# JVM concurrency: To block, or not to block?

## Compare blocking and nonblocking approaches to handling asynchronous events in Java 8

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

22 July 2014

The `CompletableFuture` class added in Java™ 8 gives you new ways to handle the completion of asynchronous processing, including nonblocking ways to compose and combine events. This article helps you understand the differences between blocking and nonblocking approaches to handling completions and gives you some reasons to prefer nonblocking approaches.

View more content in this series

Asynchronous event handling is crucial in any concurrent application. The sources of the events can be separate computational tasks, I/O operations, or interactions with external systems. Whatever the source, the application code must keep track of the events and coordinate actions taken in response to the events. Two basic approaches to asynchronous event handling are available to a Java application: The application can have a coordination thread wait for the event and then take action, or the event can perform an action (which often takes the form of executing code that the application supplies) directly when the event completes. Having a thread wait for an event is called a *blocking* approach. Having the event perform an action without a thread explicitly waiting for it is called a *nonblocking* approach.

The old `java.util.concurrent.Future` class provides a simple way to handle the expected completion of an event, but only by means of either polling or waiting for completion. The `java.util.concurrent.CompletableFuture` class added in Java 8 extends this capability with a range of methods for composing or handling events. (Read the previous article in this series, "Java 8 concurrency basics," for an introduction to `CompletableFuture`.) In particular, `CompletableFuture` gives you standard techniques for executing application code when an event completes, including various ways to combine tasks (as represented by futures). This combination technique makes it easy (or at least easier than in the past) to write nonblocking code for handling events. In this article, you'll see how to use `CompletableFuture` for both blocking and nonblocking event handling.

And you'll get some pointers on why the nonblocking approach can be worth some extra effort. (Get the full sample code from the author's GitHub repository.)

### *Blocking* and *nonblocking*

In computing, the terms *blocking* and *nonblocking* are often used in somewhat different ways, depending on context. Nonblocking algorithms for shared data structures, for instance, don't require threads to wait for access to the data structures. In nonblocking I/O, an application thread can start an I/O operation and then go off and do other things while the operation takes place asynchronously. In this article, *nonblocking* refers to the completion of an event performing an action that would otherwise require a waiting thread. The common concept among these usages is that a blocking operation requires a thread to wait for something, and a nonblocking operation doesn't.

## Composing events

Waiting for a completion is straightforward: You have a thread wait on the event, and when the thread resumes, you know that the event is complete. If your thread has other things to do in the meantime it can go off and do them before waiting. The thread can even use a polling approach whereby it interrupts its other activities to check if the event has completed. But the basic principle is the same: At the point when you need the result of the event, you park the thread so that it waits for the event to complete.

### About this series

Now that multicore systems are ubiquitous, concurrent programming must be applied more widely than ever before. But concurrency can be difficult to implement correctly, and you need new tools to help you use it. Many of the JVM-based languages are developing tools of this type, and Scala has been particularly active in this area. This series gives you a look at some of the newer approaches to concurrent programming for the Java and Scala languages.

Blocking is easy to do and relatively foolproof, as long as you have a single main thread that waits for events to complete. When you work with multiple threads that are blocking to wait for one another, you can encounter issues such as:

- Deadlocks: Two or more threads each control resources that the other thread(s) need to progress.
- Starvation: Some threads might be unable to progress because other threads are hogging shared resources.
- Livelocks: Threads are trying to adjust to one another but end up making no progress.

Nonblocking approaches leave much more room for creativity. Callbacks are one common technique for nonblocking event handling. Callbacks are the epitome of flexibility, because you can execute whatever arbitrary code you want when the event occurs. The downside is that when you handle many events with callbacks, your code can get messy. And callbacks can be especially confusing to debug because the flow of control doesn't match the order of the code in your application.

Java 8 `CompletableFuture`s support both blocking and nonblocking approaches, including regular callbacks, to handling events. `CompletableFuture` also provides ways to compose and combine events to get the flexibility of callbacks with clean, simple, readable code. In this section, you'll

see examples of both blocking and nonblocking approaches to working with events that are represented by `CompletableFuture`s.

## Tasks and sequencing

Applications often must perform multiple processing steps in the course of a particular operation. For example, before returning a response to the user, a web application might need to:

1. Look up information for the user in a database
2. Use the looked-up information for a web service call and perhaps another database query
3. Perform a database update based on the results from the preceding step.

