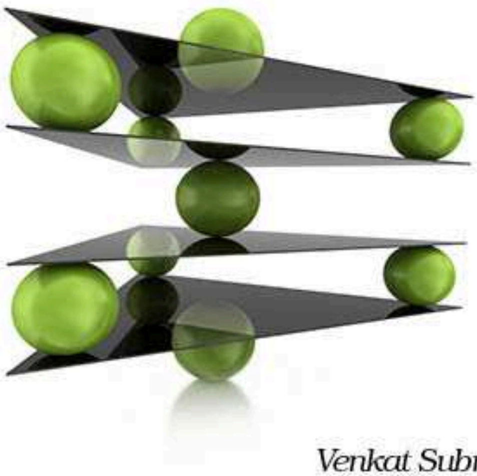
The Pragmatic Programmers

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
Raleigh, North Carolina   Dallas, Texas

Printed in the United States of America.

ISBN-10: 1-934356-76-X

ISBN-13: 978-1-934356-76-0

Printed on acid-free paper.

B1.0 printing, March 23, 2011

Version: 2011-3-18

Contents

Preface **9**
Who's this book for? 10
What's in this book? 11
Concurrency for Polyglot Programmers 12
Examples and Performance Measurements 13
Acknowledgements 14

1 The Power and Perils of Concurrency **16**
1.1 Threads—The Flow of Execution 17
1.2 The Power of Concurrency 17
1.3 The Perils of Concurrency 20
1.4 Recap 27

I Strategies for Concurrency **28**

2 Division of Labor **29**
2.1 From Sequential to Concurrent 29
2.2 Concurrency in IO Intensive Apps 32
2.3 Speedup for the IO Intensive App 39
2.4 Concurrency in Computationally Intensive Apps . . 40
2.5 Speedup for the Computationally Intensive App . . 46
2.6 Strategies for Effective Concurrency 48
2.7 Recap 49

3 Design Approaches **50**
3.1 Dealing with State 50
3.2 Exploring Design Options 51
3.3 Shared Mutable Design 52
3.4 Isolated Mutable Design 53
3.5 Purely Immutable Design 53
3.6 Persistent/Immutable Data Structures 54
3.7 Selecting a Design Approach 59

3.8 Recap 59

II Modern Java/JDK Concurrency **60**

4 Scalability and Thread Safety **61**
4.1 Managing Threads with ExecutorService 62
4.2 Coordinating Threads 64
4.3 Exchanging Data 73
4.4 Scalable Collections 76
4.5 Lock vs. synchronized 79
4.6 Recap 83

5 Taming Shared Mutability **85**
5.1 Shared Mutability != public 85
5.2 Spotting Concurrency Issues 86
5.3 Preserve Invariant 88
5.4 Mind Your Resources 89
5.5 Ensure Visibility 90
5.6 Enhance Concurrency 92
5.7 Ensure Atomicity 94
5.8 Recap 97

III Software Transaction Memory **98**

6 Introduction to Software Transactional Memory **99**
6.1 Synchronization Damns Concurrency 100
6.2 The Deficiency of the Object Model 100
6.3 Separation of Identity and State 101
6.4 Software Transaction Memory 103
6.5 Transactions in STM 105
6.6 Concurrency using STM 106
6.7 Concurrency using Akka/Multiverse STM 112
6.8 Creating Transactions 114
6.9 Creating Nested Transactions 121
6.10 Configuring Transactions 129
6.11 Blocking Transactions—Sensible Wait 132
6.12 Commit and Rollback Events 135
6.13 Collections and Transactions 139
6.14 Dealing with Write Skew Anomaly 143
6.15 Limitations of STM 146

6.16 Recap 150

7 STM in Clojure, Groovy, Java, JRuby, and Scala 151

7.1 Clojure STM 152
7.2 Groovy Integration 152
7.3 Java Integration 157
7.4 JRuby Integration 160
7.5 Choices in Scala 167
7.6 Recap 170

IV Actor-based Concurrency 171

8 Favoring Isolated Mutability 172

8.1 Isolating Mutability using Actors 173
8.2 Actor Qualities 175
8.3 Creating Actors 176
8.4 Sending and Receiving Messages 183
8.5 Working with Multiple Actors 188
8.6 Coordinating Actors 193
8.7 Using Typed Actors 201
8.8 Typed Actors and Murmurs 206
8.9 Mixing Actors and STM 212
8.10 Using Transactors 213
8.11 Coordinating Typed Actors 221
8.12 Limitations of Actor Based Model 227
8.13 Recap 228

9 Actors in Clojure, Groovy, Java, JRuby, and Scala 229

9.1 Clojure Agents 229
9.2 Actors in Groovy with GPars 234
9.3 Java Integration 244
9.4 JRuby Akka integration 245
9.5 Choices in Scala 249
9.6 Recap 249

V Epilogue 250

10 Zen of Programming Concurrency **251**
10.1 Exercise Your Options 252
10.2 Concurrency: Programmer's Guide 252
10.3 Concurrency: Architect's Guide 254
10.4 Choose Wisely 255

A Web Resources **256**

B Bibliography **258**

Index **260**

Preface

Speed. Aside from caffeine, nothing quite quickens the pulse of a programmer as much as the blazingly fast execution of a piece of code. How can you fulfill your need for computational speed? Moore's Law takes you some of the way but, multicore is the real future, and to take full advantage of multicore, you 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 you'll learn in this book.

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

When you want to get from point A to point B quickly, you've got several options, based on how critical the travel time is, the availability of transport, your budget, and so on. You can walk, take the bus, drive your pimped-up convertible, take a bullet train, or a jet. In writing Java for speed, you'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 (STM) model, and
- the actor-based concurrency model.

I call the familiar Java Development Kit (JDK) synchronization model "synchronize and suffer" because the result is unpredictable if you forget to synchronize shared mutable state or synchronize it at the wrong level. If you're lucky, you catch the problems during development; if you miss, it can come out in odd and unfortunate ways during production. You get no compilation errors, no warning, 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 you 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 you use the modern JDK concurrency API, you will have to put significant efforts to ensure that you've synchronized properly. STM makes synchronization implicit and greatly reduces the chances of errors. The actor-based model, on the other hand, helps you 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 like Java, Clojure, Groovy, JRuby, and Scala.

If you're new to Java, this book will not help you to learn the basics of Java. There are several good books that will help you learn 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 only interested in the solutions directly provided in Java and the JDK—the Java Threading and concurrency library—I refer you to two very good books already in 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 suites you the best to solve your immediate concurrency problems.

What's in this book?

The rest of this book will help you 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 Transaction Memory, Actor-based Concurrency, and an Epilogue.

In Chapter 1, *The Power and Perils of Concurrency*, on page 16 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 29 we'll try to understand what affects concurrency and speedup, and discuss strategies for achieving effective concurrency.

The design approach you 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 50.

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 61.

While we certainly want to avoid shared mutable state, in Chapter 5, *Taming Shared Mutability*, on page 85 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 99 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 151.

In Chapter 8, *Favoring Isolated Mutability*, on page 172, we'll learn how the actor based model can entirely remove concurrency concerns if you 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 Clojure, Groovy, Java, JRuby, and Scala*, on page 229.

Finally in Chapter 10, *Zen of Programming Concurrency*, on page 251, we'll review the solutions we've discussed in this book and conclude with some take-away points that can help you succeed with concurrency.

Concurrency for Polyglot Programmers

The word Java today stands more for the platform than for the language. The Java Virtual Machine, along with the ubiquitous set of libraries, has turned 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 like Clojure, JRuby, and Groovy, are dynamically typed. Some, like Clojure and Scala, are greatly influenced by functional style of programming. Yet all of them have one thing in common—they're concise and highly expressive. While 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 A, on page 256.

In this book you'll learn how to use the java.util.concurrent API, the STM, and the actor-based model using Akka and GPars. You'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 syntax of these language to a minimal. Where there is a choice, rather than using a syntax or construct specific to a particular language, I've leaned towards something that's easier to read and familiar to programmers mostly comfortable with Java.

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

- Akka 1.0 (https://github.com/jboner/akka/downloads)
- Clojure 1.2 (http://clojure.org/downloads)
- Groovy 1.7.10 (http://groovy.codehaus.org/Download)
- GPars 0.11 (http://gpars.codehaus.org)
- Java SE 1.6 (http://www.java.com/en/download)
- JRuby 1.6.0 (http://jruby.org/download)
- Scala 2.8.1 (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 2.8 GHz Intel dual core processor and 4 GB memory running Mac OS X 10.6.6 and Java version 1.6 update 24. For some of the examples, I also use an 8-core Sunfire 2.33 GHz processor with 8 GB of memory running 64-bit Windows XP and Java version 1.6.

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

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

In the listing of code examples, I've not shown the import statements as these often become lengthy. When trying out 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 the website for this book http://pragprog.com/titles/vspcon.

Acknowledgements

Several people concurrently helped me 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 offered valuable feedback—this is a better book due to them. However, any errors you find in this book is 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 Boner (@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 to Raju Gandhi (@looselytyped), Ramamurthy Gopalakrishnan, 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.

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 forum for the book.

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 details 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 if I'm done with the book. Now I can say yes and its where it belongs—in the hands of programmers who'll put it to good use.

Need and struggle are what excite and inspire us.
► William James

Chapter 1

The Power and Perils of Concurrency

You’ve promised the boss that you’ll turn the new powerful multicore processor into a blazingly fast workhorse for your application. You’d love to exploit the power on hand and beat your competition with faster, responsive application that provides great user experience. Those gleeful thoughts are interrupted by your colleague’s cry for help, he’s run into yet another synchronization issue.

Most programmers have a love-hate relationship with concurrency.

If concurrency is hard, why do it? The processing power we have at our disposal, at such an affordable cost, is something that our parents could only dream of. We can exploit the ability to run multiple concurrent tasks to create stellar applications. We have the ability to write applications that can provide a great user experience by staying a few steps ahead of the user. Features that would’ve made apps sluggish a decade ago are quite practical today. To realize these, however, we’d have to program concurrency.

I ask you in this chapter to take a short amount of time to refresh the reasons to exploit concurrency and understand the perils that this path is mired with. Learn these now as you prepare to take the exciting journey of exploring the options for concurrency presented in this book.

1.1 Threads—The Flow of Execution

As a Java programmer, you already know that a thread is a flow of execution in a process. When you run a program, there is at least one thread of execution for its process. You can create threads to start additional flows of execution, in order to perform additional tasks concurrently. The libraries or framework you use may also start additional threads behind the scenes, depending on their need.

When multiple threads run as part of a single application, or a JVM, you have multiple tasks or operations running concurrently. A concurrent application makes use of multiple threads or concurrent flows of execution.

On a single processor, these concurrent tasks are often multiplexed or multitasked. That is, the processor rapidly switches between the context of each flow of execution. However, only one thread, and hence only one flow of execution, is performed at any given instance. On a multicore processor, however, more than one flow of execution (thread) is performed at any given instance. That number depends on the number of cores available on the processor and the number of concurrent threads for an application depends on the number of cores associated with its process.

1.2 The Power of Concurrency

You'd be interested in concurrency for two reasons: to make your application responsive/improve the user experience and to make it faster.

Making Apps More Responsive

When you start an application, the main thread of execution often takes on multiple responsibilities, sequentially, depending on the actions you ask it to perform: receive input from a user, read from a file, perform some calculations, access a web service, update a database, display a response to the user, etc.; if each of these operations only takes fractions of a second, then there may be no real need to introduce additional flows of execution, a single thread may be quite adequate for your needs.

In most non-trivial applications, however, these operations may not be that quick. Your calculation may take anywhere from a couple of seconds to a few minutes. Your request for data from that web service encounters some network delay and your thread waits for the response to arrive. While this is happening, there's no way for the users of your application to interact with or interrupt your application as the single thread is held on some operation to finish.

Let's consider an example that illustrates the need for more than one thread and how it impacts responsiveness. I want to time an event and so decided to create a stopwatch application, we can click a button to start the watch and it runs until we click the button again. A naively written code for this is shown next (only the action handler for the button is shown, you can download the full code from the website for this book):

Download introduction/NaiveStopWatch.java

```
//This will not work
public void actionPerformed(ActionEvent event) {
  if (running) stopCounting(); else startCounting();
}

private void startCounting() {
  startStopButton.setText("Stop");
  running = true;
  for(int count = 0; running; count++) {
    timeLabel.setText(String.format("%d", count));
    try { Thread.sleep(1000); } catch(InterruptedException ex) {}
  }
}

private void stopCounting() {
  running = false;
  startStopButton.setText("Start");
}
```

When we run the little stopwatch application, a window with a "Start" button and a "0" label will appear. Unfortunately, when we click on the button, we'll not see any change—the button does not change to "Stop" and the label does not show the time count. What's worse, the application will not even respond to a quit request.

When clicked on the Start button, the main event dispatch thread went into the event handler actionPerformed(), but it was held hostage by the method startCounting() as it started counting. Now, as we click on buttons or try to quit, those events are dropped into the event queue, but the main thread is too busy and will not respond to those events—ever.

We'd need an additional thread, or a timer which in turn would use an additional thread, to make the application responsive. We need to delegate the task of counting and relieve the main event thread of that responsibility.

Not only can threads help make applications responsive, they can help enhance the user experience. Your application can look ahead at operations the user may perform next and carry out the necessary actions, like indexing or caching some data the user would need.

Making Apps Faster

Review the operations that your application performs. Do you see operations that are currently performed sequentially, one after the other, that can be performed concurrently? You can make your application faster by running each of those operations in separate threads.

Quite a few kinds of applications can run faster by using concurrency. Among these are services, computationally-intensive applications, and data-crunching applications.

Services

You may be building an application service that has to receive and process a large volume of independent service requests. Maybe your application is responsible for a large number of invoices from different vendors and for applying some rules and business workflow on them. Each invoice is independent of the others, and the requests to process each invoice may appear at any time and in any order. Processing these invoices sequentially will not yield the throughput nor utilize the resources well. Your application needs to process these invoices concurrently.

Computationally-Intensive Apps

If your application involves intensive scientific computing, you may find a way to break the problem into pieces so that several pieces can be computed concurrently and then the partial results merged together. There is a variety of problems that lend themselves to the divide-and-conquer approach, and these would readily benefit from your ability to write concurrent programs.

Data Crunchers

Maybe your application has to receive data from several sources or receive several independent pieces of data from one source. Instead of receiving data sequentially, you can request for them concurrently. You may be able to start processing these data as soon as you receive them. Such applications can realize a greater speedup by using concurrency.

I was once asked to build a personal finance application that has to go out to a web service to get the current price and other details for a number of stocks. We need to present the user with total asset value and details on the volume of trading for each stock, and so on. For a wealthy user, the application may track shares in 100 different stocks. During heavy traffic time, it may take a few seconds to receive the information from the web. That would turn into a few minutes of wait for our user before we receive all the data and begin processing. If we delegate the request to multiple threads, we could bring this wait time to as low as a mere second or two assuming the network delay per request is a second or two and the system we're running the app on has adequate resources and capabilities to spawn hundreds of threads.

More Responsive or Faster?

In general, concurrency can help make apps responsive, reduce latency, and increase throughput. Whether an application will benefit from a speed gain depends on how it will lend itself to exploit concurrency. There's a significant number of applications that do fall into this category. On the other hand, most applications typically block on user input or wait for data access. By delegating those to another thread, they can gain responsiveness.

To use concurrency in order to make apps responsive or faster, we have to design applications differently. This involves rethinking how problems can be solved, how to partition it, and how to effectively utilize the programming APIs to safely implement concurrency. The focus of this book is to help you with this last part, to help you understand and make good use of the concurrency libraries and APIs available to you on the Java platform.

1.3 The Perils of Concurrency

Right now, you're probably thinking, "I can get a better throughput by breaking up my problem and letting multiple threads work on these parts." Unfortunately, problems rarely can be divided into isolated parts that can be run totally independent of each other. Often, you can perform some operations independently, but then have to merge the partial results to get the final result. This requires threads to communicate the partial results and therefore possibly wait for those results to be ready. This requires coordination between threads and can lead to synchronization and locking woes.

Some of the most common problems you'll run into are starvation, deadlock, and race conditions. The first two are somewhat easier to detect and even avoid. The last one, however, is a real nemesis that should be eliminated at the root.

Starvation and Deadlocks

Your application is about to perform a critical task, but requires user confirmation. It pops a dialog just as the user steps out to lunch. While the user enjoys a good meal, your application has entered a phase of starvation. Starvation occurs when a thread waits for an event that may take a very long time or forever to happen. It can happen when your thread waits for an input from a user, for some external event to occur, or another thread to release a lock. The thread will stay alive while it waits doing nothing. You can prevent starvation by placing a timeout. You design your solution in such a way that the thread waits for only a finite amount of time. If the input does not arrive, the event does not happen, or your thread does not gain the lock within that time, your thread bails out and takes an alternate action to make progress.

You run into deadlock if a group of two or more threads are waiting on each other for some action or resource. Placing a timeout, unfortunately, will not help avoid the repeat of deadlock. It's possible that each thread gives up its resources, only to repeat its steps that leads again into a deadlocd—See "The Dining Philosophers Problem" in Appendix A, on page 256. Tools like JConsole can help detect deadlocks and you can prevent deadlock by acquiring resources in a specific order. A better alternative, in the first place, would be to avoid explicit locks and the mutable state that goes with them. I'll show you how to do that later in the book.

Race Conditions

If two threads compete to use the same resource or data, you have a race condition. A race condition doesn't just happen when two threads modify data. It can happen even when one is changing data while the other is trying to read it. Race conditions can render your program behavior unpredictable, produce incorrect execution, and yield incorrect results.

There are two forces that can lead to race conditions—the Just-in-time (JIT) compiler optimization and the Java Memory Model. For an exceptional treatise on the topic of Java Memory Model and how it affects concurrency, refer to Brian Goetz's seminal book *Java Concurrency in Practice* [Goe06].

Let's take a look at a fairly simple example that illustrates the problem. In the following code, the main thread creates a thread, sleeps for 2 seconds, and sets the flag done to true. The thread created, in the mean time, loops over the flag, as long as it's false. Go ahead, compile and run the code and see what happens.

Download introduction/RaceCondition.java

```
package com.agiledeveloper;

public class RaceCondition {
  private static boolean done;

  public static void main(final String[] args) throws InterruptedException{
      new Thread(
              new Runnable() {
                  public void run() {
                      int i = 0;
                      while(!done) { i++; }
                      System.out.println("Done!");
                  }
              }
      ).start();

      System.out.println("OS: " + System.getProperty("os.name"));
      Thread.sleep(2000);
      done = true;
      System.out.println("flag done set to true");
  }
}
```

If you run that little program on Windows 7 (32-bit version), using the command java RaceCondition, you'll notice something tlike this, the order of output may differ on your machine:

```
OS: Windows 7
flag done set to true
Done!
```

If you tried the same command on a Mac, you'd notice that the thread that's watching over the flag never finished as you see in the output:

```
OS: Mac OS X
flag done set to true
```

Wait, don't put the book down and tweet "Windows Rocks, Mac sucks!" The problem is a bit deeper than the two runs above revealed.

Let's try again, this time, on Windows, run the program using the command java -server RaceCondition (asking it to be run in server mode on the Windows) and on the Mac, run it using the command java -d32 RaceCondition (asking it to be run in client mode on the Mac).

On Windows, you'd see something like:

```
OS: Windows 7
flag done set to true
```

However, now on the Mac, you'll see something like:

```
OS: Mac OS X
Done!
flag done set to true
```

So the behavior is consistent on both platforms—the program terminates in client mode on both. Java runs in client mode by default on Windows, and in server mode by default on the Mac. When run in server mode, however, the second thread never sees the change to the flag done, even though the main thread set its value to true. But, don't be quick to blame the Java server Just-in-time (JIT) compiler—it's a powerful tool that works hard to optimize code to make it run faster. All that this means is that broken programs may appear to work on some platforms and fail on others.

This is a problem of visibility.

Know your Visibility—Understand the Memory Barrier

The problem with the previous example is that the change by the main thread to the field done may not be visible to the thread we created. First, the JIT compiler may optimize the while loop; after all it does not see variable done changing within the context of the thread. Furthermore, the second thread may end up reading the value of the flag from its registers or cache instead of going to the memory—as a result it may never see the change made by the first thread to this flag—see the *Joe Asks...* on the following page.

A quick fix to the problem is to mark the flag done as volatile. You can change

Joe Asks...

What's this Memory Barrier?

Simply put, it is the copying from local or working memory to main memory.

A change made by one thread is guaranteed to be visible to another thread only if the writing thread crosses the memory barrier[*] and then the reading thread crosses the memory barrier. synchronized and volatile force that the changes are globally visible on a timely basis, these help cross the memory barrier—accidentally or intentionally.

The changes are first made locally in the registers and caches, and then cross the memory barrier as they are copied to the main memory. The sequence or ordering of these crossing is called *happens-before*—see "The Java Memory Model" Appendix A, on page 256 and Brian Goetz's *Java Concurrency in Practice* (Goe06).

The write has to *happens-before* the read, meaning the writing thread has to cross the memory barrier before the reading thread does, for the change to be visible.

There are quite a few operations in the concurrency API that implicitly crosses the memory barrier. volatile, synchronized, methods on Thread like start() and interrupt(), and methods on ExecutorService, and some synchronization facilitators like CountDownLatch.

[*]. See Doug Lea's article "The JSR-133 Cookbook for Compiler Writers" in the Appendix A, on page 256.

```
private static boolean done;
```

to

```
private volatile static boolean done;
```

The **volatile** keyword tells the JIT compiler not to perform any optimization that may affect the ordering of access to that variable. It warns that the variable may change behind the back of a thread and that each access, read or write, to this variable should bypass cache and go all the way to the memory. I call this a quick fix because arbitrarily making all variables volatile may avoid the problem but will result in very poor performance as every access has to cross the memory barrier. Also volatile does not help with atomicity when multiple fields are accessed, as the access to each of the volatile fields are separately handled and not coordinated into one access—this would leave a wide opportunity for threads to see partial changes to some fields and not the others.

An alternate way to handle this problem is to avoid direct access to the flag and channel all access through synchronized getter and setter as follows:

```
private static boolean done;
synchronized public static boolean getFlag() { return done; }
synchronized public static void setFlag(boolean flag) { done = flag; }
```

The **synchronized** marker helps here, since it is one of the primitives that makes the calling threads cross the memory barrier both when they enter and when they exit the synchronized block. A thread is guaranteed to see the change made by another thread if both threads synchronize on the same instance and the change-making thread happens-before the other thread—again see the *Joe Asks...* on the previous page.

Avoid Shared Mutability

Unfortunately, the consequence of forgetting to do either—using volatile or synchronized where needed—is quite unpredictable. The real worry is not that you'd forget to synchronize. The core problem is that you're dealing with shared mutability.

We're quite used to programming Java applications with mutability—creating and modifying state of an object by changing its fields. However, great books such as Joshua Bloch's *Effective Java* [Blo08] have advised us to promote immutability. Immutability can help us avoid the problem at its root.

Mutability in itself is not entirely bad, though it's often used in ways that can lead to trouble. Sharing is a good thing (remember mom always advised us to share). While these two things by themselves are OK, mixing them together is not.

Suppose that you have a non-final (mutable) field. Each time a thread changes the value, you have to consider if you have to put the change back to the memory or leave it in the registers/cache. Each time you read the field, you need to be concerned if you read the latest valid value or a stale value left behind in the cache. You need to ensure the changes to variables are atomic, that is, threads don't see partial changes. Furthermore, you need to worry about protecting multiple threads from changing the data at the same time.

For an application that deals with mutability, every single access to shared mutable state must be verified to be correct. Even if one of them is broken, the entire application is broken. This is a tall order—for your concurrent app to fall apart, only a single line of code that deals with concurrency needs to take a wrong step. In fact, significant number of concurrent Java apps are broken and we simply don't know it.

Now suppose you have a final (immutable) field to an immutable instance[1] and you let multiple threads access it. Such sharing has no hidden problems. Any thread can read it and upon first access, gets a copy of the value that it can keep locally in its cache. Since the value is immutable, subsequent access to the value from the local cache is quite adequate and you can even enjoy good performance.

Shared mutability is pure evil, avoid it.

You may wonder how we can make applications do anything if we can't change a thing. This is a valid concern, but we need to design our applications around shared immutability. One approach is to keep the mutable state well encapsulated and only share immutable data. An alternate approach, promoted by pure functional languages, is to make everything immutable but use function composition. In this approach we apply a series of transformations where we transition from one immutable state to another immutable state. Yet another approach is to use a library that will watch over the changes and warn us of any

1. For example, instances of String, Integer, and Long are immutable in Java, while instances of StringBuilder and ArrayList are mutable.

violations. We'll look at these techniques using examples of problems that we'll solve using concurrency through out this book.

1.4 Recap

Whether you're writing an interactive client-side desktop application or a high performance service for the backend, concurrency will play a vital role in your programming efforts, should you reap the benefits of the evolving hardware trends and multicore processors. It's a way to influence the user experience, responsiveness, and speed of applications that run on these powerful machines. The traditional programming model of concurrency on the JVM—dealing with shared mutability—is froth with problems. Besides creating threads, you have to work hard to prevent starvation, deadlocks, and race conditions—things that are hard to trace and easy to get wrong. By avoiding shared mutability you remove the problems at the root. As you'll learn how later in this book, lean toward shared immutability to make programming concurrency easy, safe, and fun.

