Extracted from:

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

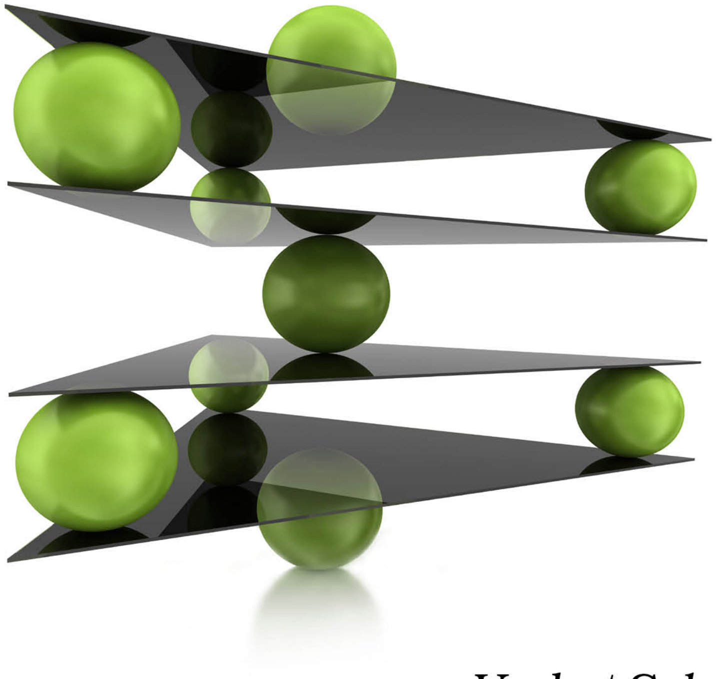
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors



Venkat Subramaniam

edited by Brian P. Hogan

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-76-0
Printed on acid-free paper.
Book version: P1.0—August 2011

To Mom and Dad, for teaching the values of integrity, honesty, and diligence.

Speed. Aside from caffeine, nothing quickens the pulse of a programmer as much as the blazingly fast execution of a piece of code. How can we fulfill the need for computational speed? Moore's law takes us some of the way, but multicore is the real future. To take full advantage of multicore, we need to program with concurrency in mind.

In a concurrent program, two or more actions take place simultaneously. A concurrent program may download multiple files while performing computations and updating the database. We often write concurrent programs using threads in Java. Multithreading on the Java Virtual Machine (JVM) has been around from the beginning, but how we program concurrency is still evolving, as we'll learn in this book.

The hard part is reaping the benefits of concurrency without being burned. Starting threads is easy, but their execution sequence is nondeterministic. We're soon drawn into a battle to coordinate threads and ensure they're handling data consistently.

To get from point A to point B quickly, we have several options, based on how critical the travel time is, the availability of transport, the budget, and so on. We can walk, take the bus, drive that pimped-up convertible, take a bullet train, or fly on a jet. In writing Java for speed, we've also got choices.

There are three prominent options for concurrency on the JVM:

- What I call the "synchronize and suffer" model
- The Software-Transactional Memory model
- The actor-based concurrency model

I call the familiar Java Development Kit (JDK) synchronization model "synchronize and suffer" because the results are unpredictable if we forget to synchronize shared mutable state or synchronize it at the wrong level. If we're lucky, we catch the problems during development; if we miss, it can come out in odd and unfortunate ways during production. We get no compilation errors, no warning, and simply no sign of trouble with that ill-fated code.

Programs that fail to synchronize access to shared mutable state are broken, but the Java compiler won't tell us that. Programming with mutability in pure Java is like working with the mother-in-law who's just waiting for you to fail. I'm sure you've felt the pain.

There are three ways to avoid problems when writing concurrent programs:

- Synchronize properly.
- Don't share state.
- Don't mutate state.

If we use the modern JDK concurrency API, we'll have to put in significant effort to synchronize properly. STM makes synchronization implicit and greatly reduces the chances of errors. The actor-based model, on the other hand, helps us avoid shared state. Avoiding mutable state is the secret weapon to winning concurrency battles.

In this book, we'll take an example-driven approach to learn the three models and how to exploit concurrency with them.

Who's This Book For?

I've written this book for experienced Java programmers who are interested in learning how to manage and make use of concurrency on the JVM, using languages such as Java, Clojure, Groovy, JRuby, and Scala.

If you're new to Java, this book will not help you learn the basics of Java. There are several good books that teach the fundamentals of Java programming, and you should make use of them.

If you have fairly good programming experience on the JVM but find yourself needing material that will help further your practical understanding of programming concurrency, this book is for you.

