5 things you didn't know about ... `java.util.concurrent`, Part 1

Multithreaded programming with concurrent Collections

Ted Neward
Principal
Neward & Associates

18 May 2010

Writing multithreaded code that both performs well and protects applications against corruption is just plain hard — which is why we have `java.util.concurrent`. Ted Neward shows you how concurrent Collections classes like `CopyOnWriteArrayList`, `BlockingQueue`, and `ConcurrentMap` retrofit standard Collections classes for your concurrency programming needs.

About this series

So you think you know about Java programming? The fact is, most developers scratch the surface of the Java platform, learning just enough to get the job done. In this series, Ted Neward digs beneath the core functionality of the Java platform to uncover little-known facts that could help you solve even the stickiest programming challenges.

Concurrent Collections were a huge addition to Java™ 5, but many Java developers lost sight of them in all the hoopla about annotations and generics. Additionally (and perhaps more truthfully), many developers avoid this package because they assume that it, like the problems it seeks to solve, must be complicated.

In fact, `java.util.concurrent` contains many classes that effectively resolve common concurrency problems, without requiring you to break a sweat. Read on to learn how `java.util.concurrent` classes like `CopyOnWriteArrayList` and `BlockingQueue` help you solve the pernicious challenges of multithreaded programming.

1. TimeUnit

Develop skills on this topic

This content is part of progressive knowledge paths for advancing your skills. See:

- Become a Java developer

- Java concurrency

While it's not a Collections class, *per se*, the `java.util.concurrent.TimeUnit` enumeration makes code vastly easier to read. Using `TimeUnit` frees developers using your method or API from the tyranny of the millisecond.

`TimeUnit` incorporates all units of time, ranging from `MILLISECONDS` and `MICROSECONDS` up through `DAYS` and `HOURS`, which means it handles almost all time-span types that a developer might need. And, thanks to conversion methods declared on the enum, it's even trivial to convert `HOURS` back to `MILLISECONDS` when time speeds up.

2. CopyOnWriteArrayList

Making a fresh copy of an array is too expensive an operation, in terms of both time and memory overhead, to consider for ordinary use; developers often resort to using a synchronized `ArrayList` instead. That's also a costly option, however, because every time you iterate across the contents of the collection, you have to synchronize all operations, including read and write, to ensure consistency.

This puts the cost structure backward for scenarios where numerous readers are reading the `ArrayList` but few are modifying it.

`CopyOnWriteArrayList` is the amazing little jewel that solves this problem. Its Javadoc defines `CopyOnWriteArrayList` as a "thread-safe variant of `ArrayList` in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the array."

The collection internally copies its contents over to a new array upon any modification, so readers accessing the contents of the array incur no synchronization costs (because they're never operating on mutable data).

Essentially, `CopyOnWriteArrayList` is ideal for the exact scenario where `ArrayList` fails us: read-often, write-rarely collections such as the `Listener`s for a JavaBean event.

3. BlockingQueue

The `BlockingQueue` interface states that it is a `Queue`, meaning that its items are stored in first in, first out (FIFO) order. Items inserted in a particular order are retrieved in that same order — but with the added guarantee that any attempt to retrieve an item from an empty queue will block the calling thread until the item is ready to be retrieved. Likewise, any attempt to insert an item into a queue that is full will block the calling thread until space becomes available in the queue's storage.

`BlockingQueue` neatly solves the problem of how to "hand off" items gathered by one thread to another thread for processing, without explicit concern for synchronization issues. The Guarded Blocks trail in the Java Tutorial is a good example. It builds a single-slot bounded buffer using manual synchronization and `wait()`/`notifyAll()` to signal between threads when a new item is available for consumption, and when the slot is ready to be filled with a new item. (See the Guarded Blocks implementation for details.)

Despite the fact that the code in the Guarded Blocks tutorial works, it's long, messy, and not entirely intuitive. Back in the early days of the Java platform, yes, Java developers had to tangle with such code; but this is 2010 — surely things have improved?

Listing 1 shows a rewritten version of the Guarded Blocks code where I've employed an `ArrayBlockingQueue` instead of the hand-written `Drop`.