Part I

Strategies for Concurrency

The most effective way to do it, is to do it.
► Amelia Earhart

Chapter 2
Division of Labor

The long awaited multicore processor arrives tomorrow, and you can't wait to see how the app you're building runs on it. You've run it several times on a single core, but you're eager to see the speedup on the new machine. Is the speedup going to be in proportion to the number of cores? More? Less? A lot less? I've been there and have felt the struggle to arrive at a reasonable expectation.

You should've seen my face the first time I ran my code on a multicore, and it performed much worse than I had expected. How could more cores yield slower speed? That was years ago, and I've grown wiser since and learned a few lessons along the way. Now I have better instinct and ways to gauge speedup that I'd like to share with you in this chapter.

2.1 From Sequential to Concurrent

You can't run a single-threaded application on a multicore processor and expect better results. You have to divide it and run multiple tasks concurrently. But, programs don't divide the same way and benefit from the same number of threads.

I have worked on scientific applications that are computation intensive and also on business applications that are IO intensive as they involve file, database, and web service calls. The nature of these two types of applications are different and so are the ways to make them concurrent.

We'll pick two applications to work with in this chapter. The first one is to compute the net asset value for a wealthy user—an IO intensive application. The second one is to compute the total number of primes within a range of numbers—a rather simple computation intensive program, but quite helpful to learn about concurrency. These two applications will help us learn how many threads to create, how to divide the problem, and how much speedup to expect.

Divide and Conquer

If you have hundreds of stocks to process, fetching them one at a time would be the easiest way... to lose the job. The user would stand fuming while our application chugs away processing each stock sequentially.

In order to gain speedup we need to divide the problem into concurrently running tasks. That involves creating these parts or tasks and delegating them to threads so they can run concurrently. For a large problem, you may create as many parts as you like, but you can't create too many threads as these are limited resources.

Decide the Number of Threads

For a large problem, you'd want to have at least as many threads as the number of available cores. This will ensure that as many cores as available to your process are put to work to solve your problem. You can easily find the number of available cores, all you need is a simple call from your code:

```
Runtime.getRuntime().availableProcessors();
```

So, the minimum number of threads is equal to the number of available cores. If all your tasks are computation intensive, then this is all you need. Having more threads will actually hurt in this case as cores would be context switching between your threads when there is still work to do. If your tasks are IO intensive, then you should have more threads.

When a task performs an IO operation its thread gets blocked. The processor immediately context switches to run other eligible threads. If you had only as many threads as the number of available cores, even though you have tasks to perform, they can't run as you've not scheduled them on threads for the processors to pick up.

Intuitively we can figure out the number of threads we should have. If your tasks spend 50% of the time being blocked, then the number of threads should be twice the number of available cores. If they spend less time begin blocked, that is they're computation intensive, then you should have fewer threads, but no less than the number of cores. If they spend more time blocked, that is they're IO intensive, then you should have more threads, several multiples of the number of cores.

So, we can compute the total number of threads we'd need as:

```
Number of threads = Number of Available Cores / (1 - Blocking Coefficient)
```

where the blocking coefficient is between 0 and 1.

A computation intensive task has a blocking coefficient of 0 and an IO intensive task has a value close to 1—a fully blocked task is doomed so we don't have to worry about the value reaching 1.

In order to determine the number of threads you need to know two things:

- The number of available cores
- The blocking coefficient of your tasks

The first one is easy to determine, you can look up for that information, even at runtime as you saw earlier. It takes some effort to determine the blocking coefficient. You can try to guess it. Or you can also use profiling tools or the java.lang.management API to determine the amount of time a thread spends on System/IO operations vs. on CPU intensive tasks.

Decide the Number of Parts

We know the number of threads we need for our concurrent applications. Now we have to decide how to divide the problem. Each part will be run concurrently, so, on first thought, we could have as many parts as the number of threads. That's a good start, but not adequate; we've ignored the nature of the problem being solved.

In the net asset value application, the effort to fetch the price for each stock is the same. So, dividing the total number of stocks into as many groups as the number of threads should be enough.

However, in the primes application, the effort to determine if a number is primes is not the same for all numbers. Even numbers fizzle out rather quickly and larger primes take more time than smaller primes. Taking the range of numbers and slicing them into as many groups as the number of threads would not help us get good performance. Some tasks would finish faster than others and poorly utilize the cores.

In other words, you'd want the parts to have even work distribution. You can spend a lot of time and effort to divide the problem so the parts have an fair distribution of load. The would, however, be a problem in two ways. One, this is hard, it would take a lot of effort and time. Second, the code to divide the problem into equal parts and distribute it across the threads is unlikely to be simple.

It turns out that more than even distribution of load across parts, keeping the cores busy on your problem is the key. While there's work left to be done, you must ensure no available core is left to idle, from your process point of view. So rather than splitting hair over even distribution of load across parts, we can achieve this by creating far more parts than the number of threads. Set the number of parts large enough so there's enough work for all the available cores to work on your program.

2.2 Concurrency in IO Intensive Apps

An IO intensive application has a large blocking coefficient and will benefit from more threads than the number of available cores.

Let's build the financial application I mentioned earlier. The (rich) users of the application want to determine the total net asset value of their shares at any given time. Let's work with one user who has shares in 40 stocks. We know the ticker symbols and the number of shares for each stock. From the web, we need to fetch the price of each share for each symbol. Let's take a look at writing the code for calculating the net asset value.

Sequential Computation of Net Asset Value

The first order of business is to fetch the price for ticker symbols. Thankfully Yahoo provides historic data that comes in handy. Here is the code to communicate with Yahoo's financial webservice to get the last trading price for a ticker symbol (as of previous day):[1]

Download divideAndConquer/YahooFinance.java

```java
public class YahooFinance {
  public static double getPrice(final String ticker) throws IOException {
    final URL url =
      new URL("http://ichart.finance.yahoo.com/table.csv?s=" + ticker);

    final BufferedReader reader = new BufferedReader(
      new InputStreamReader(url.openStream()));

    //Date,Open,High,Low,Close,Volume,Adj Close
    //2011-03-17,336.83,339.61,330.66,334.64,23519400,334.64
    String discardHeader = reader.readLine();
    final String data = reader.readLine();
    final String[] dataItems = data.split(",");
```

1. In the examples in this book we'll simply let the exceptions propagate instead of logging or handling them—be sure to handle exceptions properly in your production code.

```
    double priceIsTheLastValue = Double.valueOf(dataItems[dataItems.length - 1]);
    return priceIsTheLastValue;
  }
}
```

That was pretty straightforward, we sent out a request to http://ichart.finance.yahoo.com and parsed the result to obtain the price.

The next step is to get the price for each of the stocks our user owns and display the total net asset value—in addition we'll display the time it took for this operation as well.

Download divideAndConquer/AbstractNAV.java

```
public abstract class AbstractNAV {
  public static Map<String, Integer> readTickers() throws IOException {
    final BufferedReader reader = new BufferedReader(new FileReader("stocks.txt"));

    Map<String, Integer> stocks = new HashMap<String, Integer>();

    String stockInfo = null;
    while((stockInfo = reader.readLine()) != null) {
      final String[] stockInfoData = stockInfo.split(",");
      final String stockTicker = stockInfoData[0];
      final Integer quantity = Integer.valueOf(stockInfoData[1]);

      stocks.put(stockTicker, quantity);
    }

    return stocks;
  }

  public void timeAndComputeValue()
    throws ExecutionException, InterruptedException, IOException {
    final long start = System.nanoTime();

    final Map<String, Integer> stocks = readTickers();
    final double nav = computeNetAssetValue(stocks);

    final long end = System.nanoTime();

    final String value = new DecimalFormat("$##,##0.00").format(nav);
    System.out.println("Your net asset value is " + value);
    System.out.println("Time (seconds) taken " + (end - start)/1.0e9);
  }

  public abstract double computeNetAssetValue(final Map<String, Integer> stocks)
    throws ExecutionException, InterruptedException, IOException;
}
```

The readTickers() method of AbstractNAV reads the ticker symbol and the number of shares owned for each symbol from a file stocks.txt, part of which is shown next.

```
AAPL,2505
AMGN,3406
AMZN,9354
BAC,9839
BMY,5099
...
```

The timeAndComputeValue() times the call to an abstract method computeNetAssetValue() which will be implemented in a derived class. Finally it prints the total net asset value and the time it took to compute that.

The final step is to actually contact Yahoo Finance and compute the total net asset value, let's do that sequentially.

Download divideAndConquer/SequentialNAV.java

```java
public class SequentialNAV extends AbstractNAV {
  public double computeNetAssetValue(
    final Map<String, Integer> stocks) throws IOException {
    double netAssetValue = 0.0;
    for(String ticker : stocks.keySet()) {
      netAssetValue += stocks.get(ticker) * YahooFinance.getPrice(ticker);
    }
    return netAssetValue;
  }

  public static void main(final String[] args)
    throws ExecutionException, IOException, InterruptedException {
    new SequentialNAV().timeAndComputeValue();
  }
}
```

Let's run the SequentialNAV code and observe the output.

```
Your net asset value is $13,661,010.17
Time (seconds) taken 19.776223
```

The good news is we managed to help our user with the total asset value. However, our user is not very pleased. The displeasure may be partly due to the market conditions, but really it's mostly due to the wait she had to incur; it took close to 20 seconds[2] on my computer, with the network delay at the time of run, to get the results for only 40 stocks. I'm sure making this application concurrent will help with speedup and having a happier user.

2. More than a couple of seconds delay feels like eternity to users.

How to Make It Concurrent?

The application has very little computation to perform and spends most of the time waiting for responses from the web. There is really no reason to wait for one response to arrive before sending the next request. So, this application is a good candidate for concurrency: we'll likely get a good bump in speed.

In the sample run we had 40 stocks, but in reality we may have a higher number of stocks, even hundreds. We must first decide on the number of divisions and the number of threads to use. WebServices (in this case Yahoo finance) are quite capable of receiving and processing concurrent requests[3]. So our client side sets the real limit on the number of threads. Since the blocking coefficient is fairly high, each computation will spend more time waiting for response from the web than doing work, we can bump up the number of threads by several factors of the number of cores. Let's say the blocking coefficient is 0.9—each task blocks 90% of the time and works only 10% of its lifetime—then on two cores we can have (using the formula from Section 2.1, *Decide the Number of Threads*, on page 30) 20 threads and on a 8 core processor we can go up to 80 threads, assuming we have a lot of ticker symbols.

As far as the number of divisions, the workload is basically the same for each stock. So, we can simply have as many parts as we have stocks and schedule them over the number of threads.

Let's make the application concurrent and then study the effect of threads and partitions on the code.

Concurrent Computation of Net Asset Value

There are now two challenges. First we have to schedule the parts across threads. Second, we have to receive the partial results from each part to calculate the total asset value.

We may have as many divisions as the number of stocks for this problem. We need to maintain a pool of threads to schedule these divisions on. Rather than creating and managing individual threads, it's better to use a thread pool—they have better lifecycle and resource management, reduce startup and teardown costs, and are warm and ready to quickly start scheduled tasks.

3. To prevent denial-of-service attacks (and to upsell premium services) web services may restrict the number of concurrent requests from the same client. You may notice this with Yahoo finance when you exceed 50 concurrent requests.

Joe Asks...

Is there a reason to use the old Threading API in Java?

The old threading API has several deficiencies. You'd use and throw away the instances of the Thread class since they don't allow you to restart. To handle multiple tasks you typically create multiple threads instead of reusing them. If you decide to schedule multiple tasks on a thread, you had to write quite a bit of extra code to manage that. Either way was not efficient and scalable.

Methods like wait() and notify() require synchronization and are quite hard to get right when used to communicate between threads. The join() method leads you to be concerned about death of a thread rather than a task being accomplished.

In addition, the synchronized keyword lacks granularity. It doesn't give you a way to timeout if you did not gain the lock. It also doesn't allow concurrent multiple readers. Furthermore, it is very difficult to unit test for thread-safety if you use synchronized.

The newer generation of concurrency APIs in java.util.concurrent package, spearheaded by Doug Lea, among others, has nicely replaced the old threading API.

- Wherever you use Thread class and its methods, you can now rely upon the ExecutorService and related classes.
- Instead of using the synchronized construct, you can rely upon the Lock interface and its methods that give you better control over acquiring locks.
- Whereever you use wait/notify, you can now use synchronizers like CyclicBarrier and CountdownLatch.

If you've been programming in Java for a while, you've used Thread and **synchronized**. You have some alternatives to these since the arrival of Java 5—see the *Joe Asks...* on the previous page. The new generation Java concurrency API in java.util.concurrent is far superior.

In the modern concurrency API, the Executors serves as a factory to create different types of thread pools that you can manage using the ExecutorService interface. Some of the flavors include a single threaded pool that runs all scheduled tasks in a single thread, one after another. A fixed threaded pool allows you to configure the pool size and concurrently runs, in one of the available threads, the tasks you throw at it. If there are more tasks than threads, the tasks are queued for execution and each queued task is run as soon as a thread is available. A cached threaded pool will create threads as needed and will reuse existing threads if possible. If no activity is scheduled on a thread for well over a minute, it will start shutting down the inactive threads.

The fixed threaded pool fits the bill well for the pool of threads we need in the net asset value application. Based on the number of cores and the presumed blocking coefficient, we decide the thread pool size. The threads in this pool will execute the tasks that belong to each part. In the sample run, we had 40 stocks; if we create 20 threads (for a 2 core processor), then half the parts get scheduled right away. The other half are enqueued and run as soon as threads become available. This will take little effort on our part, let's write the code to get this stock price concurrently.

Download divideAndConquer/ConcurrentNAV.java

```java
public class ConcurrentNAV extends AbstractNAV {
  public double computeNetAssetValue(final Map<String, Integer> stocks)
      throws InterruptedException, ExecutionException {
    final int numberOfCores = Runtime.getRuntime().availableProcessors();
    final double blockingCoefficient = 0.9;
    final int poolSize = (int)(numberOfCores / (1 - blockingCoefficient));

    System.out.println("Number of Cores available is " + numberOfCores);
    System.out.println("Pool size is " + poolSize);
    final List<Callable<Double>> partitions = new ArrayList<Callable<Double>>();
    for(final String ticker : stocks.keySet()) {
      partitions.add(new Callable<Double>() {
        public Double call() throws Exception {
          return stocks.get(ticker) * YahooFinance.getPrice(ticker);
        }
      });
    }

    final ExecutorService executorPool = Executors.newFixedThreadPool(poolSize);
    final List<Future<Double>> valueOfStocks =
      executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);

    double netAssetValue = 0.0;
    for(final Future<Double> valueOfAStock : valueOfStocks)
      netAssetValue += valueOfAStock.get();

    executorPool.shutdown();
    return netAssetValue;
  }

  public static void main(final String[] args)
    throws ExecutionException, InterruptedException, IOException {
    new ConcurrentNAV().timeAndComputeValue();
  }
}
```

In the computeNetAssetValue() method we determine the thread pool size based on the number of cores (Runtime's availableProcessor() method gives you that detail) and presumed blocking coefficient. We then place each part—to fetch the price for each ticker symbol—into the anonymous code block of the Callable interface. This interface provides a call() method that returns a value of the parametrized type of this interface (Double in the example). We then schedule these parts on the fixed size pool using the invokeAll() method. The executor takes the responsibility of concurrently running as many of the parts as possible. If there are more divisions than the pool size, they get queued for their execution turn. Since the parts run concurrently and asynchronously, the dispatching main thread can't get the results right away. So, the invokeAll() method returns a collection of Future objects. We can check with the future object if a result is available, request for cancellation of a task, or, as in this example, request for the result. The call to get() on the Future blocks until the result is available. As the results from the tasks arrive we add the partial results to the net asset value. Let's see how the concurrent version performed.

```
Number of Cores available is 2
Pool size is 20
Your net asset value is $13,661,010.17
Time (seconds) taken 0.967484
```

In contrast to the sequential run, the concurrent run took less than a second. You can vary the number of threads in the pool by varying the presumed blocking coefficient and see if the speed varies. You can also try different number of stocks as well and see the result and speed change between the sequential and the concurrent version.

Isolated Mutability

In this problem the executor service pretty much eliminated any synchronization concerns—it allowed us to nicely delegate tasks and receive their results from a coordinating thread. The only mutable variable we have in the previous code is netAssetValue, which we defined on line 23. The only place where we mutate this variable is on line 25. This mutation happens only in one thread, the main thread—so we have only isolated mutability here and not shared mutability. Since there is no shared state, there is nothing to synchronize in this example. With the help of Future we were able to safely send the result from the threads fetching the data to the main thread.

There's one limitation to the approach in this example. We're iterating through the Future objects in the loop on line 24. So we request results from one part at a time, pretty much in the order you created/scheduled the divisions. Even if one of the later parts has returned first, we'll not process its results until we process the results of parts before that. In this particular example that may not be an issue. However, if we have quite a bit of computation to perform upon receiving the response, then we'd rather process results as they become available instead of waiting for response in sequence. The Future does not, unfortunately, help us with that. We could use the JDK CompletionService for this. We'll revisit this concern and look at some alternate solutions later.

2.3 Speedup for the IO Intensive App

The nature of IO intensive applications allows for a greater degree of concurrency even when you have fewer cores. When you're blocked on a IO operation you can switch to perform other tasks, or request for other IO operations to be started. We estimated that on a two core machine, about 20 threads would be reasonable for the stock total asset value application. Let's analyze the performance on a two core processor for various number of threads—from 1 to 40. Since the total number of divisions is 40, it would not make any sense to create more threads than that. You can observe the speedup as the number of threads is increased in Figure 2.1, on the following page.

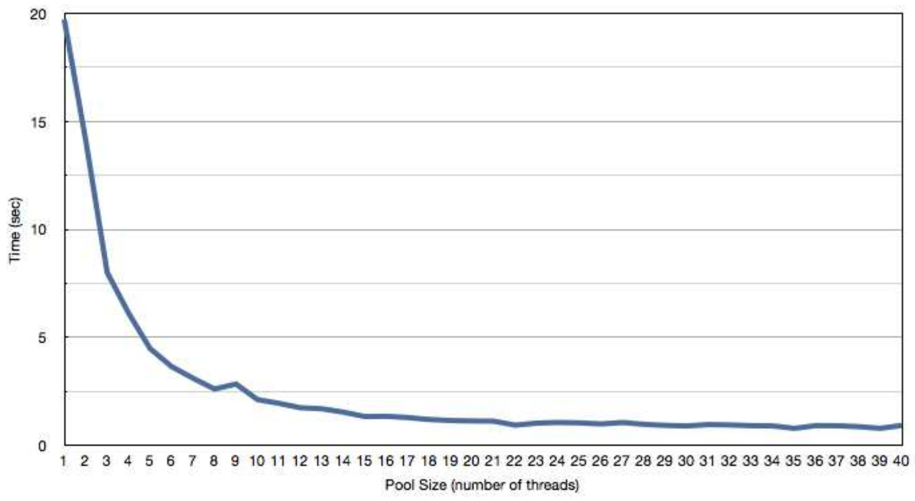


Figure 2.1: Speedup as the pool size is increased

You can see the curve begins to flatten right about 20 threads in the pool. This tells us that our estimate was decent and that having more threads beyond our estimate will not help.

This application is a perfect candidate for concurrency—the workload across the parts is about the same and large blocking due to data requests from the web lend itself really well to exploit the threads. We were able to gain a greater speedup by increasing the number of threads. Not all problems, however, will lend themselves to speedup that way as you'll see next.

2.4 Concurrency in Computationally Intensive Apps

The number of cores has a greater influence on the speedup of computation intensive applications than on IO bound applications as you'll see in this section. The example we'll use is very simple, however, it has a hidden surprise—the uneven workload will affect the speedup.

Let's write a program to compute the number of primes between 1 and ten million. Let's first solve this sequentially and then we'll solve it concurrently.

Sequential Computation of Prime Numbers

Let's start with an abstract class AbstractPrimeFinder that will help group some common methods for the problem at hand. The isPrime() method will tell if a given number is prime and the countPrimesInRange() method uses it to count the number of primes in a range of numbers. Finally, the timeAndCompute() method will keep a tab of the time it took to count the primes.

Download divideAndConquer/AbstractPrimeFinder.java

```java
public abstract class AbstractPrimeFinder {
  public boolean isPrime(final int number) {
    if (number <= 1) return false;

    for(int i = 2; i <= Math.sqrt(number); i++)
      if (number % i == 0) return false;

    return true;
  }

  public int countPrimesInRange(final int lower, final int upper) {
    int total = 0;

    for(int i = lower; i <= upper; i++)
      if (isPrime(i)) total++;

    return total;
  }

  public void timeAndCompute(final int number) {
    final long start = System.nanoTime();

    final long numberOfPrimes = countPrimes(number);

    final long end = System.nanoTime();

    System.out.printf("Number of primes under %d is %d\n", number, numberOfPrimes);
    System.out.println("Time (seconds) taken is " + (end - start)/1.0e9);
  }

  public abstract int countPrimes(final int number);
}
```

Now it's time to invoke the code sequentially and that's an easy task as you see next. For a sequential count of primes between 1 and a given number, simply call countPrimesInRange() with the parameters 1 and number.

Download divideAndConquer/SequentialPrimeFinder.java

```
public class SequentialPrimeFinder extends AbstractPrimeFinder {
  public int countPrimes(final int number) {
    return countPrimesInRange(1, number);
  }

  public static void main(final String[] args) {
    new SequentialPrimeFinder().timeAndCompute(Integer.parseInt(args[0]));
  }
}
```

Let's exercise the code with 10000000 (ten million) as argument and observe the time taken for this sequential run.

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 6.544368
```

On my dual core processor, it took this sequential code, run in server mode, a little more than 6 seconds to count the number of primes under ten million. It would take a lot longer if we run it in client mode. Let's see what speedup we can achieve by turning this to run in concurrent mode.

Concurrent Computation of Prime Numbers

Since this is a computationally intensive task, throwing a lot of threads on the problem will not help. The blocking coefficient is 0 and so the suggested number of threads, from the formula in Section 2.1, *Decide the Number of Threads*, on page 30, is equal to the number of cores. Having more threads than that will not help speedup; on the contrary, it may actually slow down. This is because it makes little sense for the non-blocking task to be paused for another thread to run part of our own problem. We might as well finish the task before taking up the next one when all the cores are busy. Let's first write the concurrent code, give it two threads and two divisions and see what happens. We've ducked the question of number of parts right now, we'll revisit it after we make some progress.

The concurrent version for primes counting is structurally similar to the concurrent version of the net asset value application. However, instead of computing the number of threads and parts, here we'll take those two as parameters, in addition to the candidate number for primes count. Here's the code for the concurrent primes count.

Download divideAndConquer/ConcurrentPrimeFinder.java

```
public class ConcurrentPrimeFinder extends AbstractPrimeFinder {
  private final int poolSize;
  private final int numberOfPartitions;

  public ConcurrentPrimeFinder(final int thePoolSize,
    final int theNumberOfPartitions) {
    poolSize = thePoolSize;
    numberOfPartitions = theNumberOfPartitions;
  }

  public int countPrimes(final int number) {
    int count = 0;
    try {
      final List<Callable<Integer>> partitions =
        new ArrayList<Callable<Integer>>();
      final int chunksPerPartition = number / numberOfPartitions;
      for(int i = 0; i < numberOfPartitions; i++) {
        final int lower = (i * chunksPerPartition) + 1;
        final int upper =
          (i == numberOfPartitions - 1) ? number : lower + chunksPerPartition - 1;

        partitions.add(new Callable<Integer>() {
          public Integer call() {
            return countPrimesInRange(lower, upper);
          }
        });
      }

      final ExecutorService executorPool = Executors.newFixedThreadPool(poolSize);
      final List<Future<Integer>> resultFromPartitions = executorPool.invokeAll(
        partitions, 10000, TimeUnit.SECONDS);
      executorPool.shutdown();

      for(final Future<Integer> result : resultFromPartitions)
        count += result.get();
    } catch(Exception ex) { throw new RuntimeException(ex); }

    return count;
  }

  public static void main(final String[] args) {
    if (args.length < 3)
      System.out.println("Usage: number poolsize numberOfPartitions");
    else
      new ConcurrentPrimeFinder(
          Integer.parseInt(args[1]), Integer.parseInt(args[2]))
        .timeAndCompute(Integer.parseInt(args[0]));
  }
}
```

We divided the range from 1 to the given number into the requested number of parts, with each division containing about the same number of values (lines 14 to 27). We then delegated the job of counting the number of primes in each part to individual instances of Callable. Now that we have the parts on hand, it's time to schedule their execution. We'll use a thread pool for that and submit the execution of these parts to the executor service (lines 29 to 32). The final step, line 35, is to total the results received from the parts.

We're ready to take this for a ride. I have two cores, so if I create two threads and two parts, I expect to get about twice the speedup (should take about half the time as the sequential run). Let's see how that goes by running the program for ten million numbers, with both the pool size and the number of partitions being 2, using the command java ConcurrentPrimeFinder 10000000 2 2.

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 4.236507
```

The number of primes in our sequential and concurrent runs are the same, that's good, the same amount of work was done in the two versions. However, I expected the runtime to be around 3 seconds with two cores, but we came a little over 4 seconds—a rather 1.5 times speedup instead of the 2 times speedup. Time to go back to the drawing board to figure out what we missed.

You probably have a hunch that this has to do with the number of divisions—certainly increasing the number of threads without increasing the number of divisions will not help. However, since this is computationally intensive, we already know that increasing the threads to more than number of cores will not help. So, it's gotta be the number of parts. Let's try to justify that and arrive at a solution.

