inserts a mapping for a key if none was present and returns the previous value associated with the key, or `null` if there was none. This makes it easy to implement thread-safe canonicalizing maps. For example, this method simulates the behavior of `String.intern`:

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<String, String>();

public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

In fact, you can do even better. `ConcurrentHashMap` is optimized for retrieval operations, such as `get`. Therefore, it is worth invoking `get` initially and calling `putIfAbsent` only if `get` indicates that it is necessary:

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

Besides offering excellent concurrency, `ConcurrentHashMap` is very fast. On my machine the optimized `intern` method above is over six times faster than `String.intern` (but keep in mind that `String.intern` must use some sort of weak reference to keep from leaking memory over time). Unless you have a compelling reason to do otherwise, **use `ConcurrentHashMap` in preference to `Collections.synchronizedMap` or `Hashtable`**. Simply replacing old-style synchronized maps with concurrent maps can dramatically increase the performance of concurrent applications. More generally, use concurrent collections in preference to externally synchronized collections.

Some of the collection interfaces have been extended with *blocking operations*, which wait (or *block*) until they can be successfully performed. For example, `BlockingQueue` extends `Queue` and adds several methods, including `take`, which removes and returns the head element from the queue, waiting if the queue is empty. This allows blocking queues to be used for *work queues* (also known as *producer-consumer queues*), to which one or more *producer threads* enqueue work items and from which one or more *consumer threads* dequeue and process items as they become available. As you'd expect, most `ExecutorService` implementations, including `ThreadPoolExecutor`, use a `BlockingQueue` (Item 68).

Synchronizers are objects that enable threads to wait for one another, allowing them to coordinate their activities. The most commonly used synchronizers are `CountDownLatch` and `Semaphore`. Less commonly used are `CyclicBarrier` and `Exchanger`.

Countdown latches are single-use barriers that allow one or more threads to wait for one or more other threads to do something. The sole constructor for `CountDownLatch` takes an `int` that is the number of times the `countDown` method must be invoked on the latch before all waiting threads are allowed to proceed.

It is surprisingly easy to build useful things atop this simple primitive. For example, suppose you want to build a simple framework for timing the concurrent execution of an action. This framework consists of a single method that takes an executor to execute the action, a concurrency level representing the number of

actions to be executed concurrently, and a runnable representing the action. All of the worker threads ready themselves to run the action before the timer thread starts the clock (this is necessary to get an accurate timing). When the last worker thread is ready to run the action, the timer thread “fires the starting gun,” allowing the worker threads to perform the action. As soon as the last worker thread finishes performing the action, the timer thread stops the clock. Implementing this logic directly on top of wait and notify would be messy to say the least, but it is surprisingly straightforward on top of CountdownLatch:

```
// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
    final Runnable action) throws InterruptedException {
    final CountdownLatch ready = new CountdownLatch(concurrency);
    final CountdownLatch start = new CountdownLatch(1);
    final CountdownLatch done = new CountdownLatch(concurrency);
    for (int i = 0; i < concurrency; i++) {
        executor.execute(new Runnable() {
            public void run() {
                ready.countDown(); // Tell timer we're ready
                try {
                    start.await(); // Wait till peers are ready
                    action.run();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    done.countDown(); // Tell timer we're done
                }
            }
        });
    }
    ready.await(); // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await(); // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```

Note that the method uses three countdown latches. The first, `ready`, is used by worker threads to tell the timer thread when they're ready. The worker threads then wait on the second latch, which is `start`. When the last worker thread invokes `ready.countDown`, the timer thread records the start time and invokes `start.countDown`, allowing all of the worker threads to proceed. Then the timer thread waits on the third latch, `done`, until the last of the worker threads finishes running the action and calls `done.countDown`. As soon as this happens, the timer thread awakens and records the end time.

A few more details bear noting. The executor that is passed to the `time` method must allow for the creation of at least as many threads as the given concurrency level, or the test will never complete. This is known as a *thread starvation deadlock* [Goetz06 8.1.1]. If a worker thread catches an `InterruptedException`, it reasserts the interrupt using the idiom `Thread.currentThread().interrupt()` and returns from its `run` method. This allows the executor to deal with the interrupt as it sees fit, which is as it should be. Finally, note that `System.nanoTime` is used to time the activity rather than `System.currentTimeMillis`. **For interval timing, always use `System.nanoTime` in preference to `System.currentTimeMillis`.** `System.nanoTime` is both more accurate and more precise, and it is not affected by adjustments to the system's real-time clock.

This item only scratches the surface of the concurrency utilities. For example, the three countdown latches in the previous example can be replaced by a single cyclic barrier. The resulting code is even more concise, but it is more difficult to understand. For more information, see *Java Concurrency in Practice* [Goetz06].

While you should always use the concurrency utilities in preference to `wait` and `notify`, you might have to maintain legacy code that uses `wait` and `notify`. The `wait` method is used to make a thread wait for some condition. It must be invoked inside a synchronized region that locks the object on which it is invoked. Here is the standard idiom for using the `wait` method:

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(); // (Releases lock, and reacquires on wakeup)

    ... // Perform action appropriate to condition
}
```

