CSP for Java programmers, Part 3

Advanced topics in JCSP

Abhijit Belapurkar 21 June 2005

Abhijit Belapurkar concludes his three-part introduction to CSP for Java™ developers with an inquiry into advanced topics in JCSP development, including parallels to AOP, JCSP compared to `java.util.concurrent`, and advanced synchronization with JCSP.

If you've been reading along with this series from the beginning, you should agree by now that CSP is a programming paradigm worth investigating and the JCSP library a worthy addition to your toolkit. In this third part I'll simply give you a few more reasons to investigate JCSP. I'll start by discussing some of the parallels between the compositional approach of CSP and the modular approach of aspect-oriented programming, or AOP. I'll then introduce you to some of JCSP's more advanced synchronization constructs and discuss the scenarios where you might apply them, as well as discussing more broadly the benefits of multithreaded programming with JCSP -- namely verification, maintenance, and an approach inherently applicable to distributed programming.

I'll also answer the question of whether there is a point to learning JCSP if you have already adopted the concurrency utilities in `java.util.concurrent`. (Obviously, I think there is!)

If you haven't read Part 1 and Part 2 of this article you may want to do so before continuing.

AOP and CSP

Aspect-oriented programming, or AOP, is a term that is surely known to most Java developers by now. AOP looks at systems as composed of aspects or concerns rather than purely of objects. It strives to improve the modularity of systems by grouping crosscutting concerns into individual modules whose code addresses one specific concern. These modules can then be weaved together using techniques such as static or dynamic weaving.

Having come this far in the article series, it should be easy for you to see the parallel between the AOP concept of *weaving together modules* and the CSP concept of *composing process networks*. The parallel (pun intended!) between the two technologies is largely due to the compositional nature of the latter.

An example based on custom security checks is frequently used to make the case for AOP. In this example, a module containing code addressing an application's security concern is written

© Copyright IBM Corporation 2005

and packaged as an aspect; this aspect is then applied to all the appropriate business object(s) throughout the application. To resolve the same problem using CSP you would likely start by writing one process containing the functionality of the business object and another containing the application's security checking functionality. You would then connect the two processes using a `One2One` channel. The advantage of the CSP approach is similar to that of the AOP approach: namely separation of concerns.

Don't miss the rest of the series!

"CSP for Java programmers" is a three-part introduction to Communicating Sequential Processes (CSP), a paradigm for concurrent programming that honors its complexity without abandoning you to it. Read the other parts of the series:
Part 1: Pitfalls of mutlithreaded programming on the Java platform
Part 2: Concurrent programming with JCSP

Advanced JCSP synchronization

Channels aren't the only available option for synchronization between processes, nor are they always the most appropriate. A channel always synchronizes between just two processes: the reader and the writer. It's immaterial whether there is more than one reader/writer process at work -- as is the case in the `Any2One`, `One2Any`, and `Any2Any` modes -- as only one process of each type can be actively engaged in an ongoing conversation on either side of the channel.

For more advanced synchronization, that is, synchronization between multiple processes, JCSP provides the `Barrier`, `Bucket`, and `CREW` constructs, as well as a class called `ProcessManager` for managing `CSProcess` instances.

The Barrier construct

A *barrier* is a JCSP event that acts as a synchronization mechanism between multiple processes. Each barrier can be associated with multiple processes and any one process that synchronizes on that barrier will be blocked until all other processes have also synchronized.

The `Barrier` class must be created with the number of processes that will be synchronizing on it, with each process running on a separate thread. After completing its work for the current round, each process invokes the `sync()` method on the `Barrier` instance, and so waits for the other processes to finish.

Barrier internals

Internally, `Barrier` contains a member variable that is initialized to the total number of processes specified when the `Barrier` object is created. Every time one of the processes invokes the `sync()` method, this variable is decremented by one and a `wait()` call is issued. (You will remember that a Java thread must be holding the appropriate lock before it can call the `wait()` method -- in this case, the `Barrier` instance itself serves as the lock by virtue of the fact that its `sync()` method is marked synchronized.) The check happens on the thread of the last process to invoke `sync()`). As soon as the member variable becomes `0` it is clear that all processes are done invoking `sync()` and it is safe to issue a `notifyAll()`. At this time the internal counter is reset to the initially provided total number of processes, making the `Barrier` instance available for reuse in the next round of synchronization.

Note that the `Barrier` doesn't keep the identities of the processes associated with it. As a result, any arbitrary process (that is, besides the ones supposed to be synchronizing on it), provided it can get hold of the `Barrier` object, can call `sync` on it, thereby shutting out one or more of the rightful processes. This can be prevented by imposing a strict control on the visibility of the `Barrier` object to only the set of processes that have synchronized on it.