Let's bring up the activity monitor (on Windows bring up your Task Manager and on Linux bring up your system/resource monitor) to see the activities of the cores. Run the prime finder in sequential mode and then in concurrent mode. You will see results similar to the one in Figure 2.2, on the following page.

The cores are underutilized during the sequential run as you'd expect. During the concurrent run, however, for most part, both the cores are quite active. However, right about 60% into it only one core is active. This tells us that workload across the two threads/cores is not evenly distributed—the first half finished much faster than the second. If we think about the nature of the given problem, it now becomes obvious—the effort to check a larger prime in the second part is more than the

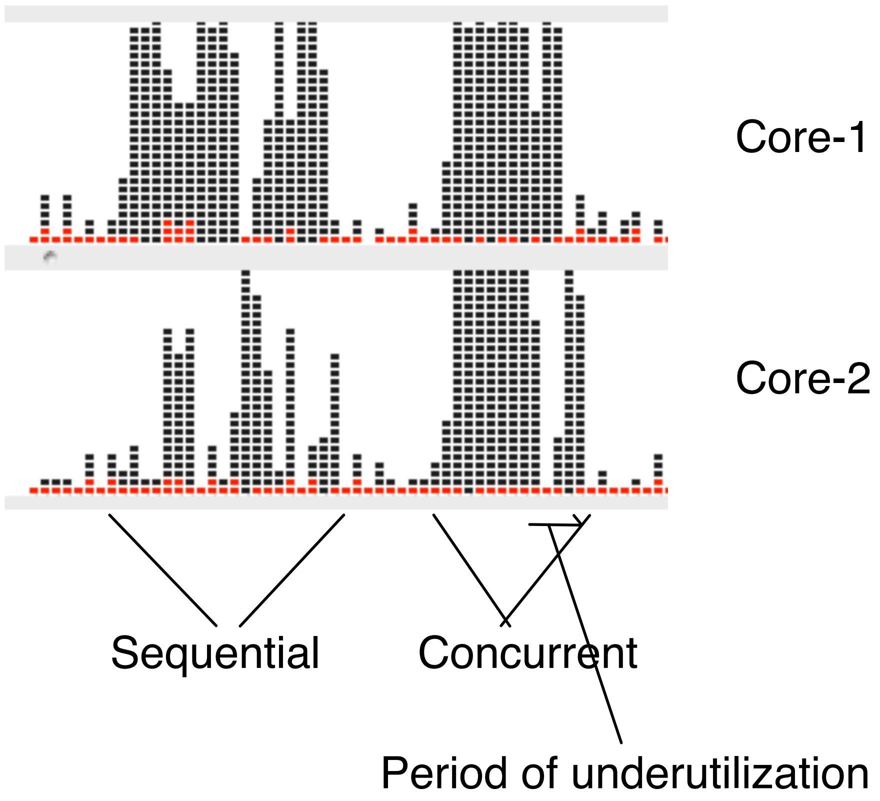


Figure 2.2: Activities of cores during sequential and concurrent primes counting

effort to check a smaller prime in the first part. In order to achieve the maximum speedup, you need to partition the problem across the two threads evenly.

In general, a fair distribution of workload among the parts may not be easy to achieve; it requires fairly good understanding of the problem and its behavior across the range of input. For the count primes problem, the approach we took was to split the range into equal parts in serial order. Instead, we can try to group them into combinations of big and small numbers to get a uniform distribution. For instance, to divide into two parts, we may consider taking the first and last quarter in the range into one part and the middle two quarters into the second.

The code we currently have does not lend itself to that kind of partitioning. If we change the code to do that, we will notice that the runtime reduces. The split, however, gets harder as we consider greater number of parts—with more cores, we'd want to have more parts. Fortunately there is a simpler solution.

The main problem with fewer parts, like two, is that one core does more work while the other cores twiddle their digit. The finer we divide the problem, more likely there will be enough slice to keep all the cores busy. Start with more parts than threads, then as threads complete smaller parts, they will pick up other parts to execute. Some cores can work on long running tasks while other cores can pick up several short running tasks. For the example, let's see how the pool size and partitions affect performance, we'll keep the pool size to 2, but vary the number of parts to 100.

```
Number of primes under 10000000 is 664579
Time (seconds) taken is 3.550659
```

That's about 1.85 times speedup, not quite the desired 2 times, but close. You could try to vary the number of parts or the partitioning strategy. However, with fairly ad-hoc partitioning, we're able to get decent speed increase. The ad-hoc partitioning may not work all the time. The key is that you should ensure the cores have fairly uniform workload for good utilization.

2.5 Speedup for the Computationally Intensive App

The counting primes examples was trivial, but it gave us some insights. It showed that for computationally intensive operations, a fair balance of workload among parts is critical.

We had to choose the partition size and the pool size. Let's see what effect these two factors have on the performance. I ran the program in client mode (because it takes longer to run than server mode and that makes it easier to compare the results) on a 8-core processor. The measure of performance is shown in Figure 2.3, on the next page.

There are a few lessons we can learn from the figure about making computationally intensive applications concurrent:

- You gotta have at least as many partitions as the number of cores—you see the performance was poor for less than 8 parts.

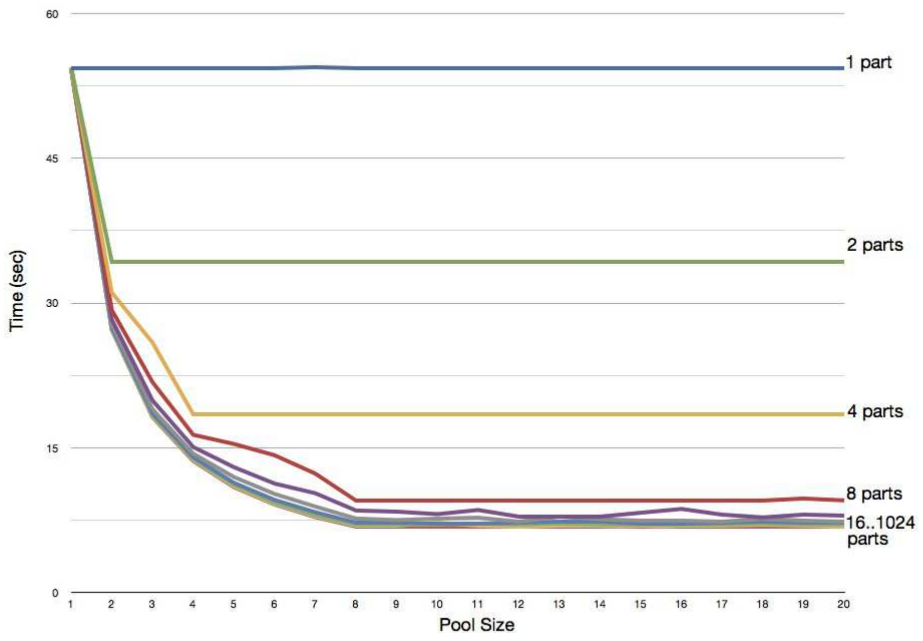


Figure 2.3: Primes Calculation in Client mode on an 8-core processor: Effect of pool size and parts

- Having more threads than the number of cores does not help—you see that for each partitioning, the curve flattens past the pool size of 8.
- Having more partitions is better than having fewer, as more partitions help keep the cores busy—this is quite evident from the curves.
- After a certain number of large partitions, having more partitions has little benefit—for example, I observed that, with 8 cores and a pool size of 8, the time taken for 16, 64, 256, 512, and 1024 partitions were 8.4s, 7.2s, 6.9s, 6.8s, and 6.8s, respectively.

Keep an eye on the effort to divide the problem, if that takes significant computation, that's time wasted from real computations. The key is to have reasonable parts so all the cores have enough work to do. Devise a simple method to divide the problem and see if that provides a fair utilization of all the cores.

2.6 Strategies for Effective Concurrency

On the one hand you want to ensure consistency and accuracy when using concurrency. On the other hand, you want to ensure you're getting as much performance as possible on the given hardware. In this chapter we looked at ways to meet both these goals.

You can easily avoid race conditions or consistency issues once you fully eliminating shared mutable state. When threads don't compete to access mutable data, there's no issue of visibility and crossing memory barrier. You also don't have to worry about controlling the execution sequence of threads, since they don't compete, there are no mutually exclusive sections to safeguard in code.

Provide shared immutability where you can. Otherwise, follow isolated mutability—ensure only one thread ever can access that mutable variable. We're not talking about synchronizing shared state here. We're ensuring that only one thread ever has access to that mutable variable, period.

How many threads you create and how you partition the problem affects the performance of your concurrent application.

First, in order to benefit from concurrency you must be able to partition your problem into smaller tasks that can be run concurrently. If your problem has a significant part that can't be partitioned, then your application may not really see much performance gain by making it concurrent.

If your tasks are IO intensive or have significant IO operations, then having more threads will help. In this case, the number of threads should be much greater than the number of cores. You can estimate the number of threads using the formula presented in Section 2.1, *Decide the Number of Threads*, on page 30.

For a computationally intensive process, having more threads than cores may actually hurt—see Section 2.4, *Concurrent Computation of Prime Numbers*, on page 42. However, you'll benefit by having at least as many threads as the number of cores, assuming the problem can be partitioned into at least as many tasks as the number of threads.

While the number of threads affects performance, it is not the only thing. The workload of each part and how much time each part takes to complete relative to others also affects performance. A uniform partitioning of your problem may take too much effort and may not yield better results than ad-hoc partitioning. Weigh the efforts vs. the benefit. Try to see if you can arrive at a fair workload by using a simple approach to partitioning. In any case, good partitioning requires understanding the nature of the problem and its behavior for different input. The difficulty of this depends on the complexity of the problem on hand.

2.7 Recap

To reap the benefits of multicore processors:

- You have to divide your application into multiple tasks that can be run concurrently.
- Choose at least as many threads as the number of cores, provided the problem is large enough to benefit from those many threads.
- For computational intensive applications, you should limit the number of threads to the number of cores.
- For IO intense applications, the time spent blocking influences the number of threads you'd create.
- Estimate the number of threads using the formula

  ```
  Number of threads = Number of Available Cores / (1 - Blocking Coefficient)
  ```

  where $0 \leq$ blocking coefficient < 1
- Slice the problem into several parts so there is enough work for cores and they're utilized well.
- Avoid shared mutable state and, instead use either isolated mutability and shared immutability.
- Make good use of the modern threading API and thread pools.

In the middle of difficulties lies opportunity.

► Albert Einstein

Chapter 3 Design Approaches

We can't avoid manipulating state, that's integral to programs we create. Your compiler, for instance, takes a set of source files and creates bytecode. Your mail app keeps tab of unread emails, among other things.

As you gain experience with Java, you're probably set in the way you design programs, like your humble author. Your objects encapsulate state and their methods help transition between select valid states. I scoffed when someone said we can create significant code that doesn't change state. Later when he showed how, I was intrigued and enlightened, like that kid in the bookstore.[1]

Manipulating state doesn't necessarily mean mutating state. Think of state transformation instead of state modification. It's a way to design state change without modifying anything. If you wonder how that's possible—to create programs that don't change anything in memory—you'll find the answers in this chapter. Don't be tricked by the length of this chapter—it's short, but captures some fundamental approaches to design.

3.1 Dealing with State

There's no escape from dealing with state, but there are three ways to do so—shared mutability, isolated mutability, and pure immutability.

One extreme is shared mutability—we create variables and allow any thread to modify it, in a controlled fashion of course. Programming with

1. My kids taught me it's more fun hanging out at the bookstore than at the candy store.

shared mutability is simply the way of life for most of us Java programmers. However, this leads to the undesirable synchronize and suffer model. You have to ensure that your code crosses the memory barrier at appropriate time, that you have good visibility on the variables. With shared mutability, you must also ensure that no two threads modify a field at the same time, and that changes to multiple fields are consistent. You get no support from the compiler or the runtime to determine correctness; you've to analyze the code to ensure you did the right thing. The minute you touch the code, you have to reanalyze for correctness, as synchronization is too easy to get wrong. Fortunately we have other options.

In the middle is isolated mutability where variables are mutable, but are never seen by more than one thread, ever. You ensure that anything that's shared between threads are immutable. Java programmers will find this fairly easy to design, and so, the isolated mutability may be a reasonable approach.

Pure immutability is the other extreme where nothing is allowed to change. Designing for this is not easy, partly due to the nature of the problems, but mostly due to our inexperience with this approach. The language also makes this hard, it requires a lot of effort and discipline to program with immutability in Java. It's a paradigm shift and requires exploring some different data structures and ways than we're used to in Java. However, if you can realize such a practical design, the result is rewarding—easy and safe concurrency.

How you design your application in one of these ways depends on the problem and your team's willingness to explore design options. Pure immutability is ideal. However, that's easier said than done, especially for programmers who've spent years mutating shared state. At a minimum you should aim for isolated mutability and avoid the purely evil shared mutability.

In this chapter you'll learn how to use all these approaches to solve a single problem. We'll explore design ways that you can use on your own problems.

3.2 Exploring Design Options

Dealing with state is an activity that takes practice in the art of programming. What you do with the input you receive, the results of your calculations, and the files you change, all involve state. You can't avoid state, but you have a choice of how to deal with it. Here we'll first pick an example and in the next few sections explore different options to dealing with its state.

At a recent Java user group meeting, the conversation rolled over to the topic of collective experience among the members. I saw the young and the mature in the gathering and volunteered to total the number of years of experience. I invite you to help me out with this activity.

At first thought, the net years is a variable that'll change as we total each person's work years—can't avoid mutability, it appears. There are several folks in the room and we need to quickly devise a way to get the total. Let's discuss the pros and cons of the three options for dealing with state.

3.3 Shared Mutable Design

Our first approach to total work years is a familiar approach, write `0` on the board and ask everyone to walkup to add their work years to that total.

Fred who's closest to the board jumps up to add his years to the total, and before we know a number of people have lined up, creating contention around the board. As soon as Fred's done, Jane gets her turn. We'll have to keep an eye to ensure pranksters don't change the entry to something impossible like infinity.

Shared mutability is at work here, along with the challenges it brings. We have to police more than one person trying to change the number at a given time. Furthermore, people have to patiently wait for their turn to write on the board as each person takes their turn. And I hope not to be that last person in the line...

In programming terms, if each person in the room were a separate thread, we'll have to synchronize their access to the shared mutable variable `total`. One badly behaved participant ruins it for the whole group. Also this approach involves quite a bit of blocking of threads, high thread-safety, but low concurrency.

For a large group, this task could become time consuming and frustrating. We want to ease that pain, so let's explore another way.

3.4 Isolated Mutable Design

You walk up to the board, but, instead of putting down the initial total of 0, you put your phone number so everyone in the room can text their work years to you.

Each person now sends you the information from the comfort of their seats. They don't have to wait in line and they're done as fast as they can punch those smartphone keys.

You receive the years in sequence as they arrive, but the senders are concurrent and non-blocking.

Isolated mutability is at work here. The total is isolated, only you hold this value and others can't access it. By isolating the mutable state, you've eliminated the problems of the previous approach. No worries of two or more people changing the stuff at the same time.

You continue to total the years as you receive them. Your carrier took care of turning the concurrent messages into a nice sequential chain of messages for you to process. You can announce the total at the end of the meeting or as soon as you think you're done.

In programming terms, if each person in the room, including you, were a thread (actually an *actor* as we'll see in Chapter 8, *Favoring Isolated Mutability*, on page 172), each of them simply sends an asynchronous message to you. The message they sent itself is immutable—much like the text message, the data you receive is a copy on your side and there should be no way to change anything that affects the sender's view. You're simply modifying a local well encapsulated variable. Since you're the only one (thread) changing this data, there's no need to synchronize around that change.

If you ensure that the shared data is immutable and the mutable data is isolated, you have no concerns of synchronization in this approach.

3.5 Purely Immutable Design

Dealing with full immutability takes a bit of getting used to. Changing things feels natural, and it's easy because we're used to that way of programming in Java. You may wonder how you can find the total of the work years without changing anything—it's possible.

Ask everyone in the room to form a chain, while remaining in their seats. Instruct everyone to receive a number from the person on their left, add their own work years to that number, and pass the combined total to the person to their right. You then provide your own work years to the first person in the room who's eagerly waiting on you.

In this approach no one ever changes anything. Each person holds the total of work years up to them in the chain. The total you receive from the last person in the chain is the total work years of the entire group.

You were able to compute the total without changing a thing. Each person took a partial total and created a new total. The old total value they received is something they could retain or discard (garbage collect) as they wish.

This last approach you saw is common in functional programming and is achieved using function composition. You'd use methods like `foldLeft()`, `reduce()`, or `inject()` in languages like Scala, Clojure, Groovy, and JRuby to implement such operations.

This approach also does not have any concerns of concurrency. It took a bit of effort to organize or compose these sequence of operations. However, you were able to achieve the result with total immutability. If the number of people is very large, you can even partition them into smaller groups—forming a tree instead of a line—to allow greater concurrency and speed.

3.6 Persistent/Immutable Data Structures

In the pure immutable design for the net work-years example we asked each person to create a new total. As we go through the chain of people, a new number is created while an old number is discarded—not a big deal since numbers are relatively small.

The data we have to dealing with may not be that small—it could be a list, tree, or matrix. We may need to work with objects that represent a boiler, a satellite, a city, etc. We won't naively copy these large objects over and over as that'll lead to poor performance. Immutable or persistent data structures come to the rescue here.

Persistent[2] data structures version their values so older and newer values stay around or persist over time without degrading performance. Since the data are immutable, they're shared effectively to avoid copy

2. The word persistent here does not have anything to do with storage, it's about data being persisted or preserved unchanged over time.

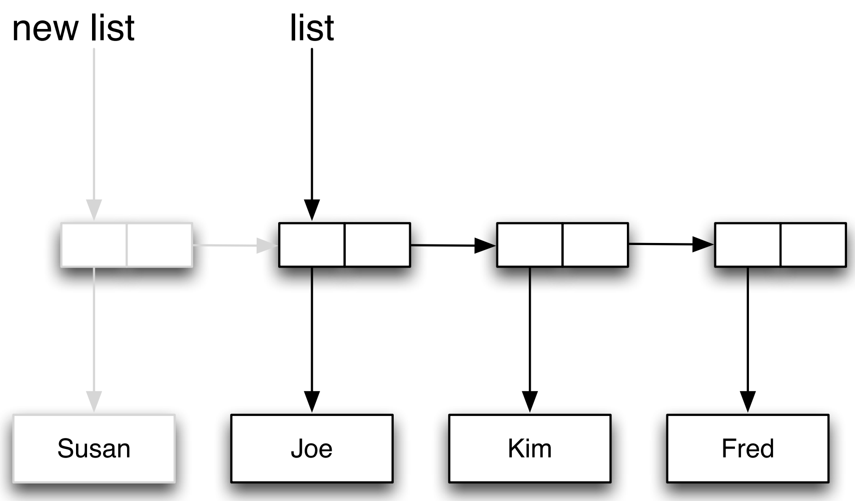


Figure 3.1: Persistent List Processing

overhead. Persistent data structures are designed to provide super efficient “updates.” Functional languages like Clojure and Scala make extensive use of these kinds of data structures.

Immutable Lists

Back to the user group meeting, I’ve been asked to collect the names of the people in attendance. We may use a single linked list where each node has two pointers or references. The first reference is to the person and the second reference to the next node in the list as in the example in Figure 3.1. I’ll show you how we can make this list immutable.

It’s time to change the list as Susan just joined the meeting. We can create a node that holds a reference to her. Since the current list is immutable we can’t add this new node to the end of the list. That operation would require changing the second reference of the last node in the list, a task that an immutable node won’t allow.

Instead of adding this new node to the end, we can add it to the beginning. The second reference in the new node we created can point to the head of the current list. Rather than modifying the existing list, we just got a new list. However, the new list shares all but the first node with the existing list, as shown in Figure 3.1, on the preceding page. In this approach, the addition takes only constant time irrespective of the list size.

Removals from the head of the list also take constant time. The new list contains one element fewer and holds a pointer to the second node from the original list.

Since adds and removes from the head of the immutable list are constant time, you will benefit from them if you can design your problems and algorithms to effectively operate on the head of lists, instead of the tail or middle.

Persistent Tries

Immutable lists give great performance and preserve immutability, but you can't organize you operations around the head of lists all the time. Also, often data is not a simple list, but more complex, like a tree or a hash-map, which can be implemented using trees. You can't get away simply changing the root of a tree, you'll have to change other elements to support insert or remove operations.

If you can flatten the tree, then a change to the tree simply becomes a change to one of its short branches. That's exactly what Phil Bagwell did by using a high branching factor, at least 32 children per node, to create what he called *Tries*—see Ideal Hash Trees in Appendix A, on page 256. The high branching factor reduces the time for operations on tries as we'll soon discuss.

In addition to the high branching factor of 32 or more, tries use a special property to organize the keys. The key for a node is determined by its path, in other words, we don't store the key in the nodes, their location is their key. For example, let's use numbers for path and keep the branching factor to 3, a rather small number compared to the recommended 32 or more, but this makes it easy to discuss. We'll use this tries to store the list of people Joe, Jill, Bob, Sara, Brad, Kate, Paul, Jake, Fred, and Bill, in that order, attending the department meeting as in Figure 3.2, on the following page.

Since the branching factor we used in this example is 3, the paths in base 3 represents the index of each node. For example, the path for Bill is 100 which, in base 3, represents the index value of 9. Similarly, Brad is at path 11 and so has a index value of 4.

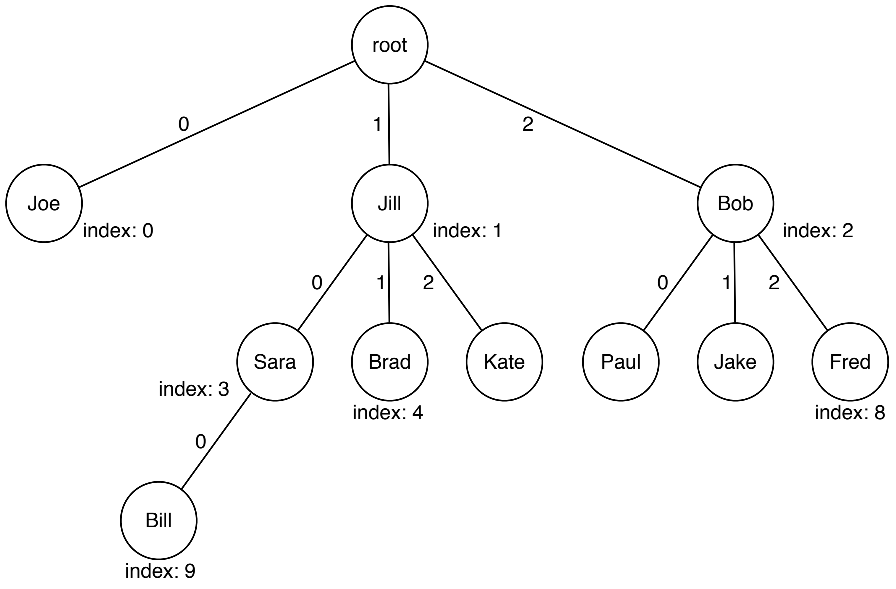


Figure 3.2: Using tries to store a list of people

While Bagwell's tries were not immutable, Rich Hickey used a variation of those to create a persistent hash implementation in Clojure (see *Programming Clojure* [Hal09] by Stuart Halloway). A trie with a branching factor of 32 requires at most 4 levels to hold up to a million elements. An operation to change any element requires only copying up to 4 elements—close to constant time operation.

You can use tries to implement different data structures like trees, hashmap, even lists. The immutable list you saw in Figure 3.1, on page 55 allowed us to add and remove elements only at the head. Tries can remove that restriction and allow us to "add" and "remove" elements at any index in an immutable list.

For example, let's add a new member Sam to the end of our meeting list we implemented using tries in Figure 3.2. Sam's index of 10 translates to path 101 in base 3, so that node should appear as a sibling of the Bill node, as a child of the Sara node. Since all the nodes are immutable, in order to accomplish this task we'd have to copy the Sara, Jill, and root nodes; rest of the nodes are unaffected. The structure of the tries after this selective copying is shown in Figure 3.3, on the next page.