Figure 1 illustrates this type of structure.

## Figure 1. Application task flow

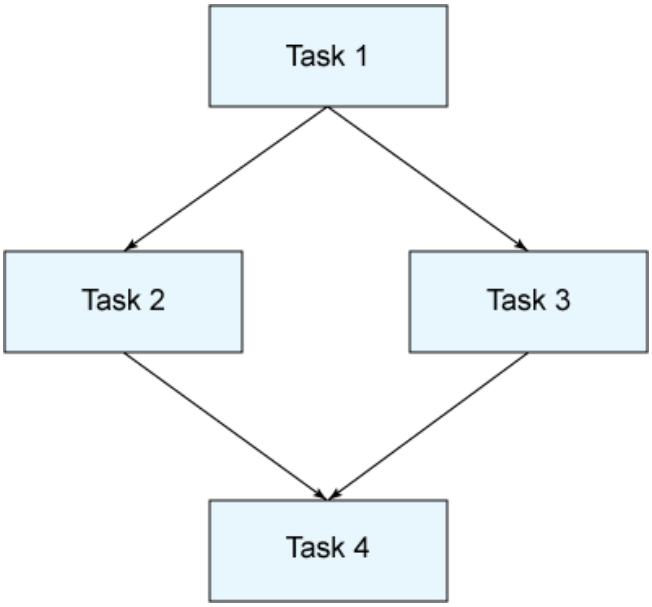



Figure 1 breaks the processing down into four separate tasks, connected by arrows representing order dependencies. Task 1 can execute directly, Task 2 and Task 3 both execute after Task 1 completes, and Task 4 executes after both Task 2 and Task 3 complete. This is the task structure I use in this article to illustrate asynchronous event processing. Real applications — especially server applications with many moving parts — are likely to be much more complex, but this simple example serves to illustrate the principles involved.

## Modeling asynchronous events

In a real system, the sources of asynchronous events are generally either parallel computations or some form of I/O operation. It's easier to model a system of this type using simple time delays, though, and that's the approach I take in this article. Listing 1 shows the basic timed-event code that I use to generate events, in the form of `CompletableFuture`s.

## Listing 1. Timed-event code

```
import java.util.Timer;
import java.util.TimerTask;
```

```
import java.util.concurrent.CompletableFuture;

public class TimedEventSupport {
    private static final Timer timer = new Timer();

    /**
     * Build a future to return the value after a delay.
     *
     * @param delay
     * @param value
     * @return future
     */
    public static <T> CompletableFuture<T> delayedSuccess(int delay, T value) {
        CompletableFuture<T> future = new CompletableFuture<T>();
        TimerTask task = new TimerTask() {
            public void run() {
                future.complete(value);
            }
        };
        timer.schedule(task, delay * 1000);
        return future;
    }

    /**
     * Build a future to return a throwable after a delay.
     *
     * @param delay
     * @param t
     * @return future
     */
    public static <T> CompletableFuture<T> delayedFailure(int delay, Throwable t) {
        CompletableFuture<T> future = new CompletableFuture<T>();
        TimerTask task = new TimerTask() {
            public void run() {
                future.completeExceptionally(t);
            }
        };
        timer.schedule(task, delay * 1000);
        return future;
    }
}
```

## Why not a lambda?

The Listing 1`TimerTask` is implemented as an anonymous inner class with a single `run()` method. You might think this is a great place to use a lambda in place of an inner class. However, you can only use lambdas as instances of interfaces, and `TimerTask` is defined as an abstract class. Unless a future extension of the lambda feature adds support for abstract classes (possible, but unlikely because of design issues) — or parallel interfaces are defined for cases such as `TimerTask`— you must continue using Java inner classes for single method implementations.

The Listing 1 code uses a `java.util.Timer` to schedule `java.util.TimerTask`s for execution after a delay. Each `TimerTask` completes an associated future when it runs. `delayedSuccess()` schedules a task to complete a `CompletableFuture<T>` successfully when it runs and returns the future to the caller. `delayedFailure()` schedules a task to complete a `CompletableFuture<T>` with an exception when it runs and returns the future to the caller.

Listing 2 shows how to use the Listing 1 code to create events, in the form of `CompletableFuture<Integer>`s, matching the four tasks in Figure 1. (This code is from the `EventComposition` class in the sample code.)