If you're interested only in the solutions directly provided in Java and the JDK—Java threading and the concurrency library—I refer you to two very good books already on the market that focus on that: Brian Goetz's *Java Concurrency in Practice* [Goe06] and Doug Lea's *Concurrent Programming in Java* [Lea00]. Those two books provide a wealth of information on the Java Memory Model and how to ensure thread safety and consistency.

My focus in this book is to help you use, but also move beyond, the solutions provided directly in the JDK to solve some practical concurrency problems. You will learn about some third-party Java libraries that help you work easily with isolated mutability. You will also learn to use libraries that reduce complexity and error by eliminating explicit locks.

My goal in this book is to help you learn the set of tools and approaches that are available to you today so you can sensibly decide which one suits you the best to solve your immediate concurrency problems.

What's in This Book?

This book will help us explore and learn about three separate concurrency solutions—the modern Java JDK concurrency model, the Software Transactional Memory (STM), and the actor-based concurrency model.

This book is divided into five parts: Strategies for Concurrency, Modern Java/JDK Concurrency, Software Transactional Memory, Actor-Based Concurrency, and an epilogue.

In Chapter 1, *The Power and Perils of Concurrency*, on page ?, we will discuss what makes concurrency so useful and the reasons why it's so hard to get it right. This chapter will set the stage for the three concurrency models we'll explore in this book.

Before we dive into these solutions, in Chapter 2, *Division of Labor*, on page ? we'll try to understand what affects concurrency and speedup and discuss strategies for achieving effective concurrency.

The design approach we take makes a world of difference between sailing the sea of concurrency and sinking in it, as we'll discuss in Chapter 3, *Design Approaches*, on page ?.

The Java concurrency API has evolved quite a bit since the introduction of Java. We'll discuss how the modern Java API helps with both thread safety and performance in Chapter 4, *Scalability and Thread Safety*, on page ?.

While we certainly want to avoid shared mutable state, in Chapter 5, *Taming Shared Mutability*, on page ? we'll look at ways to handle the realities of existing applications and things to keep in mind while refactoring legacy code.

We'll dive deep into STM in Chapter 6, *Introduction to Software Transactional Memory*, on page ? and learn how it can alleviate most of the concurrency pains, especially for applications that have very infrequent write collisions.

We'll learn how to use STM in different prominent JVM languages in Chapter 7, *STM in Clojure, Groovy, Java, JRuby, and Scala*, on page ?.

In Chapter 8, *Favoring Isolated Mutability*, on page ?, we'll learn how the actor-based model can entirely remove concurrency concerns if we can design for isolated mutability.

Again, if you're interested in different prominent JVM languages, you'll learn how to use actors from your preferred language in Chapter 9, *Actors in Groovy, Java, JRuby, and Scala*, on page ?.

Finally, in Chapter 10, *Zen of Programming Concurrency*, on page ?, we'll review the solutions we've discussed in this book and conclude with some takeaway points that can help you succeed with concurrency.

Is it Concurrency or Parallelism?

There's no clear distinction between these two terms in the industry, and the number of answers we'll hear is close to the number of people we ask for an explanation (and don't ask them concurrently…or should I say in parallel?).

Let's not debate the distinction here. We may run programs on a single core with multiple threads and later deploy them on multiple cores with multiple threads. When our code runs within a single JVM, both these deployment options have some common concerns—how do we create and manage threads, how do we ensure integrity of data, how do we deal with locks and synchronization, and are our threads crossing the memory barrier at the appropriate times…?

Whether we call it concurrent or parallel, addressing these concerns is core to ensuring that our programs run correctly and efficiently. That's what we'll focus on in this book.

Concurrency for Polyglot Programmers

Today, the word *Java* stands more for the platform than for the language. The Java Virtual Machine, along with the ubiquitous set of libraries, has evolved into a very powerful platform over the years. At the same time, the Java language is showing its age. Today there are quite a few interesting and powerful languages on the JVM—Clojure, JRuby, Groovy, and Scala, to mention a few.

Some of these modern JVM languages such as Clojure, JRuby, and Groovy are dynamically typed. Some, such as Clojure and Scala, are greatly influenced by a functional style of programming. Yet all of them have one thing in common—they're concise and highly expressive. Although it may take a bit of effort to get used to their syntax, the paradigm, or the differences, we'll mostly need less code in all these languages compared with coding in Java. What's even better, we can mix these languages with Java code and truly be a polyglot programmer—see Neal Ford's "Polyglot Programmer" in Appendix 2, *Web Resources*, on page ?.

In this book we'll learn how to use the java.util.concurrent API, the STM, and the actor-based model using Akka and GPars. We'll also learn how to program concurrency in Clojure, Java, JRuby, Groovy, and Scala. If you program in or are planning to pick up any of these languages, this book will introduce you to the concurrent programming options in them.