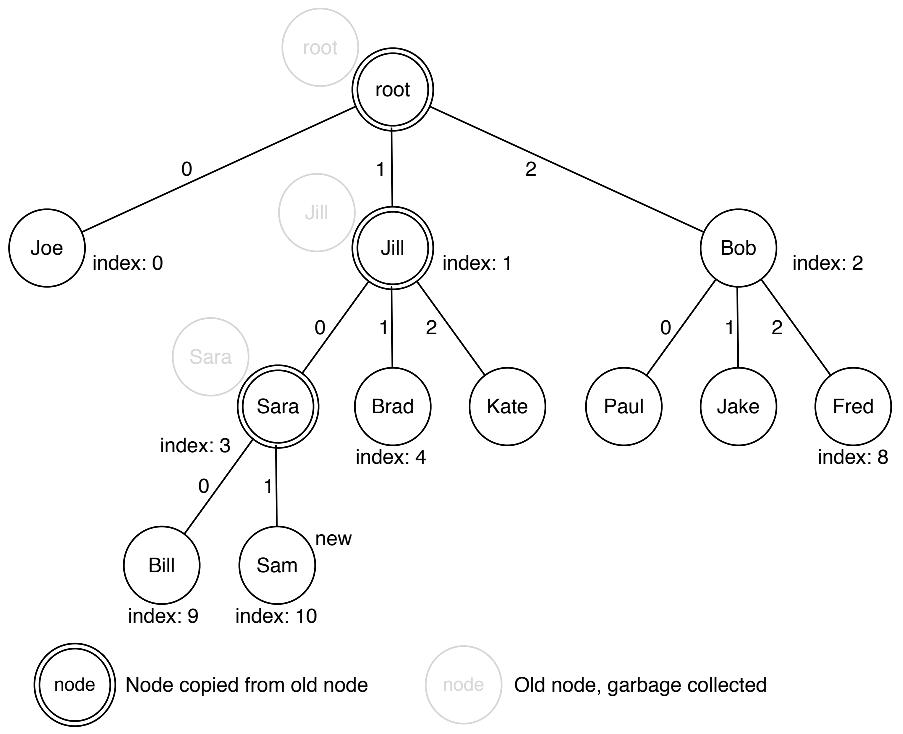


Figure 3.3: "Changing" Persistent List

Rather than having to copy all the nine elements, we only had to copy three nodes, the affected node and its ancestors. The copying itself was a shallow copy, we copied the links to the children nodes, but not the children nodes themselves. As we increase the branching factor to 32 or more, the number of affected nodes for the append operation will still remain close to 4, the number of levels, while the number of elements the list can hold approaches a million. So, appending to a list is a constant time operation for all practical purposes. Inserting elements into arbitrary index position will be slightly more expensive, however, depending on the position we will need some extra partial copying. Unaffected nodes and their decedents are shared intact.

3.7 Selecting a Design Approach

You can avoid most of the issues with concurrency by opting for isolated mutability or pure immutability. For the most part, you may find it easier to program with isolated mutability than with pure immutability.

With isolated mutability, you must ensure that the mutable variable is in fact isolated and never escapes to more than one thread. You also need to ensure that the messages you pass between threads are immutable. You can use either the concurrent API provided with the JDK or one of the actor based concurrency frameworks for message passing.

Designing for pure immutability requires more effort. It's harder to realize it in applications that use object-oriented decomposition more than a functional decomposition. You have to devise better algorithms that favor immutability, apply recursive structure or functional composition, and utilize persistent data structures.

3.8 Recap

We can't avoid dealing with state, but we have three options:

- shared mutability,
- Isolated mutability, and
- pure immutability

Though we're quite used to shared mutability, we should avoid it as much as possible. Eliminating shared mutable state is the easiest way to avoid synchronization woes. Using any of these approaches is far more than using a library or an API; it requires us to step back and think through how we'd design our applications to make use of them.

As part of the design with immutability we'll have to make use of some modern data structures that provide great performance while preserving immutability.

Part II

Modern Java/JDK Concurrency

I don't know, it all happened so fast.

► Turtle mugged by a gang of snails

Chapter 4

Scalability and Thread Safety

Suppose you're working on a program like a disk utility, and one of the tasks is to find the total size of all the files in a directory. The code is working, but you quickly realize it's taking too long when the directories have deep hierarchies. To speed it up, you turn to concurrency for help.

If you started programming in Java in the last century like I did, you've endured the multithreading and collections API that shipped with earlier Java versions. It handles threads and provides thread safety, but doesn't quite consider performance or scalability. Even though threads are relatively lightweight, it takes time and resources to create them. Also, thread safety is provided at the expense of scalability—overly conservative synchronization limits performance.

So, even though it's been around the longest and is still available, the old threading API is not the right choice if you want maintainable code with better scalability and throughput.

It's a new century, and there's a newer concurrency API released with Java 5. It's quite an overhaul of the threading API, and it's better in three main ways:

- It's less painful and more efficient to deal with threads, especially thread pools.
- The new synchronization primitives offer finer granularity—thus more control—than the original primitives.
- The new data structures are scalable—thread safety with reasonable concurrent performance.

In this chapter we'll dive into the Java 5 concurrency API to see these benefits while working on an example.

4.1 Managing Threads with ExecutorService

To find the directory size, you might split the operation into parts, with each task exploring a subdirectory. Since each task has to run in a separate thread, one option is to start an instance of Thread. But threads are not reusable. You can't restart them, and scheduling multiple tasks on them is not easy. You certainly don't want to create as many threads as the number of subdirectories you find—it's not scalable and is the quickest way to fail. The java.util.concurrent API was introduced exactly for this purpose—to manage a pool of threads.

You met the ExecutorService and the different types of executors in Section 2.2, *Concurrent Computation of Net Asset Value*, on page 35. The ExecutorService, the Executors factory, and the related API ease the pain of working with pools of threads. They conveniently separate the types of pools from the operations you perform on them.

Each ExecutorService represents a thread pool. Rather than tying the lifetime of a thread to the task it runs, the ExecutorService separates how you create the thread from what it runs. You can configure the type of thread pool—single threaded, cached, priority based, scheduled/periodic, or fixed size—and the size of the wait queue for tasks scheduled to run. You can easily schedule any number of tasks to run. If you simply want to send off a task to run, wrap the task in a Runnable. For tasks that will return a result, wrap them in Callable.

If you want to schedule arbitrary tasks, use the execute() or submit() methods of ExecutorService. To schedule a collection of tasks use the invokeAll() method. If you only care if one of the tasks completes, as in optimization problems where one of many possible results is adequate, use the invokeAny() method. These methods are overloaded with a timeout parameter, which is the amount of time you're willing to wait for the results.

As soon as you create an executor service, the pool of threads is ready and active to serve. If you have no tasks for them, then the threads in the pool idle away—except in the case of cached pool, where they die after a delay. If you no longer need the pool of threads, invoke the shutdown() method. This method doesn't kill the pool instantly. It completes all tasks currently scheduled before it shuts down the pool, but you can't schedule any new tasks. A call to shutdownNow() tries to force cancel currently executing tasks. There's no guarantee, however, as it relies on the tasks to respond well to the interrupt() call.

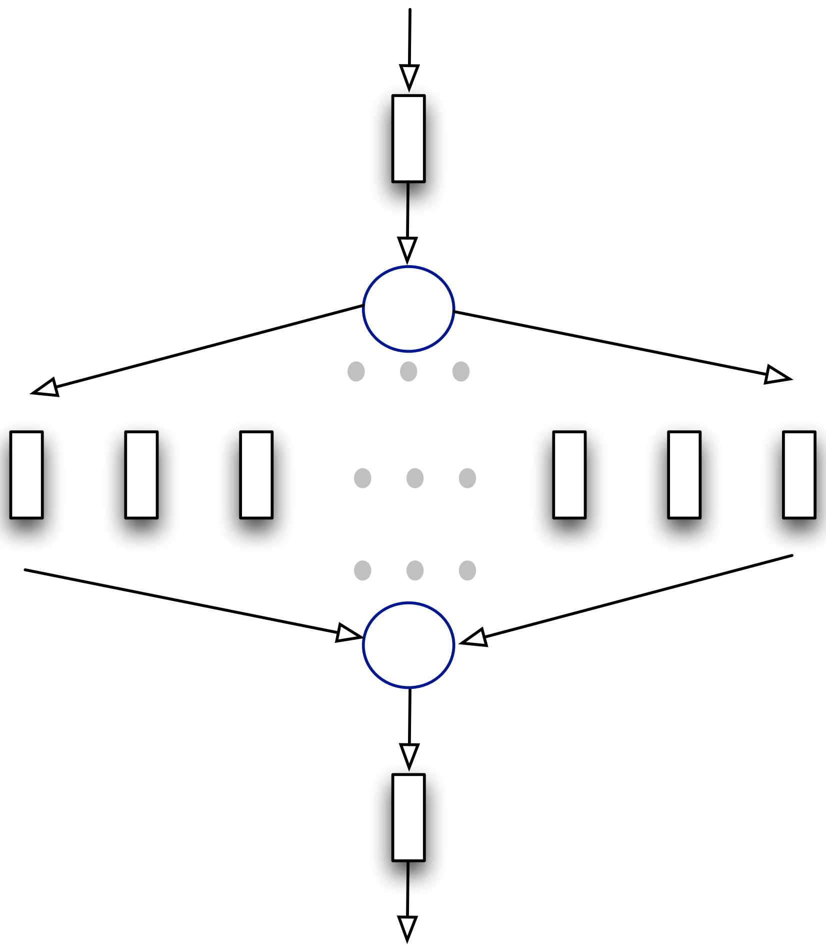


Figure 4.1: Thread Coordination—scheduling and joining

The ExecutorService also provides methods that allow you to check if the service has terminated or shutdown. However, don't rely on them. Design for task completion rather than thread/service death—focus on completion of work (application logic) instead of termination of threads (infrastructure activity).

4.2 Coordinating Threads

Once you divide a problem into parts, you schedule several concurrent tasks in a pool of threads and wait for their results to arrive. When these tasks complete you proceed as in Figure 4.1, on the previous page. You don't want to wait for the threads to die, because these threads may be reused to run several tasks. The result of the scheduled tasks is what you really care about. You can easily achieve this using the Callable interface and the submit() or invokeAll() methods of ExecutorService. Let's look at an example.

To find the total size of all files in a directory hierarchy with potentially thousands of files, we can run several parts concurrently. When parts finish, we need to total up the partial results.

Let's first take a look at the sequential code for totaling the file size.

Download scalabilityAndTreadSafety/coordinating/TotalFileSizeSequential.java

```
public class TotalFileSizeSequential {
  private long getTotalSizeOfFilesInDir(final File file) {
    if (file.isFile()) return file.length();

    final File[] children = file.listFiles();
    long total = 0;
    if (children != null)
      for(final File child : children)
        total += getTotalSizeOfFilesInDir(child);
    return total;
  }

  public static void main(final String[] args) {
    final long start = System.nanoTime();
    final long total = new TotalFileSizeSequential()
      .getTotalSizeOfFilesInDir(new File(args[0]));
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

We start at a given directory and total the size of all the files, recursively drilling down the subdirectories.

When you run this and other file size programs in this book, you will observe it takes quite a long time on the first run and the time drops down for subsequent runs performed within minutes. This is due to the caching of the file system. I have discarded the time from the first run so that all the runs used in comparison have the advantage of cache.

Let's run this on a few directories. We'll use the output to compare the performance of the concurrent version we'll create soon.

```
>java TotalFileSizeSequential /etc
Total Size: 2266456
Time taken: 0.011836

>java TotalFileSizeSequential /usr
Total Size: 3793911517
Time taken: 18.570144
```

We certainly can hope to gain an increase in speed by making this code concurrent. We can divide the problem into tasks where each task accepts a directory and returns its total size. The Callable interface is a good fit for this as its call() method can return a result when done. As we loop through each file in the directory, we can schedule the task using the submit() method of ExecutorService. We can then gather up all the partial results by simply calling the get() method of the Future object, this object acts as a delegate that will provide us the result when available. Let's implement the logic we just discussed in a NaivelyConcurrentTotalFileSize class.

Download scalabilityAndTreadSafety/coordinating/NaivelyConcurrentTotalFileSize.java

```java
public class NaivelyConcurrentTotalFileSize {

  private ExecutorService service;

  private long getTotalSizeOfFilesInDir(final File file)
    throws InterruptedException, ExecutionException, TimeoutException {
    if (file.isFile()) return file.length();

    long total = 0;
    final File[] children = file.listFiles();

    if (children != null) {
      List<Future<Long>> partialTotalFutures = new ArrayList<Future<Long>>();
      for(final File child : children) {
        partialTotalFutures.add(service.submit(new Callable<Long>() {
          public Long call()
            throws InterruptedException, ExecutionException, TimeoutException {
            return getTotalSizeOfFilesInDir(child);
          }
        }));
      }

      for(final Future<Long> partialTotalFuture : partialTotalFutures)
        total += partialTotalFuture.get(100, TimeUnit.SECONDS);
    }

    return total;
  }
```

The code starts to explore a given directory. For each file or subdirectory, we create a task for finding its size and schedule it on the pool on line 15. Once we've scheduled tasks for all files and subdirectories under a directory, we wait for their size to be returned through the Future objects. To compute the total size for a directory, we then iterate over futures on line 23. We don't want to block endlessly for a result to arrive, so there's a time limit for the get() method to respond. In other words, bail out with an error if a task doesn't finish within that time. Each time you recursively call the getTotalSizeOfFilesInDir() methods, you schedule more tasks on the executor service pool, as long as there are more directories and files to visit.

We've not created the pool of threads yet, so let's get to that now.

Download scalabilityAndTreadSafety/coordinating/NaivelyConcurrentTotalFileSize.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException, ExecutionException, TimeoutException {
    service  = Executors.newFixedThreadPool(100);
    try {
     return getTotalSizeOfFilesInDir(new File(fileName));
    } finally {
      service.shutdown();
    }
  }

  public static void main(final String[] args)
    throws InterruptedException, ExecutionException, TimeoutException {
    final long start = System.nanoTime();
    final long total = new NaivelyConcurrentTotalFileSize()
      .getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

We created a pool of size 100. As threads are limited resource, you don't want to create too many of them; see Chapter 2, *Division of Labor*, on page 29 on how to estimate. Once we create the pool we set the exploration of the given directory in motion using the getTotalSizeOfFilesInDir() method. Let's take the code for a drive and see how it performs.

```
>java NaivelyConcurrentTotalFileSize /etc
Total Size: 2266456
Time taken: 0.12506
```

The file size reported here is the same as in the sequential version, but the speed wasn't encouraging. It actually took longer to run. A lot of time was spent in scheduling, instead of doing real work for the rather flat directory hierarchy under /etc. Let's not let our hopes fade yet. Perhaps we'll see an improvement for the /usr directory, which took over 18 seconds in the sequential run. Let's give that a try:

```
>java NaivelyConcurrentTotalFileSize /usr
Exception in thread "main" java.util.concurrent.TimeoutException
        at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:228)
        at java.util.concurrent.FutureTask.get(FutureTask.java:91)
        at NaivelyConcurrentTotalFileSize.getTotalSizeOfFilesInDir(
          NaivelyConcurrentTotalFileSize.java:34)
        at NaivelyConcurrentTotalFileSize.getTotalSizeOfFile(
          NaivelyConcurrentTotalFileSize.java:45)
        at NaivelyConcurrentTotalFileSize.main(
          NaivelyConcurrentTotalFileSize.java:54)
```

Hum..., Not quite what we wanted to see, a timeout problem. Don't panic; there's a reason I called this version *naive*.

The flaw is in the getTotalSizeOfFilesInDir() method, it clogs the thread pool. As this method discovers subdirectories, it schedules the task of exploring them to other threads. Once it schedules all these tasks, this method awaits response from each one of them. If you only had a few directories, then it's no big deal. But if you have a deep hierarchy this method will get stuck. While threads wait for response from the tasks they create, these tasks end up waiting in the ExecutorService's queue for their turn to run. A potential "pool induced deadlock," if you didn't have the timeout. Since we used a timeout, we're able to at least terminate unfavorably rather than wait forever. Making code concurrent is not trivial—it's time to go back to the drawing board to fix this code.

You want to delegate the computation of size for various directories to different threads, but not hold on to the calling thread while you wait for these tasks/threads to respond.[1]

One way to tackle this is for each task to return a list of subdirectories it finds, instead of the full size for a given directory. Then from the main task you can dispatch other tasks to navigate these subdirectories. This will prevent holding threads for any period longer than simply fetching the immediate subdirectories. While the tasks fetch subdirectories, they

1. Later you'll see how the actor based model fits really well to solve this problem.

can also total the size of files in their directories. Let's put that design to work in a ConcurrentTotalFileSize and see if it gives us better results.

As we discover subdirectories and files, we need to pass the list of subdirectories and the total size of files to the main thread. We'd need an immutable object to hold these values, so let's create an inner class SubDirectoriesAndSize to hold that.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
public class ConcurrentTotalFileSize {
  class SubDirectoriesAndSize {
    final public long size;
    final public List<File> subDirectories;
    public SubDirectoriesAndSize(
      final long totalSize, final List<File> theSubDirs) {
      size = totalSize;
      subDirectories = Collections.unmodifiableList(theSubDirs);
    }
  }
```

Next we'll write the method that, given a directory, will return an instance of SubDirectoriesAndSize that holds the subdirectories of the given directory along with size of files it contains.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
private SubDirectoriesAndSize getTotalAndSubDirs(final File file) {
  long total = 0;
  final List<File> subDirectories = new ArrayList<File>();
  if(file.isDirectory()) {
    final File[] children = file.listFiles();
    if (children != null)
      for(final File child : children) {
        if (child.isFile())
          total += child.length();
        else
          subDirectories.add(child);
      }
  }

  return new SubDirectoriesAndSize(total, subDirectories);
}
```

As we discover subdirectories, we want to delegate the task of exploring them to other threads. Most of the efforts for concurrency will go into a new method getTotalSizeOfFilesInDir().

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
private long getTotalSizeOfFilesInDir(final File file)
  throws InterruptedException, ExecutionException, TimeoutException {
```

```
  ExecutorService service = Executors.newFixedThreadPool(100);

  try {
    long total = 0;

    final List<File> directories = new ArrayList<File>();
    directories.add(file);

    while(!directories.isEmpty()) {
      final List<Future<SubDirectoriesAndSize>> partialResults =
        new ArrayList<Future<SubDirectoriesAndSize>>();

      for(final File directory : directories) {
        partialResults.add(service.submit(new Callable<SubDirectoriesAndSize>() {
          public SubDirectoriesAndSize call() {
            return getTotalAndSubDirs(directory);
          }
        }));
      }

      directories.clear();

      for(final Future<SubDirectoriesAndSize> partialResultFuture :
        partialResults) {
        SubDirectoriesAndSize subDirectoriesAndSize =
          partialResultFuture.get(100, TimeUnit.SECONDS);
        directories.addAll(subDirectoriesAndSize.subDirectories);
        total += subDirectoriesAndSize.size;
      }
    }

    return total;
  } finally {
    service.shutdown();
  }
}
```

We create a thread pool of size 100 and add the given top level directory to a list of directories to explore. Then, while there are still files left to explore, we invoke the getTotalAndSubDirs() in separate threads for each directory on hand. As the response from these threads arrive, we then add the partial file size they return to a total and subdirectories to the list of directories to explore. Once all the subdirectories have been explored we return the total file size. The last step is to get this code in motion, we need the main() method for that.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSize.java

```
  public static void main(final String[] args)
    throws InterruptedException, ExecutionException, TimeoutException {
```

```
    final long start = System.nanoTime();
    final long total = new ConcurrentTotalFileSize()
      .getTotalSizeOfFilesInDir(new File(args[0]));
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

This version of concurrent implementation took quite some effort, but when compared to the old NaivelyConcurrentTotalFileSize, the new ConcurrentTotalFileSize is a better design—it does not hold a thread for a long time. It quickly gets a list of subdirectories to you, so you can separately schedule the visits to these. It looks like this should give us the desired result; let's see if that's true.

```
>java ConcurrentTotalFileSize /usr
Total Size: 3793911517
Time taken: 8.220475
```

First, unlike the naive version, this version completed successfully. To find the total size of the /usr directory it took only about 8 seconds compared to over 18 seconds the sequential run took. Time to celebrate? Not so fast.

Let's review what we did in this example. The main idea is to dispatch tasks to threads and then wait for their results only in the main thread. All the other threads are quick, in that they only take time to find the total size of files, and, in addition, return the list of subdirectories in a given directory.

While that design idea is quite simple, implementing it wasn't easy. You had to create a class to hold the immutable results from the tasks. It also took some effort to continuously dispatch tasks and coordinate their results. The end result: better performance, but quite a bit of added complexity. Let's see if we can simplify this further.

Coordination using CountDownLatch

Future served you in two ways in the previous example. First it helped you get the result of the tasks. Implicitly, it also helped you coordinate your thread with these tasks/threads. It allowed you to wait for those results to arrive before your thread proceeded with its work. Future, however, is not helpful as a coordination tool if your tasks have no results to return. You don't want to have an artificial return result simply for the sake of coordination. CountDownLatch comes in handy in situations like this.

The NaivelyConcurrentTotalFileSize code was a lot simpler and shorter than the ConcurrentTotalFileSize code. I prefer simple code that works. The problem with the NaivelyConcurrentTotalFileSize is that each thread waited for the tasks it scheduled to complete. The nice thing about both the versions of code was they had no mutable shared state. If you compromise a little on the shared mutability,[2] you can keep the code simple and make it work as well. Let's see how.

Instead of returning the subdirectories and the file size, we can let each thread update a shared variable. With nothing to return, the code will be a lot simpler. We still have to ensure that the main thread waits for all the subdirectories to be visited. We can use the CountDownLatch for this, to signal the end of wait. The latch works as a synchronization point for one or more threads to wait for other threads to reach a point of completion. Here we simply use the latch as a switch.

Let's create a class ConcurrentTotalFileSizeWLatch that will use the CountDownLatch. We'll recursively delegate the task of exploring subdirectories to different threads. When a thread discovers a file, instead of returning the results, it updates a shared variable totalSize of type AtomicLong. AtomicLong provides thread-safe methods to modify and get the values of a simple long variable. In addition you'll use another AtomicLong variable pendingFileVisits to keep a tab on the number of files still to be visited. When this count goes to zero, you release the latch by calling countDown().

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWLatch.java

```
public class ConcurrentTotalFileSizeWLatch {
  private ExecutorService service;
  final private AtomicLong pendingFileVisits = new AtomicLong();
  final private AtomicLong totalSize = new AtomicLong();
  final private CountDownLatch latch = new CountDownLatch(1);

  private void findTotalSizeOfFilesInDir(final File file) {
    long fileSize = 0;
    if (file.isFile())
      fileSize = file.length();
    else {
      final File[] children = file.listFiles();
      if (children != null) {
```

2. See how easy it is to fall into this trap of using shared mutability.

```
          for(final File child : children) {
            if (child.isFile())
              fileSize += child.length();
            else {
              pendingFileVisits.incrementAndGet();
              service.execute(new Runnable() {
                public void run() { findTotalSizeOfFilesInDir(child); }
              });
            }
          }
        }
      }

      while(true) {
        final long size = totalSize.longValue();
        if (totalSize.compareAndSet(size, size + fileSize)) break;
      }

      if(pendingFileVisits.decrementAndGet() == 0) latch.countDown();
    }
```

The mechanics of exploring directories is done, we now need code to create the pool of threads, set the exploration of directories running, and wait on the latch. When the latch is released by findTotalSizeOfFilesInDir(), the main thread will be released from its await() call and returns the total size it knows.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWLatch.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    pendingFileVisits.incrementAndGet();
    try {
     findTotalSizeOfFilesInDir(new File(fileName));
     latch.await(100, TimeUnit.SECONDS);
     return totalSize.longValue();
    } finally {
      service.shutdown();
    }
  }

  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
    final long total = new ConcurrentTotalFileSizeWLatch()
      .getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

This version has a lot less code, let's run it.

```
>java ConcurrentTotalFileSizeWLatch /usr
Total Size: 3793911517
Time taken: 10.22789
```

The time taken is slightly more than the ConcurrentTotalFileSize version. This is due to the extra synchronization in all the threads—shared mutability requires protection for thread safety, and that lowers concurrency.

In the previous example we used CountDownLatch as a simple switch by setting the latch value to 1. We could also use higher value and let multiple threads wait on it. That would be useful if you want multiple threads to reach a coordination point before continuing to perform some task. A CountDownLatch, however, is not reusable. Once it's used for a synchronization, it must be discarded. If we want a reusable synchronization point, we should use a CyclicBarrier instead.

On line 27 we had to loop to update totalSize. AtomicLong's compareAndSet() is a non-blocking method. If the current value is the same as the one expected, then in a thread-safe manner it changes the value to the new value. If some other thread sneaks in and modifies the value while we're at it, then the change doesn't take effect and the operation returns a false. We can retry to set the value, but we have to remember to refetch the latest value. In the previous example we try until we succeed to update the size.

The performance of the code was better than the sequential version TotalFileSizeSequential. It was a bit worse than the ConcurrentTotalFileSize version, but a lot simpler. However, there's an added risk of accessing a shared mutable variable—something I have cautioned against quite a bit. It would be so much better if you can keep the code simple and still avoid the shared mutability. We will see how to do that later in Chapter 8, *Favoring Isolated Mutability*, on page 172.

4.3 Exchanging Data

You will often want to exchange data between multiple cooperating threads. In the previous examples we used Future and AtomicLong. Future is quite useful when you want to get a response from a task on completion. AtomicLong and the other atomic classes in the java.util.concurrent.atomic package are useful for dealing with single shared data values. While these are useful for exchanging data, they can get unwieldy as you saw in the ConcurrentTotalFileSize example. To work with multiple data values or to exchange data quite frequently, you'll want a better mechanism than either of these provided. The java.util.concurrent API has a number of classes that provide thread-safe ways to communicate arbitrary data between threads.

If you simply want to exchange data between two threads, you can use the Exchanger class. This serves as a synchronization point where two threads can swap data in a thread-safe manner. The faster thread is blocked until the slower thread catches up to the synchronization point to exchange data.

If you want to send a bunch of data between threads, instances of the BlockingQueue interface can come in handy. As the name indicates, inserts are blocked until space is available and removals are blocked until data is available. Quite a few flavors of BlockingQueue are available in the JDK. For example, if you want to match inserts with removals, use a SynchronousQueue. If you want data to bubble up based on some priority, use the PriorityBlockingQueue. For a simple blocking queue, you can select a linked-list flavor from LinkedBlockingQueue or an array flavor from ArrayBlockingQueue.

We can use a blocking queue to solve the total file size problem concurrently. Rather than using the AtomicLong mutable variable, we can have each thread place the partial file size they compute into a queue. The main thread can then take these partial results from the queue and compute the total locally. Let's first write the code to explore the directories.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWQueue.java

```
public class ConcurrentTotalFileSizeWQueue {
  private ExecutorService service;
  final private BlockingQueue<Long> fileSizes = new ArrayBlockingQueue<Long>(500);
  final AtomicLong pendingFileVisits = new AtomicLong();

  private void exploreDir(final File file) {
    long fileSize = 0;
    if (file.isFile())
      fileSize = file.length();
    else {
      final File[] children = file.listFiles();
      if (children != null)
        for(final File child : children) {
          if (child.isFile())
            fileSize += child.length();
          else {
            pendingFileVisits.incrementAndGet();
            service.execute(new Runnable() {
              public void run() { exploreDir(child); }
            });
          }
        }
    }

    try {
      fileSizes.put(fileSize);
    } catch(Exception ex) { throw new RuntimeException(ex); }

    pendingFileVisits.decrementAndGet();
  }
```

We delegate separate tasks to explore the subdirectories. Each task keeps a tab of the total size of files directly under a given directory and in the end puts the total size into the queue using the blocking call put(). Any subdirectories discovered by the tasks are explored in separate tasks/threads.

The job of the main thread is to set the previous code in motion and simply loop through the queue and total the file sizes it receives from the tasks until all subdirectories have been explored.

Download scalabilityAndTreadSafety/coordinating/ConcurrentTotalFileSizeWQueue.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    try {
     pendingFileVisits.incrementAndGet();
     exploreDir(new File(fileName));
     long totalSize = 0;
     while(pendingFileVisits.get() > 0 || fileSizes.size() > 0)
     {
       Long size = fileSizes.poll(10, TimeUnit.SECONDS);
       totalSize += size;
     }
     return totalSize;
    } finally {
      service.shutdown();
    }
  }

  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
    final long total = new ConcurrentTotalFileSizeWQueue()
      .getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

Let's see how this version performs.