## Listing 2. Events for the sample tasks

```
// task definitions
private static CompletableFuture<Integer> task1(int input) {
    return TimedEventSupport.delayedSuccess(1, input + 1);
}
private static CompletableFuture<Integer> task2(int input) {
    return TimedEventSupport.delayedSuccess(2, input + 2);
}
private static CompletableFuture<Integer> task3(int input) {
    return TimedEventSupport.delayedSuccess(3, input + 3);
}
private static CompletableFuture<Integer> task4(int input) {
    return TimedEventSupport.delayedSuccess(1, input + 4);
}
```

Each of the four task methods in Listing 2 uses particular delay values for when the task will complete: 1 second for `task1`, 2 seconds for `task2`, 3 seconds for `task3`, and back down to 1 second for `task4`. Each also takes an input value and uses that input plus the task number as the (eventual) result value for the future. These methods all use the success form of the future; you'll see examples using the failure form later.

The intention with these tasks is that you run them in the order shown in Figure 1 and pass each task the result value returned by the preceding task (or the sum of the two preceding task results, in the case of `task4`). The total execution time should be approximately 5 seconds (1 second + max(2 seconds, 3 seconds) + 1 second) if the two middle tasks execute concurrently. If 1 is the input to `task1`, the result is 2. If that result is passed to `task2` and `task3`, the results are 4 and 5. And if the sum of these two results (9) is passed as the input to `task4`, the final result is 13.

## Blocking waits

Now that the stage is set, it's time for some action. The simplest way to coordinate the execution of the four tasks is to use blocking waits: The main thread waits for each task to complete. Listing 3 (again, from the `EventComposition` class in the sample code) shows this approach.

## Listing 3. Blocking waits for tasks

```
private static CompletableFuture<Integer> runBlocking() {
    Integer i1 = task1(1).join();
    CompletableFuture<Integer> future2 = task2(i1);
    CompletableFuture<Integer> future3 = task3(i1);
    Integer result = task4(future2.join() + future3.join()).join();
    return CompletableFuture.completedFuture(result);
}
```

Listing 3 uses `CompletableFuture`'s `join()` method to do the blocking waits. `join()` waits for the completion and then returns the result value if the completion was successful or throws an unchecked exception if the completion failed or was canceled. The code first waits for the `task1` result, then starts both `task2` and `task3` before waiting for both the returned futures in turn, and finally waits for the `task4` result. `runBlocking()` returns a `CompletableFuture` to be consistent with the nonblocking form that I show next, but in this case, the future will actually be complete before the method returns.

## Composing and combining futures

Listing 4 (again from the `EventComposition` class in the sample code) shows how you can link futures together to execute the tasks in the correct order and with the correct dependencies, all without blocking.

## Listing 4. Nonblocking composition and combination

```
private static CompletableFuture<Integer> runNonblocking() {
    return task1(1).thenCompose(i1 -> ((CompletableFuture<Integer>)task2(i1)
        .thenCombine(task3(i1), (i2,i3) -> i2+i3)))
        .thenCompose(i4 -> task4(i4));
}
```

The Listing 4 code essentially constructs an execution plan that specifies how the different tasks are to be performed and how they relate to one another. This code is elegant and concise but perhaps difficult to understand if you're not familiar with the `CompletableFuture` methods. Listing 5 shows the same code refactored into a more easily understandable form by separating out the `task2` and `task3` parts into a new method, `runTask2and3`.

## Listing 5. Nonblocking composition and combination refactored

```
private static CompletableFuture<Integer> runTask2and3(Integer i1) {
    CompletableFuture<Integer> task2 = task2(i1);
    CompletableFuture<Integer> task3 = task3(i1);
    BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;
    return task2.thenCombine(task3, sum);
}

private static CompletableFuture<Integer> runNonblockingAlt() {
    CompletableFuture<Integer> task1 = task1(1);
    CompletableFuture<Integer> comp123 = task1.thenCompose(EventComposition::runTask2and3);
    return comp123.thenCompose(EventComposition::task4);    }
```

In Listing 5, the `runTask2and3()` method represents the middle portion of the task flow, in which `task2` and `task3` execute concurrently and then their result values are combined. This sequence is coded by using the `thenCombine()` method on a future, which takes another future as its first parameter and a binary function instance (with input types matching the future result types) as its second parameter. `thenCombine()` returns a third future representing the value of the function applied to the results of the original two futures. In this case, the two futures are `task2` and `task3`, and the function is to sum the result values.