Listing 1. BlockingQueue

```
import java.util.*;
import java.util.concurrent.*;

class Producer
    implements Runnable
{
    private BlockingQueue<String> drop;
    List<String> messages = Arrays.asList(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "Wouldn't you eat ivy too?");

    public Producer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            for (String s : messages)
                drop.put(s);
            drop.put("DONE");
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

class Consumer
    implements Runnable
{
    private BlockingQueue<String> drop;
    public Consumer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            String msg = null;
            while (!((msg = drop.take()).equals("DONE")))
                System.out.println(msg);
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

public class ABQApp
{
```

```
    public static void main(String[] args)
    {
        BlockingQueue<String> drop = new ArrayBlockingQueue(1, true);
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

The `ArrayBlockingQueue` also honors "fairness" — meaning that it can give reader and writer threads first in, first out access. The alternative would be a more efficient policy that risks starving out some threads. (That is, it would be more efficient to allow readers to run while other readers held the lock, but you'd risk a constant stream of reader threads keeping the writer from ever doing its job.)

Bug watch!

By the way, you're right if you noticed that Guarded Blocks contains a huge bug — what would happen if a developer synchronized on the `Drop` instance inside of `main()`?

`BlockingQueue` also supports methods that take a time parameter, indicating how long the thread should block before returning to signal failure to insert or retrieve the item in question. Doing this avoids an unbounded wait, which can be death to a production system, given an unbounded wait can all too easily turn into a system hang requiring a reboot.

4. ConcurrentMap

`Map` hosts a subtle concurrency bug that has led many an unwary Java developer astray. `ConcurrentMap` is the easy solution.

When a `Map` is accessed from multiple threads, it's common to use either `containsKey()` or `get()` to find out whether a given key is present before storing the key/value pair. But even with a synchronized `Map`, a thread could sneak in during this process and seize control of the `Map`. The problem is that the lock is acquired at the start of `get()`, then released before the lock can be acquired again, in the call to `put()`. The result is a race condition: it's a race between the two threads, and the outcome will be different based on who runs first.

If two threads call a method at exactly the same moment, each will test and each will put, losing the first thread's value in the process. Fortunately, the `ConcurrentMap` interface supports a number of additional methods that are designed to do two things under a single lock: `putIfAbsent()`, for instance, does the test first, then does a put only if the key isn't stored in the `Map`.

5. SynchronousQueues

`SynchronousQueue` is an interesting creature, according to the Javadoc:

> A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one.

Essentially, `SynchronousQueue` is another implementation of the aforementioned `BlockingQueue`. It gives us an extremely lightweight way to exchange single elements from one thread to another,

using the blocking semantics used by `ArrayBlockingQueue`. In Listing 2, I've rewritten the code from Listing 1 using `SynchronousQueue` instead of `ArrayBlockingQueue`:

Listing 2. SynchronousQueue

```
import java.util.*;
import java.util.concurrent.*;

class Producer
    implements Runnable
{
    private BlockingQueue<String> drop;
    List<String> messages = Arrays.asList(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "Wouldn't you eat ivy too?");

    public Producer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            for (String s : messages)
                drop.put(s);
            drop.put("DONE");
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

class Consumer
    implements Runnable
{
    private BlockingQueue<String> drop;
    public Consumer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            String msg = null;
            while (!((msg = drop.take()).equals("DONE")))
                System.out.println(msg);
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

public class SynQApp
{
    public static void main(String[] args)
    {
        BlockingQueue<String> drop = new SynchronousQueue<String>();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
```

```
}
```

The implementation code looks almost identical, but the application has an added benefit, in that `SynchronousQueue` will allow an insert into the queue only if there is a thread waiting to consume it.

In practice, `SynchronousQueue` is similar to the "rendezvous channels" available in languages like Ada or CSP. These are also sometimes known as "joins" in other environments, including .NET (see Resources).

In conclusion

Why struggle with introducing concurrency to your Collections classes when the Java runtime library offers handy, prebuilt equivalents? The next article in this series explores even more of the `java.util.concurrent` namespace.

Downloads

Description	Name	Size
Sample code for this article	j-5things4-src.zip	23KB

Resources

- "*Java theory and practice*: Concurrent collections classes" (Brian Goetz, developerWorks, July 2003): Learn how Doug Lea's `util.concurrent` package revitalizes standard collection types `List` and `Map`.
- Java Concurrency in Practice (Brian Goetz, et. al. Addison-Wesley, 2006): Brian's remarkable ability to distill complex concepts for readers makes this book a must on any Java developer's bookshelf.
- "Spice up collections with generics and concurrency" (John Zukowski, developerWorks, April 2008): Introduces changes to the Java Collections Framework in Java 6.
- *Package java.util.concurrent, Java platform SE 6*: Learn more about the utility classes discussed in this article.
- Guarded Blocks: The most common idiom for coordinating threads.
- "Introduction to the Collections Framework" (MageLang Institute, Sun Developer Network, 1999): This old but good tutorial is a complete introduction to the Java Collections Framework prior to concurrent collections.
- "The Collections Framework": Read the Java Collections Framework and API documentation from Sun Microsystems.
- The Joins Concurrency Library: Microsoft® Research put out this library implementing the *joins* concept as a synchronization mechanism; the associated research paper (in PDF) is a good source for learning about the theory behind joins.
- The developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.
- Get involved in the My developerWorks community.

About the author

Ted Neward

Ted Neward is the principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.