Hey, that sounds familiar!

If you're thinking that the description of how a `Barrier` works sounds familiar then you're bang on target! The underlying mechanism of the `Barrier` construct is the basis of how the `Parallel` construct runs the multiple `CSProcess` instances in its control.

As you'll recall, when you invoke the `run` method on an instance of `Parallel`, it creates *(n-1)* threads to run the first *(n-1)*`CSProcess` instances and runs the last `CSProcess` instance on its own thread. All these processes `sync` on a common `Barrier` instance created in the constructor of the `Parallel` class (and passed to each of the *n-1* threads responsible for running `CSProcess`es via their constructors).

`Barrier` is a lower level construct than `Parallel` and correspondingly provides more flexibility, in that a process can `enroll` or `resign` from a `Barrier` object at any time. For example, a worker process might want to sync with a clock (essentially simulating clock ticks) while working, then stop and rest for a while (during which time it would resign from the barrier) and then commence working again by (re)enrolling with the barrier.

The Bucket construct

A *bucket* is also a synchronization barrier for multiple processes. The difference between the bucket and barrier constructs is that in the former all processes are blocked until another (external) process decides to release them, by "flushing the bucket."

The way to use a `Bucket` is to simply create one; every process that needs to synchronize on it can then call its `fallInto()` method. `fallInto()` is a synchronized method that maintains an internal counter variable for keeping track of the number of (waiting) processes at that instant (that is, the counter variable is incremented by one for every process that invokes the `fallInto()` method). After the counter has been incremented, the invoking process does a `wait()`, which leads to the calling thread getting blocked. When an external process calls the `Bucket`'s (also synchronized) `flush()` method, a `notifyAll()` is sent, thereby waking up all blocking processes. The final count of blocking processes is output from this call.

While a `Barrier` is deterministic, a `Bucket` is non-deterministic. Whereas processes synched to a `Barrier` are guaranteed to be rescheduled for execution (after a time period equal to the execution time of the slowest one in the group), processes synched to a `Bucket` can block for arbitrary periods of time depending on when the flusher process decides to issue the call to release them.

A `Barrier` is useful when a set of processes must proceed in lockstep. That is, each process takes step number *n* and doesn't go on to step *n+1* until all other processes in that set have completed their step *n*. A `Barrier` is useful for situations where each process completes its assigned work

and parks itself in a bucket, indicating its availability for the next step. The flusher process can periodically flush this bucket (perhaps using some job-scheduling algorithm), thereby releasing all processes that were present in the bucket at that instant to begin the next set of tasks.

The CREW construct

The Concurrent-Read-Exclusive-Write construct, or CREW lock, allows multiple, parallel readers to access a shared resource provided that (1) the "read" access will not change the state of the resource and (2) no writer is already accessing the resource. When a writer is accessing a resource it has exclusive rights over it: Neither readers nor writers are allowed access to the resource.

The use of a CREW lock is intended to both eliminate race hazards (due to arbitrary interleaving of competing reader/writer processes) and yield performance improvements by letting multiple reader processes run in parallel. Whether there is actual performance improvement will of course depend on the frequency with which the resource is accessed in read mode as compared to write mode; the duration of the read and write operations; and the number of processes trying to access the shared resource in conflicting modes at the same time.

See the Javadocs accompanying the JCSP software release (in Resources) for an excellent description of the internal implementation of the CREW lock.

Managing CSProcess instances

In Part 2 I showed you how to use the `Parallel` construct to compose a network of higher-level processes from lower-level "building block" processes. While good for many programming scenarios, my examples in that article assumed that I already knew all the details about all the processes I would ever need to create. For some applications, however, you will find it necessary to create one or more processes on the fly, based on some condition that can be evaluated only at runtime, and to also be able to manage them.

JCSP.net

The JCSP library I've used for the examples in this article essentially works on threads running within the same JVM, which of course may be running on a single- or multiprocessor machine. In case you have been wondering if it is possible to compose layered networks of communicating and synchronizing processes that are spread across many JVMs, the answer is yes! A variation of the JCSP library, called *JCSP.net*, enables you to do precisely that. The best part of programming with JCSP.net is that virtual channels running over physical networks will appear more or less the same as the local, single-JVM channels you've encountered in this article, although obviously some complexities will enter in. See Resources to learn more about the commercially available version of JCSP.net.

JCSP provides a class called `ProcessManager` for this type of scenario. This class takes a `CSProcess` instance as input argument and allows for the process to be started in one of two different ways. If the `run()` method is called on the `ProcessManager`, the `ProcessManager` starts the managed process and waits for it to terminate (the managed process runs on the thread of the `ProcessManager`, so the latter is prevented from running until the managed process completes). The other option is to call the `start()` method on the `ProcessManager`, causing the managed

process to be started in a separate thread, in which case the `ProcessManager` itself keeps running as well.