Examples and Performance Measurements

Most of the examples in this book are in Java; however, you will also see quite a few examples in Clojure, Groovy, JRuby, and Scala. I've taken extra effort to keep the syntactical nuances and the language-specific idioms to a minimum. Where there is a choice, I've leaned toward something that's easier to read and familiar to programmers mostly comfortable with Java.

The following are the version of languages and libraries used in this book:

- Akka 1.1.3 (http://akka.io/downloads)
- Clojure 1.2.1 (http://clojure.org/downloads)
- Groovy 1.8 (http://groovy.codehaus.org/Download)
- GPars 0.12 (http://gpars.codehaus.org)
- Java SE 1.6 (http://www.java.com/en/download)
- JRuby 1.6.2 (http://jruby.org/download)
- Scala 2.9.0.1 (http://www.scala-lang.org/downloads)

When showing performance measures between two versions of code, I've made sure these comparisons are on the same machine. For most of the examples I've used a MacBook Pro with 2.8GHz Intel dual-core processor and 4GB memory running Mac OS X 10.6.6 and Java version 1.6 update 24. For some of the examples, I also use an eight-core Sunfire 2.33GHz processor with 8GB of memory running 64-bit Windows XP and Java version 1.6.

All the examples, unless otherwise noted, were run in server mode with the "Java HotSpot(TM) 64-Bit Server VM" Java virtual machine.

All the examples were compiled and run on both the Mac and Windows machines mentioned previously.

In the listing of code examples, I haven't shown the import statements (and the package statements) because these often become lengthy. When trying the code examples, if you're not sure which package a class belongs to, don't worry, I've included the full listing on the code website. Go ahead and download the entire source code for this book from its website (http://pragprog.com/titles/vspcon).

Acknowledgments

Several people concurrently helped me to write this book. If not for the generosity and inspiration from some of the great minds I've come to know and respect over the years, this book would have remained a great idea in my mind.

I first thank the reviewers who braved to read the draft of this book and who offered valuable feedback—this is a better book because of them. However, any errors you find in this book are entirely a reflection of my deficiencies.

I benefited a great deal from the reviews and shrewd remarks of Brian Goetz (@BrianGoetz), Alex Miller (@puredanger), and Jonas Bonér (@jboner). Almost every page in the book was improved by the thorough review and eagle eyes of Al Scherer (@al_scherer) and Scott Leberknight (@sleberknight). Thank you very much, gentlemen.

Special thanks go to Raju Gandhi (@looselytyped), Ramamurthy Gopalakrishnan, Paul King (@paulk_asert), Kurt Landrus (@koctya), Ted Neward (@tedneward), Chris Richardson (@crichardson), Andreas Rueger, Nathaniel Schutta (@ntschutta), Ken Sipe (@kensipe), and Matt Stine (@mstine) for devoting your valuable time to correct me and encourage me at the same time. Thanks to Stuart Halloway (@stuarthalloway) for his cautionary review. I've improved this book, where possible, based on his comments.

The privilege to speak on this topic at various NFJS conferences helped shape the content of this book. I thank the NFJS (@nofluff) director Jay Zimmerman for that opportunity and my friends on the conference circuit both among speakers and attendees for their conversations and discussions.

I thank the developers who took the time to read the book in the beta form and offer their feedback on the book's forum. Thanks in particular to Dave Briccetti (@dcbriccetti), Frederik De Bleser (@enigmeta), Andrei Dolganov, Rabea Gransberger, Alex Gout, Simon Sparks, Brian Tarbox, Michael Uren, Dale Visser, and Tasos Zervos. I greatly benefited from the insightful comments, corrections, and observations of Rabea Gransberger.

Thanks to the creators and committers of the wonderful languages and libraries that I rely upon in this book and to program concurrent applications on the JVM.

One of the perks of writing this book was getting to know Steve Peter, who endured the first draft of this book as the initial development editor. His sense of humor and attention to detail greatly helped during the making of

this book. Thank you, Steve. It was my privilege to have Brian P. Hogan (@bphogan) as the editor for this book. He came up to speed quickly, made observations that encouraged me, and, at the same time, provided constructive comments and suggestions in areas that required improvements. Thank you, Brian.

I thank the entire Pragmatic Bookshelf team for their efforts and encouragement along the way. Thanks to Kim Wimpsett, Susannah Pfalzer (@spfalzer), Andy Hunt (@pragmaticandy), and Dave Thomas (@pragdave) for their help, guidance, and making this so much fun.

None of this would have been possible without the support of my wife—thank you, Kavitha, for your incredible patience and sacrifice. I got quite a bit of encouragement from my sons, Karthik and Krupa; thank you, guys, for being inquisitive and frequently asking whether I'm done with the book. Now I can say yes, and it's where it belongs—in the hands of programmers who'll put it to good use.