The `runNonblockingAlt()` method uses a pair of calls to the `thenCompose()` method on a future. The parameter to `thenCompose()` is a function instance taking the value type of the original future as input and returning another future as output. The result of `thenCompose()` is a third future, with the same result type as the function. This third future serves as a placeholder for the future that will eventually be returned by the function, after the original future has completed.

The call to `task1.thenCompose()` returns a future for the result of applying the `runTask2and3()` function to the result of `task1`, which is saved as `comp123`. The call to `comp123.thenCompose()` returns a future for the result of applying the `task4()` function to the result from the first `thenCompose()`, which is the overall result of executing all the tasks.

### Trying out the example

The sample code includes a `main()` method to run each version of the events code in turn and show that the time to completion (about 5 seconds) and the result (13) are correct. Listing 6 shows the result of running this `main()` method from a console.

### Listing 6. Running the `main()` method

```
dennis@linux-guk3:~/devworks/scala3/code/bin> java com.sosnoski.concur.article3.EventComposition
Starting runBlocking
runBlocking returned 13 in 5008 ms.
Starting runNonblocking
runNonblocking returned 13 in 5002 ms.
Starting runNonblockingAlt
runNonblockingAlt returned 13 in 5001 ms.
```

## The unhappy path

So far, you've seen code to coordinate events in the form of futures that always complete successfully. In real applications, you can't depend on always staying on this happy path. Problems with processing tasks will occur, and in Java terms, these problems are normally represented by `Throwable`s.

It's easy to change the Listing 2 task definitions to use `delayedFailure()` in place of the `delayedSuccess()` method, as shown here for `task4`:

```
private static CompletableFuture<Integer> task4(int input) {
    return TimedEventSupport.delayedFailure(1, new IllegalArgumentException("This won't work!"));
}
```

If you run the Listing 3 code with only `task4` modified to complete with an exception, you get the expected `IllegalArgumentException` thrown by the `join()` call on `task4`. If the problem isn't caught in the `runBlocking()` method, the exception is passed up the call chain, ultimately terminating the executing thread if not caught. Fortunately, it's easy to modify the code so that if any of the tasks completes exceptionally, the exception is passed on to the caller for handling through the returned future. Listing 7 shows this change.

### Listing 7. Blocking waits with exceptions

```
private static CompletableFuture<Integer> runBlocking() {
    try {
        Integer i1 = task1(1).join();
        CompletableFuture<Integer> future2 = task2(i1);
        CompletableFuture<Integer> future3 = task3(i1);
        Integer result = task4(future2.join() + future3.join()).join();
        return CompletableFuture.completedFuture(result);
    } catch (CompletionException e) {
        CompletableFuture<Integer> result = new CompletableFuture<Integer>();
        result.completeExceptionally(e.getCause());
        return result;
    }}
```

Listing 7 is mostly self-explanatory. The original code is wrapped in a `try`/`catch`, and the `catch` passes the exception back as the completion of the returned future. This approach adds a little complexity but should still be easy for any Java developer to understand.

The nonblocking code in Listing 4 doesn't even require adding a `try`/`catch`. The `CompletableFuture` compose and combine operations take care of passing exceptions on for you automatically, so that dependent futures also complete with the exception.

## To block or not to block

You've seen examples of both blocking and nonblocking approaches to handling events that are represented by `CompletableFuture`s. At least for the basic task flow modeled in this article, both approaches are pretty simple. For more-complex task flows, the code also gets more complex.

In the blocking case, the added complexity is not much of an issue, provided you only need to wait for the completion of events. If you start doing other types of synchronization among threads, you get into the issues of thread starvation and even deadlock.

In the nonblocking case, code execution triggered by the completion of an event is difficult to debug. When you have many types of events going on and many interactions among events, it becomes hard to follow which event is triggering which execution. This situation is basically a case of callback hell, whether you're using conventional callbacks or the `CompletableFuture` combine and compose operations.

On balance, the simplicity advantage is usually with blocking code. So why would anyone want to use a nonblocking approach instead? This section gives you a couple of important reasons.