```
>java ConcurrentTotalFileSizeWQueue /usr
Total Size: 3793911517
Time taken: 10.293993
```

This version is comparable in performance to the previous one, but the code is a notch simpler—we did not need any active looping to update shared variable, the blocking queue took care of that. The blocking queue helped with exchanging and synchronizing the data between the threads.

You saw how to interact between threads with thread safety. The collections that are part of the java.util.concurrent package give you thread safety and act as a synchronization point. Thread safety is important, but you don't want to compromise performance. Next we'll explore java.util.concurrent data structures geared towards performance.

4.4 Scalable Collections

Collections like Vector that originally came with Java gave you thread safety at the cost of performance. All your accesses, irrespective of your need, were thread safe, but slow.

Later collections like ArrayList provided speed but lacked thread safety. Again, you traded performance for thread safety when you used one of the synchronized wrappers of the collections, like using the synchronizedList() of Collections.

The bottom line is: you had to pick between thread safety and performance. Java 5's java.util.concurrent changed that with concurrent data structures like ConcurrentHashMap and ConcurrentLinkedQueue. These collections provide better performance, at the expense of space, but you have to be willing to accept a little change in semantics.

Synchronized collections provide thread safety while concurrent collections provide both thread safety and better concurrent access performance. Using synchronized collections is like driving through a traffic-light controlled intersection, concurrent collection is like an uncongested freeway.

If you change a synchronized map while iterating over it, you're greeted with a ConcurrentModificationException. In essence you are required to hold an exclusive lock, a rather pessimistic lock, on the map when you iterate over it. This increased thread safety, but clearly lowered concurrency. Consequently, you get low throughput when you use these synchronous collections.

Concurrent collections, on the other hand, were designed to provide throughput. They let multiple operations coexist. You're welcome to modify a map, for example, while you're iterating over it. The API guarantees the integrity of the collection, and you'll never see same element appear twice in your current iteration. It's a compromise in semantics because you must be willing to accept elements being changed and removed while you're iterating over them or getting elements.

Let's explore the behavior of the ConcurrentHashMap in contrast to the older maps.

In the following code we iterate over a map of scores in a separate thread. Right in the middle of the iteration we put a new key into the map.

Download scalabilityAndTreadSafety/concurrentCollections/AccessingMap.java

```java
public class AccessingMap {
  private static void useMap(final Map<String, Integer> scores)
    throws InterruptedException {
    scores.put("Fred", 10);
    scores.put("Sara", 12);

    try {
      for(final String key : scores.keySet()) {
        System.out.println(key + " score " + scores.get(key));
        scores.put("Joe", 14);
      }
    } catch(Exception ex) {
      System.out.println("Failed: " + ex);
    }

    System.out.println("Number of elements in the map: " +  scores.keySet().size());
  }
```

If the map is either a plain vanilla HashMap or its synchronized wrapper, this code will freakout—it will result in a violation we discussed. In reality, before you iterate over a synchronized collection, you'd have to lock it to prevent such violation. If the map is a ConcurrentHashMap, however, it will not suffer the wrath of ConcurrentModificationException. In order to see this difference in behavior, let's exercise the useMap() first with an instance of the good old HashMap(), then with a synchronized wrapper, and finally with an instance of ConcurrentHashMap.

Download scalabilityAndTreadSafety/concurrentCollections/AccessingMap.java

```
  public static void main(final String[] args) throws InterruptedException {
    System.out.println("Using Plain vanilla HashMap");
    useMap(new HashMap<String, Integer>());

    System.out.println("Using Synchronized HashMap");
    useMap(Collections.synchronizedMap(new HashMap<String, Integer>()));

    System.out.println("Using Concurrent HashMap");
    useMap(new ConcurrentHashMap<String, Integer>());
  }
}
```

When you run this example you'll see that the traditional maps could not handle this modification in the middle of iteration (even from within a single thread), but the ConcurrentHashMap took it really well.

```
Using Plain vanilla HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Synchronized HashMap
Sara score 12
Failed: java.util.ConcurrentModificationException
Number of elements in the map: 3
Using Concurrent HashMap
Sara score 12
Fred score 10
Number of elements in the map: 3
```

In addition to allowing interleaved reads and writes, the concurrent collections provide better throughput than the synchronized versions. This is because rather than holding an exclusive lock, the concurrent collections allow multiple updates and reads to work concurrently. Your reads will see the latest values. If your read occurs in the middle of a bulk write, your read is not blocked for the entire update to complete. This means you may see part of the change, but not all. In any case, concurrent collections guarantee visibility or the *happens-before* behavior.

Let's take a look at the performance of concurrent map vs. the synchronized version. In this measure the workload of each thread is about the same, so, as the number of threads is increased, the overall workload in the process is increased as well—giving opportunity for more contention. In each thread I access a random key from the map. If the key

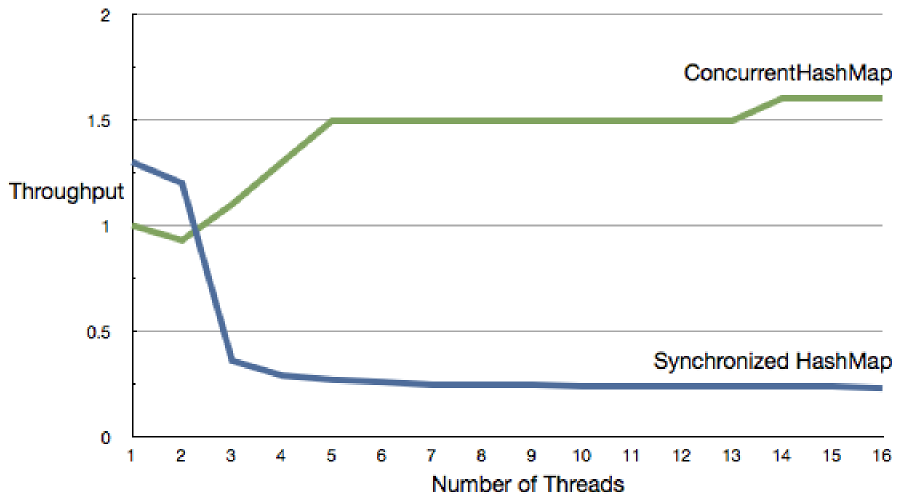


Figure 4.2: Throughput of ConcurrentHashMap vs. Synchronized HashMap on an 8-core Processor

is not found I insert it about 80% of the time and if the key is found I remove it about 20% of the time. I try out the above operations first on a ConcurrentHashMap and then on a HashMap with a synchronized wrapper.

The throughput on a 8-core processor is shown in Figure 4.2. The synchronized HashMap did slightly better than the ConcurrentHashMap when the number of threads was fewer than two, that is the synchronized version did well when there was no concurrency. As the number of threads increased the ConcurrentHashMap clearly outperformed HashMap.

If you're willing to accept the difference in semantics of the concurrent collections, you'll benefit from their better performance compared to synchronized collections.

4.5 Lock vs. synchronized

You've avoided explicit synchronization so far in the examples in this chapter. You can't escape it for long, however, if you're going to restrict yourself to the JDK concurrency API—it'll show up the minute you have to coordinate changes to multiple variables or multiple objects.

In Java, you have two constructs to acquire locks—the archaic **synchronized** and its modern counterpart, the Lock interface. Favor the Lock interface over synchronized. Let's discuss the reasons.

synchronized

You use **synchronized** to gain an explicit monitor/lock on objects. While grabbing the monitor and releasing it at the end of the block, it also helps threads cross the memory barrier. However, synchronized is very primitive in its capabilities and has quite a few limitations.

Unfortunately, synchronized makes it far too easy to run into livelock issues—threads could wait indefinitely for some events, like acquiring locks, to happen. There's no easy way to tell synchronized to wait for a finite time to acquire a lock.

If you attempt unit testing for thread safety, you'll find that synchronized makes it impossible. There is no effective, simple, and deterministic way, like replacing synchronized with a mock, to examine if you're wrapping a mutually exclusive block of code in a synchronized section or block.

Furthermore, synchronized leads to exclusive locks, and no other thread can gain access to the monitors held. This does not favor situations with multiple readers and infrequent writers. Even though the readers could run concurrently, they're serialized and this results in poor concurrent performance.

As you can see, these problems make synchronized rather undesirable for providing thread safety.

The Lock interface

The Java 5 Lock interface fixes the problems of synchronized—see The Lock interface in Appendix A, on page 256.

The implementors of the Lock interface guarantee that calls to their methods cross the memory barrier. You acquire and release locks using the Lock interface's lock() and unlock() methods, like so:

```
aMonitor.lock();
try {
  //...
} finally {
  aMonitor.unlock();
}
```

Perform the unlock() in the **finally** block to ensure proper unlock even in the case of exceptions. While lock() is a blocking call, you can call the non-blocking variation tryLock() to instantly acquire the lock if available. If you'd rather wait for a finite time, provide a timeout parameter to the tryLock() method. There's even a variation that allows you to interrupt a thread while it's waiting for a lock.

Let's see how the Lock interface fixes each of the synchronized issues.

The tryLock() method fixes the livelock concerns. Your lock requests are not forced to block. Instead you can check instantly if you've acquired the lock or not. Also, if you decide to wait for a lock, you can place a limit on the wait time. You may even have your wait interrupted quite easily.

The Lock interface makes it quite easy to unit test for thread safety; for details see "Test Driving Multithreaded Code" in Appendix A, on page 256. You can simply mock out the implementation of the Lock interface and check if the code being tested requests the lock and unlock at the appropriate time.

You can acquire concurrent read locks and exclusive write locks using ReadWriteLock. Thus multiple readers can continue concurrently without having to wait, and are delayed only when a conflicting writer is active.

Let's look at an example of using ReentrantLock which implements the Lock interface. As the name suggests, it allows threads to re-request locks they already own, thus allowing them to reenter their mutually exclusive sections.

Suppose you've been asked to write code to transfer money between two accounts, where an Account class is defined as shown next.

Download scalabilityAndTreadSafety/locking/Account.java

```
public class Account implements Comparable<Account> {
  private int balance;
  final public Lock monitor = new ReentrantLock();

  public Account(final int initialBalance) { balance = initialBalance; }

  public int compareTo(final Account other) {
    return new Integer(hashCode()).compareTo(other.hashCode());
  }

  public void deposit(final int amount) {
    monitor.lock();
    try {
      if (amount > 0) balance += amount;
    } finally { //In case there was an InterruptedException you're covered
      monitor.unlock();
    }
  }

  public boolean withdraw(final int amount) {
    try {
      monitor.lock();
      if(amount > 0 && balance >= amount)
      {
        balance -= amount;
        return true;
      }
      return false;
    } finally {
      monitor.unlock();
    }
  }
}
```

The Account class implements the Comparable interface to help place accounts in their natural order. This serves to avoid deadlocks when working with multiple accounts by locking them in their natural order. Each instance of Account exposes a ReentrantLock, which you can use to gain an exclusive lock on the instance. The deposit() and the withdraw() methods also use this monitor to gain mutual exclusion of their operation by the calling threads.

So, you're all set to write your transfer() method in an AccountService class. You first try to gain locks on the two given accounts in their natural order, that's the reason for the sort. If you can't acquire the lock on either of them within one second, or whatever duration you're comfortable with, you simply throw a LockException. If you gain the locks, you complete the transfer if sufficient funds are available.

Download scalabilityAndTreadSafety/locking/AccountService.java

```
public class AccountService {
  public boolean transfer(final Account from, final Account to, final int amount)
    throws LockException, InterruptedException {
    final Account[] accounts = new Account[] { from, to};
    Arrays.sort(accounts);

    if(accounts[0].monitor.tryLock(1, TimeUnit.SECONDS)) {
      try {
        if (accounts[1].monitor.tryLock(1, TimeUnit.SECONDS)) {
          try {
            if(from.withdraw(amount)) {
              to.deposit(amount);
              return true;
            } else {
              return false;
            }
          } finally {
            accounts[1].monitor.unlock();
          }
        }
      } finally {
        accounts[0].monitor.unlock();
      }
    }

    throw new LockException("Unable to acquire locks on the accounts");
  }
}
```

The transfer() method avoided deadlock by ordering the accounts and avoided livelock by limiting the time it waits to acquire the locks. Since the monitors used are reentrant, subsequent calls to lock() within the deposit() and withdraw() method caused no harm.

While the transfer() method avoided deadlock and livelock, the deposit() and withdraw() methods' use of lock() may potentially cause problems in another context. As an exercise, I suggest that you modify these methods to use tryLock() instead of lock().

This may be a departure from what you're used to. However, it's better to use the Lock interface and the supporting classes instead of the old **synchronized** construct.

Even though the Lock interface has eased concerns, it's still a lot of work and easy to err. In later chapters we'll achieve a worthy design goal—entirely avoid locks/synchronization—and achieve lock-free concurrency.

4.6 Recap

The modern concurrency API java.util.concurrency has improved a number of things over the older API. It allows you to better manage pools of threads, gives fine grained synchronization, and easy data exchange. You can enjoy better concurrent performance using the concurrent data structures. There's a caveat though: you have to be vigilant to avoid race conditions if you use shared mutable data. We'll discuss some of the ways to deal with shared mutability in the next chapter.

The beaten path is the safest but the traffic's terrible!
► Jeff Taylor

Chapter 5
Taming Shared Mutability

I've discouraged shared mutability quite a few times so far in this book. You may wonder, therefore, why I discuss it further in this chapter. The reason is quite simple: it's been the way of life in Java and you're likely to confront legacy code that's using shared mutability.

I certainly hope you'll heavily lean towards isolated mutability or pure immutability for any new code, even in existing projects. My goal in this chapter is to help cope with legacy code, the menacing code you've soldiered to refactor.

5.1 Shared Mutability != public

Shared mutability is not restricted to **public** fields. You may say, "gee, all my fields are **private**, so I have nothing to worry about," but it's not that simple.

A shared variable is accessed, for read or write, by more than one thread. On the other hand a variable that's never accessed by more than one thread—ever—is isolated and not shared. Shared mutable variables can really mess things up if you fail to ensure visibility or avoid race conditions. It's rumored that shared mutability is the leading cause of insomnia among Java programmers.

Irrespective of access privileges, you must ensure that any value you pass to other methods as parameters is thread safe. You must assume that the methods you call will access the passed instance from more than one thread. So passing an instance that's not thread safe will not help you sleep better at night. The same concern exists with references you return from your own methods. In other words, don't let any non-thread-safe references *escape*. See *Java Concurrency in Practice* [Goe06] for an extensive discussion of how to deal with escaping.

Escaping is quite tricky, you may not even realize until you closely examine code that it's there. In addition to passing and returning references, your variables may escape if you directly set references into other objects or into static fields. You may also escape a variable if you passed it into a collection, like the BlockingQueue we discussed previously. Don't be surprised if the hair on the back of your neck stands up the next time you open code with mutable variables.

5.2 Spotting Concurrency Issues

Let's learn to identify the perils in shared mutability with an example and see how to fix those problems. We'll refactor a piece of code that controls a fancy energy source. It allows users to drain energy and it automatically replenishes the source at regular intervals. Let's first glance at the code that's crying for our help.

Download tamingSharedMutability/originalcode/EnergySource.java

```java
package com.agiledeveloper.pcj;
//Bad code
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private boolean keepRunning = true;

  public EnergySource() {
    new Thread(new Runnable() {
      public void run() { replenish(); }
    }).start();
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { keepRunning = false; }

  private void replenish() {
    while(keepRunning) {
      if (level < MAXLEVEL) level++;

      try { Thread.sleep(1000); } catch(InterruptedException ex) {}
    }
  }
}
```

Identify the concurrency issues in the EnergySource class. There are a few easy to spot problems, but some hidden treasures as well, so take your time.

Done? OK, let's go over it. The EnergySource's methods may be called from any thread. So, the non-final private variable level is a shared mutable variable, but it's not thread safe. You have unprotected access to it, from thread-safety viewpoint, in most of the methods. That leads to both the visibility concern—calling thread may not see the change, as it was not asked to cross the memory barrier—and race condition.

That was easy to spot, but there's more.

The replenish() method spends most of the time sleeping, but it's wasting an entire thread. If you try to create a large number of EnergySources, you'll get an OutOfMemoryError due to the creation of too many threads—typically the JVM will allow you to create only a few thousand threads.

The EnergySource breaks the class invariant.[1] A well-constructed object ensures that none of its methods are called before the object is in a valid state. However, the EnergySource's constructor violated invariant when it invoked the replenish() method from another thread before the constructor completed. Also, Thread's start() method automatically inserts a memory-barrier, and so escapes the object before its initiation is complete. Starting threads from within constructors is a real bad idea. More on this in the next section.

That's quite a few issues for such a small piece of code, eh? Let's fix them one by one. I prefer not to fix problems concurrently, so I can focus on solving each in turn.

1. Class invariant is a condition that every object of the class must satisfy at all times—see *What Every Programming Should Know About Object-Oriented Design* [PJ95] and *Object-Oriented Software Construction* [Mey97]. In other words, you should never be able to access an object in an invalid state.

5.3 Preserve Invariant

We may be tempted to start threads from constructors to get background tasks running as soon as an object is instantiated. That's a good intention with undesirable side-effects. The call to start() forces a memory barrier, exposing the partially created object to other threads. Also, the thread we started may invoke methods on the instance before its construction is complete.

An object should preserve its invariant, and therefore starting threads from within constructors is forbidden.

The EnergySource is clearly in violation on this count. One way to fix this is to move the thread starting code from the constructor to a separate instance method. However, that creates a new set of problems. We have to deal with method calls that may arrive before the thread-starting method is called or a programmer may simply forget to call it. We could put a flag to deal with that, but that'd lead to ugly duplicated code. We also have to prevent the thread-starting method from being called more than once on any instance.

On the one hand, you shouldn't start threads from constructors, and on the other hand, you don't want to open up your instance for any use without fully creating it to your satisfaction. There's gotta be a way to get out of this pickle.

The answer's in the first item in *Effective Java* [Blo08]: "Consider static factory methods instead of constructors." Create the instance in the static factory method and start the thread before returning the instance to the caller.

Download tamingSharedMutability/fixingconstructor/EnergySource.java

```java
//Fixing constructor...other issues pending
private EnergySource() {}

private void init() {
  new Thread(new Runnable() {
    public void run() { replenish(); }
  }).start();
}

public static EnergySource create() {
  final EnergySource energySource = new EnergySource();
  energySource.init();
  return energySource;
}
```

We keep the constructor private and uncomplicated. We could perform simple calculations in the constructor, but avoid any method calls here. The private method init() does the bulk of the work we did earlier in the constructor. Invoke this method from within the static factory method create(). We avoided the invariant violation and, at the same time, ensured that our instance is in a valid state with its background task started upon creation.

Look around your own project, do you see threads being started in constructors? If you do, you have another cleanup task to add to your refactoring tasks list.

5.4 Mind Your Resources

Threads are limited resources and you shouldn't create them arbitrarily. It's better to reuse threads from a thread-pool. You can create and manage your own thread pool, if you'd like. Alternately, where possible, you may rely on timers, which make good use of pools. If you need more flexibility than timers, like handling uncaught exceptions, use a ScheduledThreadPoolExecutor.

The EnergySource's replenish() method is wasting a thread and limits the number of instances we can create. If more instances created their own threads like that, we'd run into resource availability problems. The replenish operation is short and quick, so it's an ideal candidate to run in a timer.

Let's refactor EnergySource to use the timer.

Download tamingSharedMutability/usingtimer/EnergySource.java

```java
//Using Timer...other issues pending
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private final Timer replenishTimer = new Timer(true);

  private EnergySource() {}

  private void init() {
    TimerTask timerTask = new TimerTask() {
      public void run() { replenish(); }
    };
    replenishTimer.scheduleAtFixedRate(timerTask, 0, 1000);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() { return level; }

  public boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { replenishTimer.cancel(); }

  private void replenish() { if (level < MAXLEVEL) level++; }
}
```

In addition to being beneficial for resource usage, the timer made the code simpler. We got rid of the keepRunning field and simply cancelled the timer in the stopEnergySource() method. Instead of starting a thread for each instance of EnergySource, the init() method scheduled the timer to run the replenish() method. This method, in turn, got even simpler—we're not concerned about the sleep or the timing, so instead we focus on the logic to increase the energy level.

Our EnergySource just lost a few pounds and is more scalable than when we created the thread internally.

Examine your own project to see where you're creating threads, especially using the Thread class. Evaluate those situations to see if you can either use a timer or delegate that task to run in an ExecutorService's thread pool.

5.5 Ensure Visibility

The memory barrier has to be crossed at the appropriate time. Crossing it in the constructor is not good, and we fixed that problem in the example. However, we must ensure that other methods that access the shared mutable variable level cross the memory barrier—See the *Joe Asks...* on page 24 to refresh your memory about the reasons to cross the memory barrier.

If you only consider race conditions, you may argue against synchronizing getter methods; you may get convinced that slightly old copy of a variable's value is adequate. The motivation to synchronize or lock getters is as much about visibility as race conditions—if you fail to cross the memory barrier, your thread is not guaranteed to see the change for an unpredictable duration of time in the future.

Let's ensure that the methods of EnergySource cross the memory barrier. The easiest way to cross the memory barrier is to use the **synchronized** keyword. Let's use that now to provide visibility and then improve on it later—I try to follow the "make it work; make it better" motto.

Each method that touches the mutable variable level to read or write needs to cross the memory barrier, so we'll mark them all as synchronized. The original version will not allow the replenish() method to be marked synchronized since it looped indefinitely. The latest version here has no such problem, since it's a short, quick method invoked from the timer. Here's the new version that ensures visibility.

Download tamingSharedMutability/ensurevisibility/EnergySource.java

```java
//Ensure visibility...other issues pending
  //...
  public synchronized long getUnitsAvailable() { return level; }

  public synchronized boolean useEnergy(final long units) {
    if (units > 0 && level >= units) {
      level -= units;
      return true;
    }
    return false;
  }

  public void stopEnergySource() { replenishTimer.cancel(); }

  private synchronized void replenish() { if (level < MAXLEVEL) level++; }
}
```

We marked the methods getUnitsAvailable(), useEnergy(), and replenish() as synchronized since these are the methods that touch the level variable. There's no reason to disturb the stopEnergySource() method as it only uses a **final** field and the Timer takes care of its own internal thread safety and visibility concerns.

Examine your own project and see if all access to mutable variables, both getters and setters, cross memory barrier using synchronized or one of the other relevant constructs. Access to final variables don't need to cross memory barriers, since they don't change and cached values are as good as the one in memory.

5.6 Enhance Concurrency

I could build a moat filled with alligators around my house to provide safety, but that would make it a challenge for me to get in and out each day. Overly conservative synchronization is like that, it can provide thread safety, but makes code slow. You want to ensure that the synchronization happens at the right level for each class so you don't compromise thread safety, but still enjoy good concurrency.

Synchronizing instances is fairly common, but has some problems. For one, its scope is the entire object and that becomes the level of granularity for concurrency. This limits you to at most one synchronized operation on the entire object at anytime. If all the operations on the object are mutually exclusive, like add and remove on a collection, then this is not entirely bad, though it can certainly be better. However, if your object can support multiple operations, like drive() and sing(), that can run concurrently but need to be synchronized with other operations, like drive() and tweet(), which should be mutually exclusive, synchronizing on the instance will not help with speed. In this case, you should have multiple synchronization points within your objects for these methods.

EnergySource managed to ensure visibility and thread safety. However, the synchronization is overreaching. There's little reason to be synchronizing on the instance in this case. Let's fix that.

Since the level was the only mutable field in the class, we can move synchronization to that directly. This will not work all the time, however. If you had more than one field, you may have to protect access around changes to multiple fields. So, you may have to provide one or more explicit Lock instances, as we'll see later.

We've determined that it would be nice to move the synchronization to the level variable. However, there's a slight problem, in that Java does not allow you to lock on primitive types like **long**. You can get around that constraint by changing the variable to AtomicLong from long. This will provide fine grained thread safety around the access to this variable. Let's take a look at the modified code first before we discuss further.

Download tamingSharedMutability/enhanceconcurrency/EnergySource.java