Managing parallel priority

I mentioned in Part 2 that the `Parallel` construct runs in parallel all the `CSProcess` instances contained in the array passed to its constructor. In some circumstances, however, these processes may have to be given different priority. JCSP provides a class called `PriParallel` that prioritizes the processes passed to it.

The priority of a controlled process is determined by its position index in the array. A process that is placed earlier in the array has a higher priority. Note that JCSP implements process priorities in terms of the underlying thread's priority mechanism; so how the priorities actually work out depends on the underlying JVM implementation.

More (and more!) constructs

The constructs discussed thus far (first in Part 2 and now here) are the core constructs of the JCSP library. They can be used to resolve both normal and advanced concurrency issues and should get you started writing pain-free (as well as error-free!) multithreaded programs. Of course, they're just a tip of the iceberg when it comes to the JCSP library. Consider the application of the following interesting constructs and you may want to investigate them on your own:

- **BlackHoleChannel** implements the `ChannelOutput` interface and contains an empty `write()` method. Any amount of data can be safely written to this channel. The `BlackHoleChannel` class is most useful if you have an existing process that you want to use in a different context, wherein some of the output is *not* going to be used. Since you cannot ever leave an output channel unhandled (for fear of causing deadlocks) your best option is to get the process to produce its output on an instance of `BlackHoleChannel`, which effectively acts as a transparent, bottomless sink for the produced data.
- **Paraplex** is a process that converts multiple streams of objects on its set of input channels onto a single output channel. It waits until some input is available on each of the input channels (accepting them as they come, no specific order required or enforced), then packs them into an array (whose size is the same as the number of input channels) and sends the array out as a single output on the output channel. `Paraplex` could come in handy if you needed to present some data in tabular form. As an example, say you have a table with three columns, each of which must be filled in with numbers. The first two columns will be filled in with numbers generated from JCSP processes you've already encountered previously: `NumbersInt` to generate a sequence of natural numbers starting from 0 for the first column and `SquaresInt` to generate the sequence of corresponding squares for the second column. A plug-and-play JCSP process called `FibonacciInt` that is available out-of-the-box with the JCSP library will be used to generate the sequence of Fibonacci numbers for the third column.
  You could easily handle all three columns with a single `ParaplexInt` instance comprised of three input channels. Each input channel would have attached to it an instance of one of the three processes mentioned. The single output channel of the `ParaplexInt` would then feed

into a `CSProcess`, which in turn would read an array with three elements and print them out in the appropriately formatted table.
- **Deparaplex** is a class that, converse to the `Paraplex` class, *de-multiplexes* data from its single-input channel to an array of output channels. So, for example, the `Deparaplex` class might read a single array object (of size *n*) and output one element each from the array onto one of its *n* output channels. The `Deparaplex` process would then wait until each element it produced had been accepted by the channel on which it was written out.

See the JCSP library download (in Resources) for further documentation of these constructs, including usage examples.

Benefits of CSP

CSP is a paradigm for concurrent programming based on a foundation of mature mathematical theory. As a result, CSP provides a rich set of design patterns and tools to guard against common multithreading pitfalls such as race hazards, deadlocks, livelocks, and resource starvation. Because all this mathematical sophistication is built into the JCSP library, you can directly use it to write application programs based on the guidelines laid out (that is, you don't necessarily have to understand the theory in order to take advantage of it; though understanding clearly gives you an edge!). Because a formal CSP model for Java exists, it is also possible to analyze and formally verify any multithreaded Java application built using JCSP constructs.

As pointed out earlier, many of the benefits of AOP apply also to CSP-based programs. A primary benefit of CSP-based programs is separation of concerns. Using CSP as the basis for your programs (for example via the JCSP library) ensures a clean decoupling between processes, since they can interact only via channels and not via direct method invocations. It also ensures the perfect encapsulation of data and application logic within processes behind channel interfaces. All side-effects are limited to within process boundaries and interactions between programmatic entities are explicitly exposed. There are no hidden interactions in JCSP-based programs -- at each level of the network all the plumbing is visible.

A second benefit of CSP is its compositional nature. Processes can be composed to form more complex processes (which can themselves be composed to form even more complex processes) and can be easily modified and extended over time. As a result, application designs based on CSP are exceptionally simplified and easy-to-understand, as well as being very robust when it comes to maintenance. CSP's compositional nature also promotes better reuse; as you saw in the programming examples in Part 2, a single JCSP process can be used in a number of different settings, with unwanted outputs being redirected into black holes if required.