Always use the wait loop idiom to invoke the `wait` method; never invoke it outside of a loop. The loop serves to test the condition before and after waiting.

Testing the condition before waiting and skipping the wait if the condition already holds are necessary to ensure liveness. If the condition already holds and the `notify` (or `notifyAll`) method has already been invoked before a thread waits, there is no guarantee that the thread will *ever* wake from the wait.

Testing the condition after waiting and waiting again if the condition does not hold are necessary to ensure safety. If the thread proceeds with the action when the condition does not hold, it can destroy the invariant guarded by the lock. There are several reasons a thread might wake up when the condition does not hold:

- Another thread could have obtained the lock and changed the guarded state between the time a thread invoked `notify` and the time the waiting thread woke.
- Another thread could have invoked `notify` accidentally or maliciously when the condition did not hold. Classes expose themselves to this sort of mischief by waiting on publicly accessible objects. Any `wait` contained in a synchronized method of a publicly accessible object is susceptible to this problem.
- The notifying thread could be overly “generous” in waking waiting threads. For example, the notifying thread might invoke `notifyAll` even if only some of the waiting threads have their condition satisfied.
- The waiting thread could (rarely) wake up in the absence of a `notify`. This is known as a *spurious wakeup* [Posix, 11.4.3.6.1; JavaSE6].

A related issue is whether you should use `notify` or `notifyAll` to wake waiting threads. (Recall that `notify` wakes a single waiting thread, assuming such a thread exists, and `notifyAll` wakes all waiting threads.) It is often said that you should *always* use `notifyAll`. This is reasonable, conservative advice. It will always yield correct results because it guarantees that you’ll wake the threads that need to be awakened. You may wake some other threads, too, but this won’t affect the correctness of your program. These threads will check the condition for which they’re waiting and, finding it false, will continue waiting.

As an optimization, you may choose to invoke `notify` instead of `notifyAll` if all threads that could be in the wait-set are waiting for the same condition and only one thread at a time can benefit from the condition becoming true.

Even if these conditions appear true, there may be cause to use `notifyAll` in place of `notify`. Just as placing the `wait` invocation in a loop protects against accidental or malicious notifications on a publicly accessible object, using `notifyAll` in place of `notify` protects against accidental or malicious waits by an unrelated thread. Such waits could otherwise “swallow” a critical notification, leaving its intended recipient waiting indefinitely.

In summary, using `wait` and `notify` directly is like programming in “concurrency assembly language,” as compared to the higher-level language provided by `java.util.concurrent`. **There is seldom, if ever, a reason to use `wait` and `notify` in new code.** If you maintain code that uses `wait` and `notify`, make sure that it always invokes `wait` from within a `while` loop using the standard idiom. The `notifyAll` method should generally be used in preference to `notify`. If `notify` is used, great care must be taken to ensure liveness.