### The cost of switching

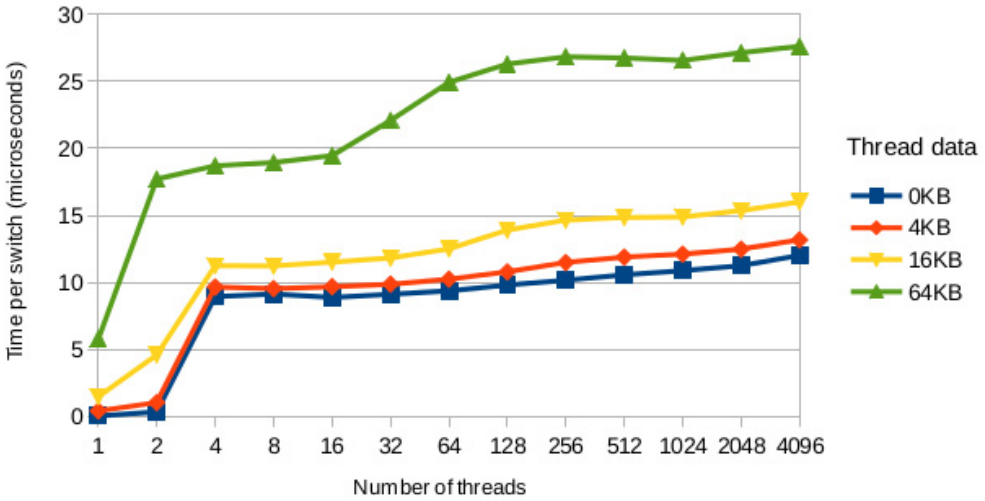
When a thread blocks, the processor core previously executing that thread moves on to another thread. The execution state of the previously executing thread must be saved to memory and the state of the new thread loaded. This operation of changing a core from running one thread to running another thread is called a *context switch*.

Beyond the direct context-switch performance cost, the new thread generally uses different data from the prior thread. Memory access is much slower than the processor clock, so modern systems use multiple layers of caches between the processor cores and main memory. Although much faster than main memory, the caches are also much smaller in capacity — in general, the faster the cache, the smaller the capacity — so only a small portion of the total memory can be present in caches at any one time. When a thread switch occurs and a core starts executing a new thread, it's likely that the memory data needed by the new thread will not be present in the caches, so the core must wait while the data is loaded from main memory.

The combined context-switch and memory-access delays incur a direct and significant performance cost. Figure 2 shows the overhead of thread switching on my four-core AMD system using Oracle's Java 8 for 64-bit Linux. This test uses a variable number of threads, from 1 up to 4,096 by powers of 2, and a variable-size block of memory per thread, from 0 to 64KB. Threads are executed in turn, using `CompletableFuture` to trigger the execution. Each time a thread executes, it first runs a simple calculation using the per-thread data, to show the overhead of loading this data into the cache, then increments a shared static variable. It finishes by creating a new `CompletableFuture` instance to trigger its next execution, then starts the next thread in

sequence by completing the `CompletableFuture` that thread is waiting on. Finally, if it needs to execute again, the thread waits for the newly created `CompletableFuture` to complete.

## Figure 2. Thread-switching cost




You can see the impact of both numbers of threads and memory per thread in the Figure 2 chart. The number of threads has the most impact up to four threads, with two threads working almost as fast as a single thread as long as the per-thread data is fairly small. There's relatively little impact on performance as the number of threads increases beyond four. The larger the memory amount per thread, the sooner the two layers of cache are filled to overflowing, resulting in bumps in the switching cost.

The timing values shown in Figure 2 are for my somewhat dated main system. The corresponding times on your system will differ and quite possibly be much smaller. However, the shape of the curves should be roughly the same.

Figure 2 shows the overhead in microseconds of a thread switch, so even though the cost of a thread switch is in the tens of thousands of processor clocks, the absolute numbers aren't huge. At the 12.5-microsecond switching time corresponding to 16KB of data (the yellow line in the graph) with a moderate number of threads, the system can execute 80,000 thread switches per second. That's far more thread switching than you're likely to see in any reasonably written single-user application and even in many server applications. But for high-performance server applications that deal with many thousands of events per second, the overhead of blocking can become a major factor in performance. For this type of application, it's important to minimize thread switches by using nonblocking code wherever possible.