The final benefit of concurrent programming with CSP will be particularly evident to developers building distributed systems. In this article I've described the different options for achieving concurrency (namely multiple threads running in a single process, multiple processes running on one processor, and multiple processes running on multiple processors). Traditionally, each of these concurrency mechanisms have required the use of completely different execution models, programming interfaces, and design paradigms. CSP is different in that it provides a unified execution model with a single set of programming interfaces to learn and use and a single design

paradigm. Once you've developed an application using CSP, the decision regarding which platform to run the application on -- a multiple-threads platform, a multiple-processes-on-a-single processor platform, or a multiple-processes-running-on-distributed-processors platform -- can be taken without impacting (at least significantly) the application design or code.

JCSP and java.util.concurrent

Most Java developers are well aware of the `java.util.concurrent` package introduced as a standard part of the J2SE 1.5 class library. Because the JCSP library pre-dates the addition of this package to the Java platform, you may wonder if there is a need for both, or why you should bother learning JCSP if you've already adapted to `java.util.concurrent`.

The `java.util.concurrent` package is essentially a reformulation of Doug Lea's superb and ever-popular `util.concurrent` library. The intent of the package was to provide a robust, high-performing set of standardized utilities for the simplification of concurrent development on the Java platform -- and that intent has been achieved, *unequivocally!* If the utilities in the `java.util.concurrent` package are a good fit for your applications, use them and you will be rewarded.

Communicating Threads in Java

JCSP is not the only available CSP implementation for the Java platform. Communicating Threads in Java (CTJ) is another popular CSP implementation developed by Gerald Hilderink and Andy Bakkers at the University of Twente. See Resources for a detailed comparison between JCSP and CTJ in terms of their philosophies, similarities and differences, performance, and usage scenarios.

As I hope the discussion in this article has shown, however, a CSP-based approach (via the JCSP library) present opportunities well beyond the scope of `java.util.concurrent`. The paradigm of building systems as networks of communicating processes and using them to compose networks within networks is both natural and simple. It will not only improve the experience of writing multithreaded applications, but also the *quality* of your products. Building applications with JCSP results in clean interfaces and lack of hidden side-effects which makes the resulting product resilient to maintenance and technology changes. Formal verification (that is, the ability to reason formally about your application's safety and liveness properties) is also a powerful asset for safety-critical and/or high-value financial applications.

In any case, the two packages are not mutually exclusive: some of the core JCSP primitives are likely to be re-implemented on top of the atomic low-level mechanisms provided by `java.util.concurrent`, which may carry much lower overheads than the standard monitor mechanisms currently used.

Conclusion to Part 3

I've covered a lot of ground in this introduction to CSP for Java programmers. In Part 1 I walked through the constructs for multithreaded programming currently supported by the Java language. I explained the origins of these constructs and discussed their relationship to the four common pitfalls of multithreaded programming on the Java platform, namely race hazards, deadlocks,

livelocks, and resource starvation. I concluded Part 1 by explaining why it is difficult to apply a formal theory to either weed out programming bugs or prove their absence in arbitrary, large, and complex multithreaded applications. I proposed CSP as an alternative to this impasse.

In Part 2 I introduced both the theory of CSP and one of its most popular Java-based implementations, the JCSP library. I used several working examples to demonstrate the use of essential JCSP constructs such as processes, channels, and guards.

With this last article, I offered a look into advanced topics in JCSP, including its parallels to aspect-oriented programming and how it compares to some of the other concurrent programming packages for the Java platform. I also demonstrated some of the JCSP constructs for resolving more complex synchronization issues and briefly discussed the possibility of distributed programming with the JCSP.net package.

As I hope I've shown, there are a multitude of advantages to writing multithreaded Java applications using JCSP. Next time you find yourself considering (and perhaps shying away from) a multithreaded application design, you may want to turn instead to the JCSP library. JCSP offers a greatly simplified approach to concurrent programming and results in programs that can be verified against the most common pitfalls of multithreaded application development, besides being easy to understand, debug and maintain in the long run.

Acknowledgments

I would like to gratefully acknowledge the kind encouragement I received from Professor Peter Welch during the writing of this article series. His busy schedule notwithstanding, he took time to do a very thorough review of a draft version and gave many valuable inputs towards enhancing the quality and accuracy of the series. All remaining errors are mine alone! The examples I have worked with in my articles are based on and/or derived from those documented in the Javadocs for the JCSP library and/or the Powerpoint presentation slides available on the JCSP Web site. Both of these sources offer a wealth of information to be explored.

© Copyright IBM Corporation 2005
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)