```java
public class EnergySource {
  private final long MAXLEVEL = 100;
  private final AtomicLong level = new AtomicLong(MAXLEVEL);
  private final Timer replenishTimer = new Timer(true);

  private EnergySource() {}

  private void init() {
    TimerTask timerTask = new TimerTask() {
     public void run() { replenish(); }
    };
    replenishTimer.scheduleAtFixedRate(timerTask, 0, 1000);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() { return level.get(); }

  public boolean useEnergy(final long units) {
    final long currentLevel = level.get();
    if (units > 0 && currentLevel >= units) {
      return level.compareAndSet(currentLevel, currentLevel - units);
    }
    return false;
  }

  public void stopEnergySource() { replenishTimer.cancel(); }

  private void replenish() { if (level.get() < MAXLEVEL) level.incrementAndGet(); }
}
```

We got rid of the synchronized marking from getUnitsAvailable(), since the AtomicLong takes care of thread safety and visibility for access to the value it holds.

We also removed the synchronized from the method useEnergy(). However, the improved concurrency comes with a slight change in semantics. Earlier we locked out any access while we examined the availability of energy. If we found enough energy, we were guaranteed to have it. However, that lowered concurrency, while one thread took the time, all others interacting with the EnergySource were blocked. In this improved version multiple threads can compete for the energy at the same time without holding exclusive locks. If two or more threads compete at the same time, one of them would succeed and the others would simply have to retry. You have much better speed for reads and safety for writes as well.

The replenish() is also not asking for an exclusive lock. It grabs the level in a thread-safe manner and then, without holding any lock, increments the value. This is OK since there is only one thread increasing the energy level. If the method found the value to be lower than MAXLEVEL it remains lower, even though the value may decrease. The increment itself is thread safe, and there is no issue of consistency.

Revisit your own project and look for places where you can improve concurrency without compromising thread safety. Check if there are places where you can introduce lock objects instead of synchronizing on the entire instance. While at it, ensure proper synchronization is in place for all methods that deal with mutable state, both reads and writes.

5.7 Ensure Atomicity

The previous example had no explicit synchronization in code, but any related euphoria is short lived. You can't avoid synchronization if your mutable state has more than one related or dependent variable and you use the JDK concurrency API, let's see how.

Our success with the refactoring efforts so far did not go unnoticed. We've now been asked to make an enhancement, keep track of the energy source's usage. Each time the energy source is drained, we need to keep a count. This means we need to ensure that the change to level and to the new variable usage are atomic. In other words, changes to both of them in a thread should be kept or both of them discarded. We should never be able to see change to one of the fields and not the other.

This brings back the need for explicit synchronization in code. Using synchronized is the simplest option, however, that would limit concurrency among multiple readers. If you don't mind a reader blocking another reader, then favor this simplicity. However, if you like a greater degree of concurrency and don't mind trading some complexity for that, then you may use a ReentrantReadWriteLock. This provides a pair of locks, a read lock and a write lock, that readers and writers can use, respectively. Using this lock will allow you to have multiple concurrent readers or one exclusive writer at any given instance. Let's use this lock in the example. Since we're using explicit locks, we can scale back to a simple long instead of the AtomicLong for the level field.

Download tamingSharedMutability/ensureatomicity/EnergySource.java

```java
public class EnergySource {
  private final long MAXLEVEL = 100;
  private long level = MAXLEVEL;
  private long usage = 0;
  private final ReadWriteLock monitor = new ReentrantReadWriteLock();
  private final Timer replenishTimer = new Timer(true);

  private EnergySource() {}

  private void init() {
    TimerTask timerTask = new TimerTask() {
     public void run() { replenish(); }
    };
    replenishTimer.scheduleAtFixedRate(timerTask, 0, 1000);
  }

  public static EnergySource create() {
    final EnergySource energySource = new EnergySource();
    energySource.init();
    return energySource;
  }

  public long getUnitsAvailable() {
    monitor.readLock().lock();
    try {
      return level;
    } finally {
      monitor.readLock().unlock();
    }
  }

  public long getUsageCount() {
    monitor.readLock().lock();
    try {
      return usage;
    } finally {
      monitor.readLock().unlock();
    }
  }

  public boolean useEnergy(final long units) {
    monitor.writeLock().lock();
    try {
      if (units > 0 && level >= units) {
        level -= units;
        usage++;
        return true;
      } else {
        return false;
      }
    } finally {
      monitor.writeLock().unlock();
    }
  }

  public void stopEnergySource() { replenishTimer.cancel(); }

  private void replenish() {
    monitor.writeLock().lock();
    try {
      if (level < MAXLEVEL) {
        level++;
      }
    } finally {
      monitor.writeLock().unlock();
    }
  }
}
```

We introduced two new fields, usage, to keep track of how many times the energy source is drained, and monitor, an instance of ReentrantReadWriteLock, to ensure the changes to the two mutable variables are atomic. In the useEnergy() method, we first acquire a write lock. We can use a timeout on the lock if we desire. Once the write lock is acquired, we change the level and the usage variables. Within the safe haven of the finally block we release the acquired lock.

Similarly we acquire the write lock in the replenish() method as well. This will ensure that the changes made by useEnergy() and replenish() are mutually exclusive.

We need to use a lock in the get methods to ensure the visibility of the fields and also that partial changes within useEnergy() are not seen. By using a read lock, we allow multiple gets to run concurrently and only block when a write is in progress.

This code provides fairly decent concurrency and thread safety at the same time. We do have to be vigilant to ensure that we've not compromised safety at any point. The code is a notch more complex than the previous version and this, unfortunately, makes it easier to make mistakes. In the next chapter we'll see how we can avoid explicit synchronization.

5.8 Recap

Working with shared mutable state is a huge burden that requires you to endure added complexity and the accompanied greater chances of error. When refactoring code, look out for some common concurrency related mistakes:

- Don't create threads from within constructors, instead create them in static factory methods—see Section 5.3, *Preserve Invariant*, on page 88.
- Don't create arbitrary threads, instead use a pool of threads to reduce the tasks startup time and resource use—see Section 5.4, *Mind Your Resources*, on page 89.
- Ensure that access to mutable fields cross memory barrier and are visible to your threads—see Section 5.5, *Ensure Visibility*, on page 90.
- Evaluate the granularity of your locks and promote concurrency. Ensure that the locks are not overly conservative but are at the right level to provide adequate thread safety and concurrency at the same time—see Section 5.6, *Enhance Concurrency*, on page 92.
- When working with multiple mutable fields, verify that the access to these variables are atomic, that is other threads don't see partial changes to these fields—see Section 5.7, *Ensure Atomicity*, on page 94.

Part III

Software Transaction Memory

We are governed not by armies and police, but by ideas.
► Mona Caird

Chapter 6

Introduction to Software Transactional Memory

Recall the last time you finished a project where you had to synchronize shared mutable variables. Instead of relaxing and enjoying that feeling of a job well done, you probably had a nagging feeling of doubt, wondering if you managed to synchronize in all the right places. My programming has involved quite a few such unnerving moments. That's mostly due to shared mutable state in Java failing the principle of least surprises. If we forget to synchronize, then unpredictable and potentially catastrophic results await us. But, to err is human: to forget is our nature. Rather than punish us, the tools we rely on should compensate for our deficiencies and help reach the targets our creative minds seek. If you desire predictable behavior and results, then look beyond the JDK.

In this chapter you'll learn how to play it safe with shared mutability using the Software Transactional Memory (STM) model popularized by Clojure. If you like, you can switch to or mix in Clojure on your projects. But you're not forced to use Clojure, as there are ways to use STM directly in Java, thanks to nice tools like Multiverse and Akka. Let's now delve into STM, learn a little about what it looks like in Clojure, and then how to program transactional memory in Java and Scala. This model of programming is quite suitable when you have frequent reads and very infrequent write collisions—it's simple to use, yet gives you predictable results.

6.1 Synchronization Damns Concurrency

Synchronization has some fundamental flaws.

If you use it improperly or forget it totally, the changes made by your threads may not be visible to other threads. We often learn the hard way where we need to synchronize in order to ensure visibility and avoid race conditions.

Unfortunately, when you synchronize, you force contending threads to wait. Concurrency is affected by the granularity of synchronization, so placing this in the hands of programmers increases the opportunity to make it less efficient or outright wrong.

Synchronization can lead to a number of liveness problems. It's easy to deadlock your application if it holds locks while waiting for others. It's also easy to run into livelock issues where your thread continuously fails to gain a particular lock.

You might try to improve concurrency by making the locks fine-grained or granular. While this is a good idea in general, the biggest risk is failing to synchronize at the right level. What's worst, you get no indication of failure to synchronize properly. Furthermore, you've only moved the point where the thread waits: your thread still is requesting an exclusive access and is expecting other threads to wait.

That's simply life in the big city for most Java programmers using the JDK concurrency facilities. We've been lead down the path of imperative style of programming with mutable state for so long that it's very hard to see alternatives to synchronization, but there are.

6.2 The Deficiency of the Object Model

As Java programmers, we're well versed in Object-Oriented programming (OOP). But the language has greatly influenced the way we model OO apps. OOP didn't quite turn out to be what Alan Kay had in mind when he coined the term. His vision was primarily message passing and wanted to get rid of data (he viewed systems to be built using messages that are passed between biological cell like objects that performed operations but did not hold any state)—see "The Meaning of Object-Oriented Programming" in Appendix A, on page 256. Somewhere along the way, OO languages started down the path of data hiding through Abstract Data Types (ADTs), binding data with procedure or combining state and behavior. This largely lead us towards encapsulating and mutating state. In the process we ended up merging the identity with state—the merging of the instance with its data.

This merging of identity and state had sneaked up on many Java programmers in a way that its consequences may not be obvious. When you traverse a pointer or reference to an instance you land at the chunk of memory that holds its state. It feels natural to manipulate the data at that location. The location represents the instance and what it contains. The combined identity and state worked out to be quite simple and easy to comprehend. However, this had some serious ramifications from the concurrency point of view.

Suppose you're running a report to print various details about a bank account—the number, current balance, transactions, minimum balance, etc. The reference on hand is the gateway to the state that could change anytime. So, you're forced to lock other threads while you view the account—the result: low concurrency. The problem didn't start with the locking, but with the merging of the account identity with its state.

We were told that OO programming models the real word. Sadly, the real world does not quite behave as the current OO paradigm models it. In the real world the state does not change, the identity does. You may wonder how that's true, so let's discuss that next.

6.3 Separation of Identity and State

Quick, what's the price of Google stock? You might argue that the value of the stock changes by the minute when the market's open, but that's in some way just playing with words. To take a simple example, the closing price of Google stock on December 10th, 2010, was $592.21, and that'll never change—it's immutable and written in history. You're looking at a snapshot of its value at a said time. Sure the price of Google stock today is different from the one on that day. If you check back a few minutes later (assuming the market is open), you're looking at a different value, but the older values don't change. You change the way you view objects and that changes the way you use them. Separate identity from it immutable state. We'll see how this change enables lock-free programming, improves concurrency, and reduces contention to the bare minimum.

This separation of identity from state is pure brilliance—a key step that Rich Hickey took in implementing Clojure's STM model—see "Values and Change—Clojure's approach to Identity and State" in Appendix A,

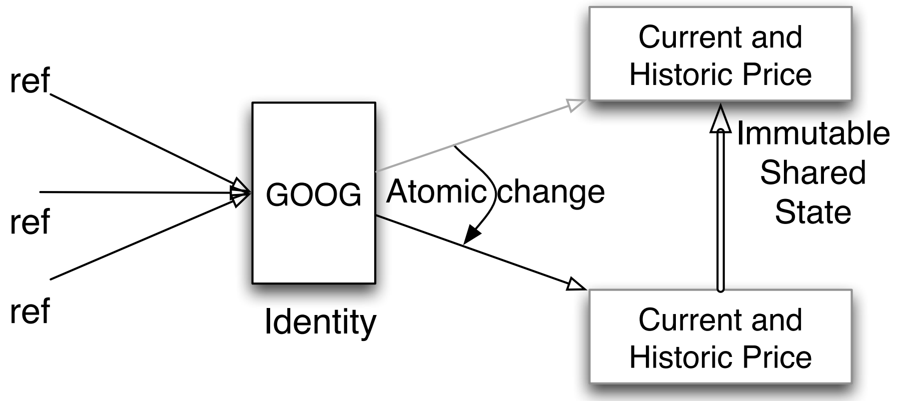


Figure 6.1: Separation of Mutable Identity from Immutable State

on page 256. Imagine your Google stock object has two parts, the first part represents the stock's identity, which in turn has a pointer to the immutable state of the latest value of the stock as shown in Figure 6.1.

When you receive a new price, you're adding to the historic price index, but not changing anything that exists. Since the old values are immutable, you share existing data. So you have no duplication and can enjoy faster operations as well if you use persistent data structures for this, as we discussed in Section 3.6, *Persistent/Immutable Data Structures*, on page 54. Once the new data is in place, you make a quick change of the identity to point to the new value.

The separation of identity from state is pure bliss for concurrency. You don't have to block any requests for stock price. Since the state does not change, you can readily hand over the pointer to the requesting thread. Any request arriving after you change the identity will see the update. Non-blocking reads means higher scalable concurrency. You simply need to ensure that the threads get a consistent view of their world. The best part of it all is: you don't have to; STM does it for you. I'm sure you're eager to learn more about STM now.

6.4 Software Transaction Memory

The separation of identity from state helps STM solve the two major concerns with synchronization: crossing the memory barrier and preventing race conditions. We'll first look at STM in the context of Clojure and then use it in Java.

Clojure removes the menacing details of memory synchronization (see *Programming Clojure* [Hal09]) by placing access to memory within *transactions*. Clojure keenly watches over and coordinates the activities of threads. If there are no conflicts, for example threads working with different accounts, then no locks are involved and there are no delays. When two threads try to access the same data, the transaction manager steps in to resolve the conflict, and again, no locking involved in your code. Let's explore how this works.

By design, states are immutable and identities are mutable only within transactions in Clojure. There's simply no way to change the state, no programing facility for that, in Clojure. Any attempts to change the identity of an object outside any transactions will fail with a stern IllegalStateException. On the other hand, when bound by transactions your changes are instantaneous when there are no conflicts. If conflicts arise, Clojure will automatically rollback your transactions and retry. It's your responsibility to ensure that the code within the transactions are idempotent—in functional programming you try to avoid side-effects, so this goes well with the programming model in Clojure.

In Clojure, states are always immutable. Clojure favors immutability of identity by default, but you can chose mutable identity by using **ref**s. A ref provides coordinated synchronous change[1] to the identity of the immutable state it represents. Let's create a ref and try to change it.

Download usingTransactionalMemory/clojure/mutate.clj

```
(def balance (ref 0))

(println "Balance is" @balance)

(ref-set balance 100)

(println "Balance now is" @balance)
```

1. Clojure provides a few other models for concurrency such as Agents, Atoms, and Vars that we will not delve into here, please refer to Clojure documentation or books if you're interested in them.

We defined a ref named balance and marked it as mutable using ref. Now balance represents a mutable identity with immutable state of 0. Then we print the current value of this variable. Next we attempt to modify balance using the command **ref-set**. It this succeeded, you should see the new balance printed by the last statement. Let's see how this turns out, I'm simply running this script using clj mutable.clj—refer to *Programming Clojure* [Hal09] or Clojure documentation on how to install and run on your system.

```
Balance is 0
Exception in thread "main"
  java.lang.IllegalStateException: No transaction running (mutate.clj:0)
...
```

The first two statements worked well: we were able to print the initial balance of 0. Our effort to change the value, however, failed with a IllegalStateException. We made the Clojure gods angry by mutating the variable outside of transactions. Contrast this clear failure to the nefarious behavior of Java when you modify a shared mutable variable without synchronization. This is quite an improvement since you'd much rather have your code behave right or fail loudly than quietly produce unpredictable results. I'd invite you to celebrate, but I see you're eager to fix that blunt error from Clojure STM, so let's move on.

Creating a transaction is easy in Clojure. You simply wrap the block of code in a **dosync** call. Structurally this feels like the **synchronized** block in Java, but there are two main differences. First, you get a clear warning if you forget to place **dosync** around mutating code. Second, rather than create an exclusive lock, **dosync** lets your code fairly compete with other threads by wrapping it in a transaction. If there was no conflict with other threads/transactions, your thread completes, and its changes are written to the memory. However, as soon as Clojure finds that some other transaction has progressed far ahead in a way to jeopardize your transaction, it quietly rolls back your changes and repeats your transaction. Let's fix the code so you can successfully change the balance variable.

Download usingTransactionalMemory/clojure/mutatesuccess.clj

```
(def balance (ref 0))

(println "Balance is" @balance)

(dosync
  (ref-set balance 100))
```

```
(println "Balance now is" @balance)
```

Let's run the code to see the change.

```
Balance is 0
Balance now is 100
```

The state value of 0 for the balance is immutable, but the identity variable balance is mutable. Within the transaction you first created a new value 100 (you need to get used to the notion of immutable value or state), but the old value of 0 is still there and balance is pointing to it at this moment. Once you created the new value, you ask Clojure to quickly modify the pointer in balance to the new value. If there are no other references to the old value, the garbage collector will take care of it appropriately.

In addition to refs, Clojure also provides **atoms**. These provide synchronous change to data, however, unlike refs, the changes are uncoordinated and can't be grouped with other changes in a transaction. Atoms don't participate in transactions (you can think of each change to atom to belong to a separate transaction). If you want isolated discrete changes, use atoms; for more coordinated or grouped changes, use refs.

6.5 Transactions in STM

You've undoubtedly used transactions in databases and are familiar with their Atomicity, Consistency, Isolation, and Durability or ACID properties. Clojure's STM provides the first three of these properties, but it can't provide durability, as the data is entirely in memory and not in a database or file system.

Atomicity: STM transactions are atomic. Either all the changes you make in a transaction are visible outside of the transaction or none of them are visible. All changes to **ref**s in a transaction are kept or none at all.

Consistency: The transactions either entirely run to completion and you see its net change or it fails, leaving things unaffected. From the outside of these transactions, you see one change consistently following another. For example, at the end of two separate and concurrent deposit and withdraw transactions, your balance is left in a consistent state with the cumulative effect of both actions.

Isolation: Your transactions don't see partial changes of other transactions, and your changes are seen only upon successful completion.

These properties focus on the integrity and visibility of data. However, the isolation does not mean lack of coordination. To the contrary, STM closely monitors the progress of all transactions and tries to help each one of them run to completion (barring any application-based exceptions).

Clojure STM uses Multiversion Concurrency Control (MVCC) much like databases do. STM concurrency control is similar to optimistic locking in databases. When you start a transaction, STM notes the timestamps and copies the refs your transaction will use. Since the state is immutable, this copying of refs is quick and inexpensive. When you make a "change" to any immutable state you really are not modifying the state. Instead, you're creating new copies of these states. The copies are kept local to your transaction, and thanks to persistent data structures (Section 3.6, *Persistent/Immutable Data Structures*, on page 54), this step is quick as well. If at any point STM determines that the refs you've changed have been modified by another transaction, it aborts and retries your transaction. When your transaction successfully completes, the changes are written to memory and the timestamps updated (see Figure 6.1, on page 102).

6.6 Concurrency using STM

Transactions are nice, but what happens if two transactions try to change the same identity? Sorry, I didn't mean to keep you on the edge of your seat this long. We'll take a look at a couple of examples of that in this section.

A word of caution, though, before you walk through the examples. In production code make sure that transactions are idempotent and don't have any side-effects, because they may be retried a number of times. That means no printing to the console, no logging, no sending out emails, or making any irreversible operations within transactions. If you do any of these, be ready to reverse your operations or deal with the consequences. In general, it's better to gather up these side-effect actions and perform them after your transaction succeeds.

Contrary to my advice you'll see print statements within transactions in the examples. This is purely for illustrative purpose here. Don't try this at the office!

You already know how to change the balance within a transaction. Let's now let multiple transactions compete for the balance.

Download usingTransactionalMemory/clojure/concurrentChangeToBalance.clj

```
(defn deposit [balance amount]
  (dosync
    (println "Ready to deposit..." amount)
    (let [currentBalance @balance]
      (println "simulating delay in deposit...")
      (. Thread sleep 2000)
      (ref-set balance (+ currentBalance amount))
      (println "done with deposit of " amount))))

(defn withdraw [balance amount]
  (dosync
    (println "Ready to withdraw..." amount)
    (let [currentBalance @balance]
      (println "simulating delay in withdraw...")
      (. Thread sleep 2000)
      (ref-set balance (- currentBalance amount))
      (println "done with withdraw of " amount))))

(def balance1 (ref 100))

(println "Balance1 is" @balance1)

(.start (Thread. (fn[] (deposit balance1 20))))
(.start (Thread. (fn[] (withdraw balance1 10))))

(. Thread sleep 10000)

(println "Balance1 now is" @balance1)
```

We created two transactions in this example, one for deposit and the other for withdrawal. In the function deposit() we first get a local copy of the given balance. We then simulate a delay to set the transactions on a collision course. After the delay we increase the balance. The withdraw() is very similar, except that we decrease the balance. It's time to exercise these two methods. Let's first initialize the variable balance1 to 100 and let the above two methods run in two separate threads. Go ahead and run the code and observe the output.

```
Balance1 is 100
Ready to deposit... 20
simulating delay in deposit...
Ready to withdraw... 10
simulating delay in withdraw...
done with deposit of  20
Ready to withdraw... 10
```

```
simulating delay in withdraw...
done with withdraw of  10
Balance1 now is 110
```

Both the functions obtained their own local copy of the balance. Right after the simulated delay, the deposit() transaction completed, but the withdraw() transaction was not that lucky. The balance was changed behind its back, and so the change it attempts to make is no longer valid. Clojure STM quietly aborted the transaction and retried. If you did not have these print statements you'd be oblivious of these activities. All that should matter to you is that the net effect of balance is consistent and reflects both your deposit and withdrawal.

You now know how to change a simple variable, but what if you have a collection of values? Lists are immutable in Clojure. However, you can obtain a mutable reference whose identity you can change, making it appear as though you changed the list. The list has not changed, you simply changed your view of the list. Let's explore this with an example. Suppose my family's wish list originally contains only one item, an iPad. I'd like to add a new MacBook Pro (MBP) and one of my kids wants to add a new bike. So, we set out in two different threads to add these items to the list. Here's a code for that.

Download usingTransactionalMemory/clojure/concurrentListChange.clj

```
(defn addItem [wishlist item]
  (dosync (alter wishlist conj item)))

(def familyWishList (ref '("iPad")))
(def originalWishList @familyWishList)

(println "Original wish list is" originalWishList)

(.start (Thread. (fn[] (addItem familyWishList "MBP"))))
(.start (Thread. (fn[] (addItem familyWishList "Bike"))))

(. Thread sleep 1000)

(println "Original wish list is" originalWishList)
(println "Updated wish list is" @familyWishList)
```

In the addItem() function, we add the given item to the list. **alter** can be invoked only within a transaction, and it alters the in-transaction **ref** by applying the provided function, in this case the conj() function. We then call the addItem() method from two different threads. When you run the code you should see the net effect of the two transactions.