It's also important to realize that these timing figures are for a best-case scenario. When running the thread-switching program, enough CPU activity is going on to keep all the cores operating at full clock speed (at least on my system). In a real application, the processing load is likely to be more bursty. At times of low activity, modern processors will transition some of the cores to sleep states to reduce overall power consumption and heat generation. The only problem with this powering-down process is that it requires time to wake the core from the sleep state when demand

picks up. The time required to go from a deep sleep state to full operation can be on the order of milliseconds rather than the microseconds seen in this thread-switch timing example.

### Reactive applications

For many applications, another reason not to block on particular threads is that those threads are used for processing events that require a timely response. The classic example is a UI thread. If you execute code in the UI thread that blocks to wait for the completion of asynchronous events, you delay processing of user-input events. Nobody likes to wait for an application to respond to what they type, click, or touch, so blocking in the UI thread tends to be reflected quickly in bug reports from users.

A more-general principle underlies the UI thread concept. Most types of applications, even non-GUI ones, also must respond to events, and in many cases keeping response times low is a critical concern. For these types of applications, blocking waits are not an acceptable option.

*Reactive programming* is the name that has come to represent the style of programming that makes for highly responsive and scalable applications. The core principles of reactive programming are that applications should be able to:

- React to events: The application should be event-driven, with loosely coupled components at every level linked by asynchronous communications.
- React to load: The application should be scalable so that it can easily be upgraded to handle increased demand.
- React to failure: The application should be resilient, with failures localized in impact and quickly correctable.
- React to users: The application should be responsive to users, even under load and in the presence of failures.

Applications that use blocking approaches to event handling can't match up to these principles. Threads are a limited resource, so tying them up in blocking waits limits scalability while also adding to latency (application response time), because blocked threads are unavailable to respond immediately to an event. Nonblocking applications can respond more rapidly to events, lowering latency while also reducing thread-switching overhead and improving throughput.

There's a lot more to reactive programming than nonblocking code. Reactive programming involves focusing on the data flows in your application and implementing these data flows as asynchronous interactions without overloading the receivers or backing up the senders. This focus on data flows helps avoid much of the complexity of traditional concurrent programming.

## Conclusion

In this article, you've seen how to use the Java 8 `CompletableFuture` class to compose and combine events into a kind of execution plan that can be expressed in code easily and concisely. This type of nonblocking composability is essential to writing reactive applications that respond well to workload and gracefully handle failures.

The next article will swing over to the Scala side and look into a very different but also interesting way of handling asynchronous computations. The `async` macro enables you to write code that looks like it does sequential blocking operations, but under the covers translates the code into a completely nonblocking structure. I'll give some examples of how this approach is useful and also take a look at how `async` is implemented.

## Resources

- Sample code for this article: Get this article's full sample code from the author's repository on GitHub.
- Scalable Scala: Series author Dennis Sosnoski shares insights and behind-the-scenes information on the content in this series and Scala development in general.
- "Java 8: Definitive guide to CompletableFuture" (Tomasz Nurkiewicz, May 2013): In this blog post, read an excellent run-through of the different ways of working with `CompletableFuture`.
- The Reactive Manifesto: This document sets out the advantages of the reactive programming style. You can register your support by signing the manifesto if you are so inclined.
- "*JVM concurrency*: Java and Scala concurrency basics" (Dennis Sosnoski, developerWorks, March 2014): Learn about the basics of handling concurrency in both Java and Scala in this first article of the series.
- "*Java theory and practice*: Introduction to nonblocking algorithms" (Brian Goetz, developerWorks, April 2006): See how Java `java.util.concurrent` classes support nonblocking interactions between threads (missing details on the latest additions, but good background).
- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

## About the author

### Dennis Sosnoski

Dennis Sosnoski is a Java and Scala developer with extensive experience developing scalable systems. He's well-known in the XML and web services areas, where his background includes the development of JiBX XML data binding and work on several open source web services frameworks (most recently, Apache CXF). Dennis is a frequent presenter at Java user groups and conferences and has written many articles for developerWorks, including the popular *Java web services* series. Learn more about his web services training and consulting work at his Sosnoski Software Associates Ltd site, and follow his ongoing explorations of concurrent programming on the JVM at his Scalable Scala site.