```
Original wish list is (iPad)
```

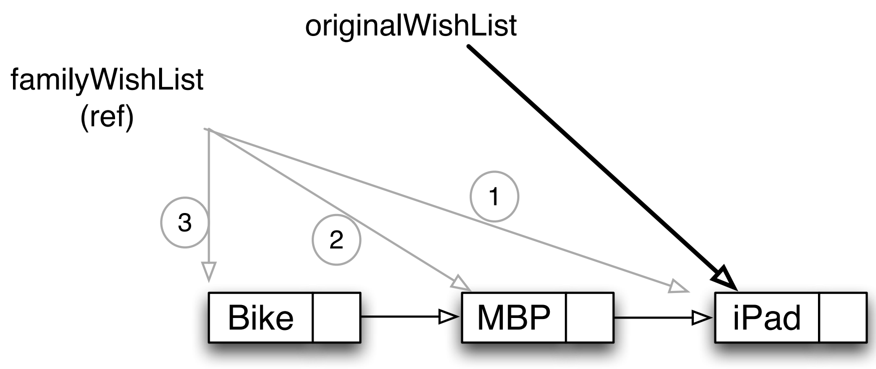


Figure 6.2: “Adding” elements to immutable wish list

```
Original wish list is (iPad)
Updated wish list is (Bike MBP iPad)
```

The original list is immutable, and so it remains the same at the end of the code. As you added elements to this list, you were getting new copies of the list. However, since a list is a persistent data structure, you gained from sharing the items (elements) under the hood as shown in Figure 6.2

Both the state of the list and the reference originalWishList are immutable. familyWishList, however, is a mutable reference. Each of the add item requests run in their own transaction. The first one to succeed changed the mutable reference to the new list (shown as (2) in the figure). However, since the list itself is immutable, the new list is able to share the element “iPad” from the original list. When the second add item succeeds, it in turn shares internally the other two items previously added (shown as (3) in the figure).

Handling Write Skew Anomaly

You saw how STM handles write conflicts between transactions. Sometimes the conflicts are not so direct. Imagine you have a checking account and a savings account in a bank that has placed minimum total balance requirements of $1000 between the two accounts. Now suppose your balance in these accounts is $500 and $600, respectively. You can withdraw up to $100 from either one of these accounts but not both. If a withdraw of $100 is attempted on each of these accounts sequentially, the first attempt will succeed, but not the second. The so-called write skew anomaly will not prevent these two transactions if done concurrently—both transactions see the total balance is over $1000 and proceed to modify two different values with no write conflicts. The net result, a balance of $900, is below the permitted limit. Let's first create this anomaly in code and then figure out how to fix it.

Download usingTransactionalMemory/clojure/writeSkew.clj

```
(def checkingBalance (ref 500))
(def savingsBalance (ref 600))

(defn withdrawAccount [fromBalance constrainingBalance amount]
  (dosync
    (let [totalBalance (+ @fromBalance @constrainingBalance)]
      (. Thread sleep 1000)
      (if (>= (- totalBalance amount) 1000)
        (ref-set fromBalance (- @fromBalance amount))
        (println "Sorry, can't withdraw due to constraint violation")))))

(println "checkingBalance is " @checkingBalance)
(println "savingsBalance is " @savingsBalance)
(println "Total balance is " (+ @checkingBalance @savingsBalance))

(.start (Thread. (fn[] (withdrawAccount checkingBalance savingsBalance 100))))
(.start (Thread. (fn[] (withdrawAccount savingsBalance checkingBalance 100))))

(. Thread sleep 2000)

(println "checkingBalance is" @checkingBalance)
(println "savingsBalance is" @savingsBalance)
(println "Total balance is " (+ @checkingBalance @savingsBalance))
```

We start with the given balances for the two accounts. In the withdrawAccount() function we first read the balance of the two accounts and compute the total balance. Then after the induced delay, which is intended to put the transactions on a collision course, we update the balance of one of the accounts represented by fromBalance only if the total balance is not less than the required minimum. In the rest of the code we concurrently run two transactions, where the first withdraws $100 from the checking balance while the second withdraws $100 from savings. Since the two transactions run in isolation and aren't modifying anything in common, they're quite oblivious of the violation and fall into the write skew as you can see from the output.

```
checkingBalance is  500
savingsBalance is  600
```

```
Total balance is  1100
checkingBalance is 400
savingsBalance is 500
Total balance is  900
```

Clojure makes it really easy to avoid write skew by means of an ensure() method. You can tell a transaction to keep an eye on a variable that is only read and not modified. STM will then ensure that the writes are committed only if the values you've read have not changed outside the transaction, it retries the transaction otherwise.

Let's modify the withdrawAccount() method.

```
Line 1  (defn withdrawAccount [fromBalance constrainingBalance amount]
     2    (dosync
     3      (ensure constrainingBalance)
     4      (let [totalBalance (+ @fromBalance @constrainingBalance)]
     5        (. Thread sleep 1000)
     6        (if (>= (- totalBalance amount) 1000)
     7          (ref-set fromBalance (- @fromBalance amount))
     8          (println "Sorry, can't withdraw due to constraint violation")))))
```

On line 3 we called ensure() on the value constrainingBalance, which we read but don't modify in the transaction. At this point, STM places a read lock on that value, preventing other transactions from gaining a write lock. When the transaction nears completion, STM releases all read locks before it performs the commit. This prevents any possibility of deadlock while increasing concurrency.

Even though the two transactions run concurrently as before, with this modified withdraw() method calling ensure(), the constraint of total balance is preserved as you see in the output.

```
checkingBalance is  500
savingsBalance is  600
Total balance is  1100
checkingBalance is 500
savingsBalance is 500
Total balance is  1000
Sorry, can't withdraw due to constraint violation
```

The lock-free execution model of STM is quite powerful. When there are no conflicts, there's no blocking at all. When there are conflicts, one winner proceeds without a hitch, while the contenders are repeated. Clojure uses a max number of tries and also ensures two threads do not repeat at the same pace and end up losing out repeatedly. STM is a great model when you have frequent reads and very infrequent write collisions.

Clojure STM is the aspirin of the programming concurrency world—it removes so much pain. If we forget to create a transaction, we're sternly rebuked. For simply placing dosync at the right place, we're rewarded with high concurrency and consistency between your threads. Such low ceremony, concise, highly expressive, and very predictable behavior makes Clojure STM an option worth serious consideration.

6.7 Concurrency using Akka/Multiverse STM

You've now seen how to use STM from within Clojure. I guess by now you're curious how to use STM in Java code. By the pattern that seems to be developing, you have not one, but a few options for STM in Java:

- You can use Clojure STM from within Java. It's fairly simple—you wrap the transactional code within an implementation of the Callable interface. We'll see this in Chapter 7, *STM in Clojure, Groovy, Java, JRuby, and Scala*, on page 151.
- If you're a fan of annotations, you may prefer Multiverse's STM API.
- In addition to using STM if you plan to use actors, then you may want to reach for the Akka library.

Multiverse, created by Peter Veentjer, is a Java-based STM implementation that can be used from many JVM languages. For pure Java code, Multiverse allows you to use annotations to indicate transaction boundaries in your code. You simply mark your class with @TransactionalObject, and all methods of your class become transaction enabled. You could also mark individual methods explicitly with @TransactionalMethod. Annotation based instrumentation is not the only option Multiverse provides. For integration with other languages on the JVM, it provides a rich set of API that provide explicit control over when transaction starts and ends.

Akka, created by Jonas Bonér, is a Scala-based solution that can be used from many different JVM languages including Java. Akka provides both Actor-based concurrency and STM, and even an option to mix them both. Akka uses Multiverse for the STM implementation and providing the ACI (subset of ACID) properties. When using it with Java, you must take extra care to ensure immutable state and mutable reference (you're responsible for this in Java whereas in Clojure it was enforced by the language).

Akka provides really good performance, and since it supports both STM- and Actor-based model (discussed in Chapter 8, *Favoring Isolated Mutability*, on page 172), we'll use it to illustrate the Java STM examples in this chapter.

Transactions in Akka/Multiverse

Akka uses Multiverse's Clojure style STM for Java code. The main difference from Clojure, other than Java-introduced verbosity, is that Akka doesn't force you to create a transaction before you can access a mutable identity. If you don't provide a transaction, Akka/Multiverse wraps your access automatically in a transaction. So, when you're in a transaction, your Akka refs behave like Clojure refs, and when you're not in a transaction they behave like Clojure atoms. In other words, start a transaction if you want coordinated synchronous change; if you don't, you get uncoordinated synchronous change. In any case, Akka ensures that your changes to refs are still atomic, isolated, and consistent, while providing varying degree of granularity of coordination.

In Akka, you can configure your transaction programmatically at the transaction level or at the application/JVM level using configuration files. You can define a transaction as readonly, and Akka will not permit any changes to any Akka references in the bounds of that transaction. You can gain performance by setting your non-mutating transactions as readonly. You can control the maximum number of times Akka will retry your transactions in the event of conflicts. There are a number of other parameters you can configure—consult Akka documentation for details.

Akka extends the nested transaction (see Section 6.9, *Creating Nested Transactions*, on page 121) support of Multiverse, so from within your transactions, you can conveniently call methods that start transactions. By default these inner or nested transactions are rolled into your outer transaction.

Using Akka Refs and Transactions

Unlike Clojure, where refs were defined at the language level, Akka can't rely on any existing Java/Scala support. Instead Akka provides, as part of the akka.stm package, a managed transactional reference Ref<T> and specialized classes for primitive types like IntRef, LongRef, etc.. The Ref<T> (and the specialized references) represents the managed mutable identity to an immutable state value of type T. The onus is on you to ensure that the state value object type you're using is immutable. Types like Integer, Long, Double, String, and other immutable types fit the bill well to serve as state value objects. If you use one of your own classes, make sure it's immutable; that is, it contains only final fields.

You can create a managed transactional reference, which is an instance of Ref<T>, by providing an initial value or by omitting the value, defaulting to null. To obtain the current value from the reference, use the get() method. To change the mutable identity to point to a new state value, use the swap() method. These calls are performed within the transaction you provide, or in their own individual transactions if none is provided.

When multiple threads try to change the same managed reference, Akka ensures that changes by one are written to memory and the others are retried. The transactional facility takes care of crossing the memory barriers. That is, Akka, through Multiverse, guarantees that the changes to a managed ref committed by a transaction happen before, and are visible to, any reads of the same ref later in other transactions.

6.8 Creating Transactions

You want to create transactions to coordinate changes to multiple managed refs. The transaction will ensure these changes are atomic, that is, all the refs are committed or all of them are discarded. Outside the transactions, you'll never see any partial changes. You'll also create transactions to coordinate a read followed by a write to a single ref.

Akka was built on Scala, and you can enjoy its concise API if you're using Scala. For programmers who can't make that switch, Akka also provides a convenient API to use its features from the Java language. Alternately, you may directly use Multiverse STM from Java. We'll see how to create transactions using Akka in Java and Scala in this section.

First we need an example to apply transactions. The EnergySource class we refactored in Chapter 5, *Taming Shared Mutability*, on page 85 used explicit lock and unlock (the final version was in Section 5.7, *Ensure Atomicity*, on page 94). Let's trade those explicit locks/unlocks for Akka's transactional API.

Creating Transactions in Java

To create your own transactions, you'll extend the Atomic class. Place your code into the atomically() method of this class to wrap it into a transaction. To run your transactional code, call the execute() method of your Atomic instance, like this:

```
return new Atomic<Object>() {
  public Object atomically() {
    //code to run in a transaction...
    return resultObject;
  }
}.execute();
```

The thread that calls the execute() method runs the code within the atomically() method. However, the call is wrapped in a transaction if the caller is not already in a transaction.

Let's implement EnergySource using Akka transactions. The first step is to wrap the immutable state in mutable Akka managed references.

Download usingTransactionalMemory/java/stm/EnergySource.java

```
public class EnergySource {
  private final long MAXLEVEL = 100;
  final Ref<Long> level = new Ref<Long>(MAXLEVEL);
  final Ref<Long> usageCount = new Ref<Long>(0L);
  private final Timer replenishTimer = new Timer(true);
```

level and usageCount are declared as Akka Refs, and each of these holds an immutable Long value. As you know, you can't change the value of a Long in Java, but you can change the managed reference (the identity) safely to point to a new state value.

The code to create an instance of EnergySource and manage the timer can be simply carried over from the previous version without any change.

Download usingTransactionalMemory/java/stm/EnergySource.java

```
private EnergySource() {}

private void init() {
  TimerTask timerTask = new TimerTask() {
   public void run() { replenish(); }
  };
  replenishTimer.scheduleAtFixedRate(timerTask, 0, 1000);
}

public static EnergySource create() {
  final EnergySource energySource = new EnergySource();
  energySource.init();
  return energySource;
}

public void stopEnergySource() { replenishTimer.cancel(); }
```

The methods that return the current energy level and the usage count will use the managed references, but that requires merely calling the get() method.

Download usingTransactionalMemory/java/stm/EnergySource.java

```
public long getUnitsAvailable() { return level.get(); }

public long getUsageCount() { return usageCount.get(); }
```

Within the getUnitsAvailable() and getUsageCount() methods, the calls to get() run in their own transactions, since we didn't wrap them explicitly in a transaction.

The useEnergy() method will need an explicit transaction, since we will change the energy level based on the current value—we need to ensure that the change is consistent with the value being read. We'll use the Atomic interface and its atomically() method to wrap our code in a transaction.

Download usingTransactionalMemory/java/stm/EnergySource.java

```
public boolean useEnergy(final long units) {
  return  new Atomic<Boolean>() {
    public Boolean atomically() {
      long currentLevel = level.get();
      if(units > 0 && currentLevel >= units) {
        level.swap(currentLevel - units);
        usageCount.swap(usageCount.get() + 1);
        return true;
      } else {
        return false;
      }
    }
  }.execute();
}
```

In the useEnergy() method we decrement from the current level. We want to ensure that both the get and the set are within the same transaction. So, we wrap the operations in the atomically() method. We finally call execute() to run the sequence of operations in one transaction.

We have one more method to take care of, replenish(), which takes care of refilling the source. We have the same transactional need in this method as well, so we'll use Atomic in it too.

Download usingTransactionalMemory/java/stm/EnergySource.java

```
  private void replenish() {
    new Atomic() {
      public Object atomically() {
        long currentLevel = level.get();
        if (currentLevel < MAXLEVEL) level.swap(currentLevel + 1);
        return null;
      }
    }.execute();
  }
}
```

Let's exercise the EnergySource using a sample runner code. This code concurrently uses energy, in units of one, from different threads.

Download usingTransactionalMemory/java/stm/UseEnergySource.java

```
public class UseEnergySource {
  private static final EnergySource energySource = EnergySource.create();

  public static void main(final String[] args)
    throws InterruptedException, ExecutionException {
    System.out.println("Energy level at start: " +
      energySource.getUnitsAvailable());

    List<Callable<Boolean>> tasks = new ArrayList<Callable<Boolean>>();

    for(int i = 0; i < 10; i++) {
      tasks.add(new Callable<Boolean>() {
        public Boolean call() {
          for(int i = 0; i < 7; i++) energySource.useEnergy(1);
          return true;
        }
      });
    }

    final ExecutorService service = Executors.newFixedThreadPool(10);
    List<Future<Boolean>> results = service.invokeAll(tasks);
    for(final Future<Boolean> result : results) result.get();

    System.out.println("Energy level at end: " +
      energySource.getUnitsAvailable());
    System.out.println("Usage: " + energySource.getUsageCount());

    energySource.stopEnergySource();
    service.shutdown();
  }
}
```

To compile and run the code, you need to include the Akka related jars as shown next.

```
export AKKA_JARS="$SCALA_HOME/lib/scala-library.jar:\
$AKKA_HOME/dist/akka-stm-1.0.jar:\
$AKKA_HOME/lib_managed/compile/multiverse-alpha-0.6.2.jar:\
$AKKA_HOME/dist/akka-actor-1.0.jar:\
$AKKA_HOME/lib_managed/compile/uuid-3.2.jar:\
$AKKA_HOME/lib_managed/compile/configgy-2.0.2-nologgy.jar:\
$AKKA_HOME/lib_managed/compile/slf4j-api-1.6.0.jar:\
$AKKA_HOME/config:\
."
```

Define the classpath based on your operating system and location where Akka is installed on your system. You can use the Scala library from Scala installation, if you have it, or the one from Akka distribution. Compile the code using the javac compiler and run it using the java command as shown here:

```
javac -classpath $AKKA_JARS EnergySource.java UseEnergySource.java
java -classpath $AKKA_JARS UseEnergySource
```

Go ahead, compile and run the code. The energy source had 100 units to begin with, and we drained 70 units from various threads we created. The net result should be 30 units of energy left—depending on the timing of replenish you may see slightly different value like 31 instead of 30.

```
Energy level at start: 100
Energy level at end: 30
Usage: 70
```

As we see here, Akka quietly managed the transactions for you behind the scenes. Spend sometime to play with that code and experiment with the transactions and threads.

Creating Transactions in Scala

You saw how to create transactions in Java (and I assume you've read that part so I don't have to repeat the details here). You'll need less code to write the same thing in Scala, partly due to the concise nature of Scala, but also due to the elegant Akka API that makes use of closures/function values.

It takes a lot less effort to create transactions in Scala compared to Java. All you need is a call to the atomic() method of Stm, like so:

```
atomic {
  //code to run in a transaction....
  /* return */ resultObject
}
```

The closure/function value you pass to atomic() is run in the current thread, but within a transaction.

Here's the Scala version of EnergySource using Akka transactions.

Download usingTransactionalMemory/scala/stm/EnergySource.scala

```
class EnergySource private() {
  private val MAXLEVEL = 100L
  val level = new Ref(MAXLEVEL)
  val usageCount = new Ref(0L)
  val replenishTimer = new Timer(true)

  private def init() = {
    val timerTask = new TimerTask() { def run() = replenish() }
    replenishTimer.scheduleAtFixedRate(timerTask, 0, 1000)
  }

  def stopEnergySource() = replenishTimer.cancel()

  def getUnitsAvailable() = level.get()

  def getUsageCount() = usageCount.get()

  def useEnergy(units : Long) = {
    atomic {
      val currentLevel = level.get()
      if(units > 0 && currentLevel >= units) {
        level.swap(currentLevel - units)
        usageCount.swap(usageCount.get() + 1)
        true
      } else false
    }
  }

  private def replenish() =
    atomic { if(level.get() < MAXLEVEL) level.swap(level.get() + 1) }
}

object EnergySource {
  def create() = {
    val energySource = new EnergySource
    energySource.init()
    energySource
  }
}
```

Scala, being a fully object-oriented language, considers static methods in classes as an aberration. So the factory method create() is moved to the companion object. The rest of the code is very similar to the Java version, but it's concise when compared. You avoided the ceremony of Atomic class and the execute() method call with the elegant atomic() method call.

The code to exercise the Scala version of EnergySource is shown next. You could've used the JDK ExecutorService to manage threads like in the Java version. Or, you could use Scala actors[2] to spawn threads for each concurrent task. Each task sends a response back to the caller when done. The caller simply uses this response to block, awaiting task completion before moving on.

Download usingTransactionalMemory/scala/stm/UseEnergySource.scala

```
object UseEnergySource {
  val energySource = EnergySource.create()

  def main(args : Array[String]) {
    println("Energy level at start: " + energySource.getUnitsAvailable())

    val caller = self
    for(i <- 1 to 10) actor {
      for(j <- 1 to 7) energySource.useEnergy(1)
      caller ! true
    }

    for(i <- 1 to 10) { receiveWithin(1000) { case _ => } }

    println("Energy level at end: " + energySource.getUnitsAvailable())
    println("Usage: " + energySource.getUsageCount())

    energySource.stopEnergySource()
  }
}
```

Compile and run the code using the Akka related jars as shown next, where AKKA_JARS is the same as defined in the Java example.

```
scalac -classpath $AKKA_JARS *.scala
java -classpath $AKKA_JARS UseEnergySource
```

The output should be no different from what you saw with the Java version, again depending on the timing of replenish you may see slightly different value like 31 instead of 30..

```
Energy level at start: 100
Energy level at end: 30
Usage: 70
```

2. Scala's actor is used here for convenience. Later we'll see how to use the more powerful Akka actors.

6.9 Creating Nested Transactions

The methods you call may create their own transactions, and their changes will get independent commits. That's not adequate if you want to coordinate the transactions in these methods into one atomic operation. You can achieve such coordination with nested transactions.

With nested transactions, all the transactions created by methods you call get rolled into your calling method's transaction by default. Akka/Multiverse provide ways to configure other options like new isolated transactions, among others. As a result, with nested transactions, all the changes get committed only when the outermost transaction commits. Your one responsibility is to ensure that methods complete within the configurable timeout period for the overall nested transactions to succeed.

The AccountService and its transfer() method from Section 4.5, *The Lock interface*, on page 80, will benefit from nested transactions. The previous version of the transfer() method had to sort the accounts in natural order and manage locks explicitly. STM removes all that burden from your shoulders. Let's first use nested transactions in that example in Java and then see how that would look like in Scala.

Nested Transactions in Java

Let's first get the Account class into the realm of transactions. The account balance should be a managed reference, so let's start with defining that field and a getter for it.

Download usingTransactionalMemory/java/nested/Account.java

```java
public class Account {
  final private Ref<Integer> balance = new Ref<Integer>();

  public Account(int initialBalance) { balance.swap(initialBalance); }

  public int getBalance() { return balance.get(); }
```

In the constructor, we set the initial balance to the given amount by calling the swap() method on the ref. This operation will run in its own transaction, since we didn't provide one (and let's assume the caller didn't provide one either). Similarly the getBalance() will access the balance in its own transaction.

The entire deposit() method has to run in one transaction, because it involves both reading and changing the balance. So, let's wrap these two operations into a separate transaction.

Download usingTransactionalMemory/java/nested/Account.java

```java
public void deposit(final int amount) {
  new Atomic<Boolean>() {
    public Boolean atomically() {
      System.out.println("Deposit " + amount);
      if (amount > 0) {
        balance.swap(balance.get() + amount);
        return true;
      }

      throw new AccountOperationFailedException();
    }
  }.execute();
}
```

Likewise the operations in withdraw() should be wrapped into a separate transaction.

Download usingTransactionalMemory/java/nested/Account.java

```java
  public void withdraw(final int amount) {
    new Atomic<Boolean>() {
      public Boolean atomically() {
        int currentBalance = balance.get();
        if (amount > 0 && currentBalance >= amount) {
          balance.swap(currentBalance - amount);
          return true;
        }

        throw new AccountOperationFailedException();
      }
    }.execute();
  }
}
```

Transactions are forced to fail upon exception, so we use this to indicate the failure when sufficient balance is not available or when the deposit/withdraw amount is invalid. Pretty simple, eh? No need to worry about synchronization, locking, deadlocking, etc.

It's time to visit the AccountService class that'll perform the transfer. Let's take a look at the transfer method first.

Download usingTransactionalMemory/java/nested/AccountService.java

```java
public class AccountService {
  public void transfer(final Account from, final Account to, final int amount) {
    new Atomic<Boolean>() {
       public Boolean atomically() {
         System.out.println("Attempting transfer...");
         to.deposit(amount);
         System.out.println("Simulating a delay in transfer...");
```

```
          try { Thread.sleep(5000); } catch(Exception ex) {}
          System.out.println("Uncommited balance after deposit $" +
            to.getBalance());
          from.withdraw(amount);
          return true;
        }
      }.execute();
    }
```

The objective in this example is to set the transactions on a collision course to illustrate their behavior while we study nested transactions. The operations in the transfer() method are bound within a transaction. As part of transfer, we first deposit money into the target account. Then, after the delay induced to cause transaction collision, we withdraw money from the source account. We want the deposit into the target account to succeed if and only if the withdraw from the source succeeds—that's the job for our transaction.

We can see whether the transfer succeeded or not by printing the balance. A convenience method to invoke the transfer(), handle the exception, and finally print the balances will be nice, so let's write one.

Download usingTransactionalMemory/java/nested/AccountService.java

```
public static void transferAndPrintBalance(Account from, Account to, int amount) {
  boolean result = true;
  try {
    new AccountService().transfer(from, to, amount);
  } catch(AccountOperationFailedException ex) {
    result = false;
  }

  System.out.println("Result of transfer is " + (result ? "Pass" : "Fail"));
  System.out.println("From account has $" + from.getBalance());
  System.out.println("To account has $" + to.getBalance());
}
```

The last method we need is the main() to get the transfer rolling.

Download usingTransactionalMemory/java/nested/AccountService.java

```
  public static void main(final String[] args) throws Exception {
    final Account account1 = new Account(2000);
    final Account account2 = new Account(100);

    final ExecutorService service = Executors.newSingleThreadExecutor();
    service.submit(new Runnable() {
      public void run() {
        try { Thread.sleep(1000); } catch(Exception ex) {}
        account2.deposit(20);
      }
    });
    service.shutdown();

    transferAndPrintBalance(account1, account2, 500);

    System.out.println("Making large transfer...");
    transferAndPrintBalance(account1, account2, 5000);
  }
}
```

In the main() method, we created two accounts and started the deposit of $20 in a separate thread. In the mean time, we ask money to be transferred between the accounts. This should set the two transactions on a collision course, as they both affect a common instance. One of them will succeed and the other will be retried. Finally, in the end we attempt a transfer amount in excess of available balance. This will illustrate that the two transactions in deposit and withdraw are not independent, but nested and atomic within the transfer's transaction. The effect of deposit should be reversed due to the failure of withdraw. Let's watch the behavior of these transactions in the output.

```
Attempting transfer...
Deposit 500
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Deposit 20
Uncommited balance after deposit $600
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Uncommited balance after deposit $620
Result of transfer is Pass
From account has $1500
To account has $620
Making large transfer...
Attempting transfer...
Deposit 5000
Simulating a delay in transfer...
Uncommited balance after deposit $5620
Result of transfer is Fail
From account has $1500
To account has $620
```

It's a bit odd that the transfer transaction got retried at the start. Such unexpected retries are due to the default speculative configuration that are optimized for a single transactional object. There are ways to configure this behavior, but that has performance consequences. Refer to Akka/Multiverse documentation to learn about the ramifications of changing it.

The deposit of $20 succeeded first. While this happened, the transfer transaction was in the middle of its simulated delay. As soon as the transaction realizes that an object it manages has changed behind its back, it quietly rolls back and starts another attempt. The retries will continue until it succeeds or the timeout is exceeded. This time around the transfer transaction succeeded. The net result of both the transactions is reflected in the balance displayed—the first account lost $500 in the transfer while the second account gained a total of $520 from the concurrent deposit and transfer activities.

Our next and final attempt was to transfer $5000. The deposit was completed but the change within that transaction was withheld to check the fate of the withdraw. The withdraw, however, failed with an exception due to insufficient balance. This caused the pending deposit transaction to rollback, leaving the balance unaffected by the final transfer attempt.

The print message and delays are certainly not good practices, but I used them here so you can see the transaction sequences and retries—avoid these kinds of printing or logging in real code. Remember: transactions should have no side-effects.

I promised that the transactions would lift much burden from your shoulders, so let's find out just how much. Take a look at the transfer() method from Section 4.5, *The Lock interface*, on page 80 shown here for your convenience.

Download scalabilityAndTreadSafety/locking/AccountService.java

```java
public boolean transfer(final Account from, final Account to, final int amount)
  throws LockException, InterruptedException {
  final Account[] accounts = new Account[] { from, to};
  Arrays.sort(accounts);

  if(accounts[0].monitor.tryLock(1, TimeUnit.SECONDS)) {
    try {
      if (accounts[1].monitor.tryLock(1, TimeUnit.SECONDS)) {
        try {
          if(from.withdraw(amount)) {
            to.deposit(amount);
            return true;
          } else {
            return false;
          }
        } finally {
          accounts[1].monitor.unlock();
        }
      }
    } finally {
      accounts[0].monitor.unlock();
    }
  }

  throw new LockException("Unable to acquire locks on the accounts");
}
```

Contrast that code with your latest version, but first get rid of the delay and print statements.

```
public void transfer(final Account from, final Account to, final int amount) {
  new Atomic<Boolean>() {
     public Boolean atomically() {
       to.deposit(amount);
       from.withdraw(amount);
       return true;
     }
  }.execute();
}
```

There was so much dancing around with ordering and locking in the old version that it's easy to mess up. The code we don't write has the fewest bugs, hands down—in the new version you reduced both code and complexity. This reminds me of C.A.R. Hoare's words "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies." Less code, lesser complexity, more time to devote to your application logic.

Nested Transactions in Scala

Nested transactions already make the code quite concise in the Java version transfer method, so you may wonder if Scala can really help any further. Sure, nested transactions removed the synchronization cruft, but there's still some Java syntax that can use some cleaning.

Here's the Scala version of the Account class.

Download usingTransactionalMemory/scala/nested/Account.scala

```
class Account(val initialBalance : Int) {
  val balance = new Ref(initialBalance)

  def getBalance() = balance.get()
  def deposit(amount : Int) = {
    atomic {
      println("Deposit " + amount)
      if(amount > 0)
        balance.swap(balance.get() + amount)
      else
        throw new AccountOperationFailedException()
    }
  }

  def withdraw(amount : Int) = {
    atomic {
      val currentBalance = balance.get()
      if(amount > 0 && currentBalance >= amount)
        balance.swap(currentBalance - amount)
      else
        throw new AccountOperationFailedException()
    }
  }
}
```

The Scala version of Account is a direct translation from the Java version with Scala and Akka conciseness. You see the same advantage in the Scala version of AccountService.

Download usingTransactionalMemory/scala/nested/AccountService.scala

```
object AccountService {
  def transfer(from : Account, to : Account, amount : Int) = {
    atomic {
      println("Attempting transfer...")
      to.deposit(amount)
      println("Simulating a delay in transfer...")
      Thread.sleep(5000)
      println("Uncommited balance after deposit $" + to.getBalance())
      from.withdraw(amount)
    }
  }

  def transferAndPrintBalance(from : Account, to : Account, amount : Int) = {
    var result = "Pass"
    try {
      AccountService.transfer(from, to, amount)
    } catch {
      case _ => result = "Fail"
    }

    println("Result of transfer is " + result)
    println("From account has $" + from.getBalance())
    println("To account has $" + to.getBalance())
  }
  def main(args : Array[String]) = {
    val account1 = new Account(2000)
    val account2 = new Account(100)

    actor {
      Thread.sleep(1000)
      account2.deposit(20)
    }

    transferAndPrintBalance(account1, account2, 500)

    println("Making large transfer...")
    transferAndPrintBalance(account1, account2, 5000)
  }
}
```

This version sets the transactions on the collision course just like the Java version did. Not surprisingly, the output is the same as the Java version's.

```
Attempting transfer...
Deposit 500
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Deposit 20
Uncommited balance after deposit $600
Attempting transfer...
Deposit 500
Simulating a delay in transfer...
Uncommited balance after deposit $620
Result of transfer is Pass
From account has $1500
To account has $620
Making large transfer...
Attempting transfer...
Deposit 5000
Simulating a delay in transfer...
Uncommited balance after deposit $5620
Result of transfer is Fail
From account has $1500
To account has $620
```

We've already compared the synchronized version of transfer with the Java version using nested transaction (which is repeated next).

```
public void transfer(final Account from, final Account to, final int amount) {
  new Atomic<Boolean>() {
     public Boolean atomically() {
       to.deposit(amount);
       from.withdraw(amount);
       return true;
```

```
    }
  }.execute();
}
```

Now compare it with the Scala version.

```
def transfer(from : Account, to : Account, amount : Int) = {
  atomic {
    to.deposit(amount)
    from.withdraw(amount)
  }
}
```

Nothing but the most essential code. It reminds me of the quotation by Alan Perlis, "A programming language is low level when its programs require attention to the irrelevant."

6.10 Configuring Transactions

Akka assumes a number of default settings, but it allows you to change these either programmatically or through the configuration file akka.conf. See the Akka documentation for details on how to specify or change the location of the configuration file.

If you'd like to change settings on a per-transaction basis, you have to do it programmatically using a TransactionFactory. Let's change some settings programmatically from Java and then from Scala.

Configuring Transactions in Java

You extended Atomic to implement transactions in Java. You can provide an optional constructor parameter of type TransactionFactory to change the transaction properties. For example, you can set a transaction as readonly to gain performance and prevent changes to any references by configuring your transaction. Let's create a class CoffeePot that holds a number of cups of coffee and try to manipulate it from within a readonly transaction.

Download usingTransactionalMemory/java/configure/CoffeePot.java

```
public class CoffeePot {
  private static final Ref<Integer> cups = new Ref<Integer>(24);

  public static int readWriteCups(final boolean write) {
    TransactionFactory factory =
      new TransactionFactoryBuilder().setReadonly(true).build();

    return new Atomic<Integer>(factory) {
      public Integer atomically() {
        if(write) cups.swap(20);
        return cups.get();
      }
    }.execute();
  }
```

To programmatically set the transaction configuration we need a TransactionFactory. The TransactionFactoryBuilder provides convenience methods to create the factory. We created an instance of TransactionFactoryBuilder to configure the TransactionFactory with the readonly option using the setReadonly() method. The TransactionFactoryBuilder implements the Cascade[3] design pattern, so you can chain more methods for other properties you want to set before calling the build() method. Send this factory instance as parameter to the constructor of Atomic and you're guaranteed that no action within the transaction can change any managed references.

So our transaction is readonly, but let's see what happens if we try to modify a reference. So, we'll call the readWriteCups() once to only read the reference and then a second time to change it.

Download usingTransactionalMemory/java/configure/CoffeePot.java

```
  public static void main(final String[] args) {
    System.out.println("Read only");
    readWriteCups(false);

    System.out.println("Attempt to write");
    try {
      readWriteCups(true);
    } catch(Exception ex) {
      System.out.println("Failed " + ex);
    }
  }
}
```

The transaction won't be happy about the change request and on the attempt to change will throw org.multiverse.api.exceptions.ReadonlyException exception and the transaction will rollback.

```
Read only
Attempt to write
Failed org.multiverse.api.exceptions.ReadonlyException:
Can't open for write transactional object 'akka.stm.Ref@1272670619'
because transaction 'DefaultTransaction' is readonly'
```

3. Some of the patterns discussed in *Smalltalk Best Practice Patterns* [Bec96] by Kent Beck are being rediscovered, especially with the rise of new languages on the JVM.

The runtime exception was raised from the call to swap(). This method modifies the value held by its ref only if the new value is different from the current value, otherwise it ignores the change request. So, in the example, instead of 20 if you set the value to 24, the current value of the cups ref, you won't get any exception.

Configuring Transactions in Scala

In Scala, you use the atomic() method instead of the Atomic class for creating transactions. As you'd guess, this method takes an optional parameter of type TransactionFactory. Creating the instance of the factory is much easier in Scala as well, because you can use the factory method on the companion object.

Download usingTransactionalMemory/scala/configure/CoffeePot.scala

```
object CoffeePot {
  val cups = new Ref(24)

  def readWriteCups(write : Boolean) = {
    val factory = TransactionFactory(readonly = true)

    atomic(factory) {
        if(write) cups.swap(20)
        cups.get()
    }
  }

  def main(args : Array[String]) : Unit = {
    println("Read only")
    readWriteCups(false)

    println("Attempt to write")
    try {
      readWriteCups(true)
    } catch {
      case ex => println("Failed " + ex)
    }
  }
}
```

Other than Scala and Akka conciseness, there's not much different between the Scala and the Java version here and the code should behave just like the Java version.

```
Read only
Attempt to write
Failed org.multiverse.api.exceptions.ReadonlyException:
Can't open for write transactional object 'akka.stm.Ref@1761506447'
because transaction 'DefaultTransaction' is readonly'
```

6.11 Blocking Transactions—Sensible Wait

Suppose you have multiple competing tasks and you expect that the failure of a certain logical condition is temporary. You may return a failure code and ask the requesting code to reissue the request after a delay. However, there's no point repeating until another task has changed the data you're expecting to be different. Akka gives you a simple facility, retry(), which will rollback and block your transaction until one of the reference objects your transaction depends on changes or the configurable block timeout is exceeded. I like to call this a "sensible wait" as that sounds better than the word blocking. Let's put blocking, I mean sensible wait, to use in an example in Java first and then in Scala.

Blocking Transactions in Java

Programmers in general are addicted to caffeine, and so your colleague who volunteered to get some coffee knows not to return with empty cups. Rather than spin wait till the coffee pot is refilled, he can place himself in a notify-when-change mode with Akka's help. Let's write a fillCup() method with sensible wait using retry().

Download usingTransactionalMemory/java/blocking/CoffeePot.java

```java
public class CoffeePot {
  private static final long start = System.nanoTime();
  private static final Ref<Integer> cups = new Ref<Integer>(24);

  private static void fillCup(final int numberOfCups) {
    TransactionFactory factory =
      new TransactionFactoryBuilder()
      .setBlockingAllowed(true)
      .setTimeout(new DurationInt(6).seconds())
      .build();

    new Atomic<Object>(factory) {
      public Object atomically() {
        if(cups.get() < numberOfCups) {
          System.out.println("retry........ at " +
            (System.nanoTime() - start)/1.0e9);
          akka.stm.StmUtil$class.retry(null);
        }
        cups.swap(cups.get() - numberOfCups);
        System.out.println("filled up...." + numberOfCups);
        System.out.println("........ at " + (System.nanoTime() - start)/1.0e9);
        return null;
      }
    }.execute();
  }
```

In the fillCup() method, we configured the transaction with blockingAllowed = true and set a timeout of six seconds for the transaction to complete. When the fillCup() method finds there's not enough coffee, rather than returning an error code, it invokes the StmUtil's retry() method. This blocks the current transaction until the participating ref cups is changed. Once any participating ref changes, the atomically code that contained the retry is retried, starting another transaction.

The call to retry() looks weird, but that's because StmUnit$class is an abstract class with static methods for the methods defined in the Scala trait StmUnit—Akka was written in Scala and you ran into a Java-Scala interoperability quirk here.

Let's call the fillCup() method to see the effect of retry().

Download usingTransactionalMemory/java/blocking/CoffeePot.java

```
  public static void main(final String[] args) {
    Timer timer = new Timer(true);
    timer.schedule(new TimerTask() {
      public void run() {
        System.out.println("Refilling.... at " + (System.nanoTime() - start)/1.0e9);
        cups.swap(24);
      }
    }, 5000);

    fillCup(20);
    fillCup(10);
    try {
      fillCup(22);
    } catch(Exception ex) {
      System.out.println("Failed: " + ex.getMessage());
    }
  }
}
```

In main(), we start a timer which will refill the coffeepot in about 5 seconds. The guy who jumped in first got 20 cups of coffee right away. When your colleague goes for the 10 cups he's blocked, a sensible wait instead of spin wait, for the refill to happen. Once the refill transaction completes, his request is automatically tried again and this time succeeds. If the refill does not happen in the timeout you configured, the transaction will fail as in the last request in the **try** block. You can observe this behavior and the benefit of retry() in the output.

```
filled up....20
........ at 0.423589
```

```
retry........ at 0.425385
retry........ at 0.427569
Refilling.... at 5.130381
filled up....10
........ at 5.131149
retry........ at 5.131357
retry........ at 5.131521
Failed: Transaction DefaultTransaction has timed with a
total timeout of 6000000000 ns
```

The fill up request for 10 cups was made in 0.40 seconds from the start of the run. Since the refill is not going to happen until after 5 seconds from the start, this request is blocked due to the call to retry(). Right after the refill transaction completed, the fill up transaction was restarted and this time ran to completion a little after 5 seconds from the start. The later request to fill up timed out since no refill happened after this request.

It is not often that you'd use retry(), but if you find yourself retrying method calls after an arbitrary delay, you'll benefit from this feature that monitors change to the data your transaction depends on.

Blocking Transactions in Scala

The Scala version can directly invoke the retry() method of the Stm trait quite easily compared to the weird syntax you saw in Java. You can also make use of the factory method to create the TransactionFactory.

Download usingTransactionalMemory/scala/blocking/CoffeePot.scala

```scala
object CoffeePot {
  val start = System.nanoTime()
  val cups = new Ref(24)

  def fillCup(numberOfCups : Int) = {
    val factory = TransactionFactory(blockingAllowed = true,
      timeout = 6 seconds)

    atomic(factory) {
      if(cups.get() < numberOfCups) {
        println("retry........ at " + (System.nanoTime() - start)/1.0e9)
        retry()
      }
      cups.swap(cups.get() - numberOfCups)
      println("filled up...." + numberOfCups)
      println("........ at " + (System.nanoTime() - start)/1.0e9)
    }
  }

  def main(args : Array[String]) : Unit = {
    val timer = new Timer(true)
    timer.schedule(new TimerTask() {
      def run() {
        println("Refilling.... at " + (System.nanoTime() - start)/1.0e9)
        cups.swap(24)
      }
    }, 5000)

    fillCup(20)
    fillCup(10)
    try {
      fillCup(22)
    } catch {
      case ex => println("Failed: " + ex.getMessage())
    }
  }
}
```

When creating the TransactionFactory, to configure the timeout rather than directly using the DurationInt, you use the implicit conversion from int using the intToDurationInt() method. This allowed you to simply invoke 6 seconds, a little syntactic sugar provided by Scala implicit conversions. The rest of the code is a simple translation from Java to Scala and the output is shown next.

```
filled up....20
........ at 0.325964
retry........ at 0.327425
retry........ at 0.329587
Refilling.... at 5.105191
filled up....10
........ at 5.106074
retry........ at 5.106296
retry........ at 5.106466
Failed: Transaction DefaultTransaction has timed with a
total timeout of 6000000000 ns
```

6.12 Commit and Rollback Events

Java's try-catch-finally facility allows you to handle exceptions and selectively run some code only when there's an exception. Similarly you can decide to run a piece of code only if a transaction committed and another piece of code only if a transaction rolled back. These are provided as deferred() and compensating() methods, respectively, on StmUtil. The deferred() method is a great place to perform all the side-effects that you were holding off to ensure the transaction completes. It's going to take just a little more effort to implement this in Java than in Scala as we'll see next.

Commit and Rollback Events in Java

StmUtil is a Scala trait. If you view it through Java eyes, you'll see an interface named StmUtil and an abstract class named StmUtil$class. It's just a bit more work to use this in Java, but nothing for a proficient Java programmer to flinch at. The deferred() and compensating() methods accept a block of code that should be run within the context of the transaction, the atomically(). So, you have to implement the interface StmUtil in your Atomic class. It's simpler to create this as a standalone abstract class—less clutter, more reusability. So, let's extend an EventHandlingAtomic from Atomic to implement the StmUtil interface.

Download usingTransactionalMemory/java/events/EventHandlingAtomic.java

```java
public  abstract class EventHandlingAtomic<T> extends Atomic<T> implements StmUtil {

  public void deferred(Function0 fn) { StmUtil$class.deferred(this, fn); }

  public void compensating(Function0 fn) { StmUtil$class.compensating(this, fn); }

  public void retry() { StmUtil$class.retry(this); }

  public Object either(Function0 fn) { return StmUtil$class.either(this, fn); }
}
```

The EventHandlingAtomic is rather basic. It simply acts as a delegate. In this class, we receive calls to the four methods of the StmUtil interface and route them to the implementation in the abstract class StmUtil$class. This is something internally taken care of in Scala when we use traits, here we're making up for the lack of traits in Java.

Methods like deferred() accept *function values* or *closures*, which are first class citizens in modern JVM languages like Scala. Since Java itself does not yet support closures, we have to find a way to create and pass them. Scala APIs that accept closures as parameters expose a special interface named Function0, Function1, and so on, depending on the number of parameters accepted by the closures. To create a closure in Java that the Scala API will accept, we will implement one of these interfaces like Function0. Alternately, we can extend from a corresponding abstract class like AbstractFunction0.

The code you want to run when the transaction succeeds should be placed in a closure or code block that's passed to the deferred() method. Likewise, compensating() will take care of code to run upon failure of a transaction. These methods have to run in the context of a transaction, so you'll have to place these within the body of atomically() method.

Download usingTransactionalMemory/java/events/Counter.java

```
public class Counter {
  private final Ref<Integer> value = new Ref<Integer>(1);

  public void decrement() {
    new EventHandlingAtomic<Integer>() {
      public Integer atomically() {

        deferred(new scala.runtime.AbstractFunction0() {
          public Object apply() {
            System.out.println("Transaction completed...send email, log, etc.");
            return null;
          }
        });

        compensating(new scala.runtime.AbstractFunction0() {
          public Object apply() {
            System.out.println("Transaction aborted...hold the phone");
            return null;
          }
        });

        if(value.get() <= 0) throw new RuntimeException("Operation not allowed");

        value.swap(value.get() - 1);
        return value.get();
      }
    }.execute();
  }
}
```

The Counter class has one instance method named decrement(). In this method we extend our abstract class EventHandlingAtomic and implement the required atomically() method. In previous examples we simply wrote the code to be run in a transaction here. In addition to that, we also provide code to run in case of transaction success and transaction rollback. Let's create a sample code to try out Counter.

Download usingTransactionalMemory/java/events/UseCounter.java

```
public class UseCounter {
  public static void main(String[] args) {
    Counter counter = new Counter();
    counter.decrement();

    System.out.println("Let's try again...");
    try {
      counter.decrement();
    } catch(Exception ex) {
      System.out.println(ex.getMessage());
    }
  }
}
```

Exercise the UseCounter to see the code blocks associated with transaction success and failure run as appropriate.

```
Transaction aborted...hold the phone
Transaction completed...send email, log, etc.
Let's try again...
Transaction aborted...hold the phone
Operation not allowed
```

When the transaction completed on the first call to decrement(), the code block provided to the deferred() method is invoked. In the second call to decrement(), as soon as we hit the exception, the transaction was rolled back and the code block provided to the compensating() method was called. You also see the rollback due to an unexpected retry at the top that's due to the speculative configuration we discussed earlier.

The deferred() handler is a great place to complete your work activity, to make your actions permanent. So, feel free to print, display messages, send notifications, commit the database transactions, and so on, from here. If you want anything to stick around, this is the place for that. The compensating() handler is a good place for logging failures. If you had messed with any unmanaged objects (those objects not controlled using the Akka Ref), this is the place to revert your actions—however it is better to avoid the desire for such design as it's messy and error prone.

Commit and Rollback Events in Scala

You can handle commit and rollback events in Scala just like you did in Java. You won't need the EventHandlingAtomic, since traits are supported in Scala. You also won't need special handling of closures, because they are part of Scala as well. Let's translate the Counter class from Java to Scala.

Download usingTransactionalMemory/scala/events/Counter.scala

```
class Counter {
  private val value = new Ref(1)

  def decrement() = {
    atomic {
      deferred { println("Transaction completed...send email, log, etc.") }

      compensating { println("Transaction aborted...hold the phone") }

      if(value.get() <= 0) throw new RuntimeException("Operation not allowed")

      value.swap(value.get() - 1)
      value.get()
    }
  }
}
```

The code to run when a transaction succeeds is placed in the closure that's passed to the deferred() method. Likewise, the code to run when a transaction rolls back is presented to the compensating() method as a closure. These two are placed within the closure presented to the atomic() method, along with the transactional code. Again, quite simple and concise. Let's translate the UseCounter class from Java to Scala.

Download usingTransactionalMemory/scala/events/UseCounter.scala

```
object UseCounter {
  def main(args : Array[String]) : Unit = {
    val counter = new Counter()
    counter.decrement()

    println("Let's try again...")
    try {
      counter.decrement()
    } catch {
      case ex => println(ex.getMessage())
    }
  }
}
```

The output from the Scala version should be the same as the Java version.

```
Transaction aborted...hold the phone
Transaction completed...send email, log, etc.
Let's try again...
Transaction aborted...hold the phone
Operation not allowed
```

6.13 Collections and Transactions

As you work though the examples, it's easy to forget that the values you're dealing with are, and should be, immutable. Only the identity is mutable, not the state value. While STM makes life easy, it can be a challenge to ensure good performance while maintaining immutability.

The first step is to ensure immutability. Make your value classes **final** and mark all their fields final (vals in Scala). Then transitively ensure that the classes for each field your values use are immutable as well. It should be first nature to make fields and classes final—this is the first step towards avoiding concurrency issues.

While immutability will make your code better and safer, one prominent reason for reluctance of many programmers to use it is performance. If nothing changes, we'll have to make copies. We discussed persistent data structures and how they can ease performance concerns in Section 3.6, *Persistent/Immutable Data Structures*, on page 54. You can make use of persistent data structures already provided in third party libraries or those you can find in Scala. You don't have to switch languages to take advantage of this. You can use those persistent data structures right from your Java code.

Not only do you want immutability, you also want your data structures to participate in transactions—their values are immutable, but identities change within managed transactions. Akka provides two managed data structures—TransactionalVector and TransactionalMap. These work like Java lists and dictionaries but derive from efficient Scala data structures. Let's take a look at how to use TransactionalMap in Java and Scala.

Using Transactional Collections in Java

Using TransactionalMap from Java is quite simple. Suppose you've been asked to keep scores for different players and that the updates to these scores will arrive concurrently. Rather than deal with synchronization and locks, you decide to handle the updates within transactions. Here's an example code for that.

Download usingTransactionalMemory/java/collections/Scores.java

```
public class Scores {
  final private TransactionalMap<String, Integer> scoreValues =
    new TransactionalMap<String, Integer>();
  final private Ref<Long> updates = new Ref<Long>(0L);

  public void updateScore(final String name, final int score) {
    new Atomic() {
      public Object atomically() {
        scoreValues.put(name, score);
        updates.swap(updates.get() + 1);
        if (score == 13) throw new RuntimeException("Reject this score");
        return null;
      }
    }.execute();
  }

  public Iterable<String> getNames() {
    return scala.collection.JavaConversions.asJavaIterable(scoreValues.keySet());
  }

  public long getNumberOfUpdates() { return updates.get(); }

  public long getScore(final String name) { return scoreValues.get(name).get(); }
}
```

In the updateScore() method, you set the score value for a player and increment the update count in a transaction. Both the fields scoreValue of type TransactionalMap and updates of type Ref are managed. The TransactionalMap supports methods you'd expect on a Map, but these methods are transactional—any changes you make to them are discarded if the transaction is rolled back. To see this effect in action, you roll back the transaction after making the change if the score value is 13.

In Java you can use the for-each statement, like for(String name : collectionOfNames), if the collection implements the Iterable interface. The TransactionalMap is a Scala collection and does not directly support that interface. No worries, Scala provides a façade JavaConversions that provides convenience methods to get favorable Java interfaces. You're using its asJavaIterable() method to get the interface you desire in the getNames() method.

The Scores class is ready, we now need a class to exercise its methods.

Download usingTransactionalMemory/java/collections/UseScores.java

```
public class UseScores {
  public static void main(final String[] args) {
    final Scores scores = new Scores();

    scores.updateScore("Joe", 14);
    scores.updateScore("Sally", 15);
    scores.updateScore("Bernie", 12);

    System.out.println("Number of updates: " + scores.getNumberOfUpdates());

    try {
      scores.updateScore("Bill", 13);
    } catch(Exception ex) {
      System.out.println("update failed for score 13");
    }

    System.out.println("Number of updates: " + scores.getNumberOfUpdates());

    for(String name : scores.getNames()) {
      System.out.println(
        String.format("Score for %s is %d", name, scores.getScore(name)));
    }

  }
}
```

We first add scores for three players. Then we add another score value that will result in the transaction rollback. This last score update should have no effect. Finally we iterate over the scores in the transactional map. Let's observe the output of this code.

```
Number of updates: 3
update failed for score 13
Number of updates: 3
Score for Joe is 14
Score for Bernie is 12
Score for Sally is 15
```

Using Transactional Collections in Scala

You can use the transactional collections from within Scala much like you did in Java. Since in Scala you'd use Scala's internal iterators, you don't have to use the JavaConversions façade. Let's translate the Scores class to Scala.

Download usingTransactionalMemory/scala/collections/Scores.scala

```
class Scores {
  private val scoreValues = new TransactionalMap[String, Int]()
  private val updates = new Ref(0L)

  def updateScore(name : String, score : Int) = {
    atomic {
      scoreValues.put(name, score)
      updates.swap(updates.get() + 1)
      if (score == 13) throw new RuntimeException("Reject this score")
    }
  }

  def foreach(codeBlock : ((String, Int)) => Unit) = scoreValues.foreach(codeBlock)

  def getNumberOfUpdates() = updates.get()
}
```

The updateScore() method is pretty equivalent to the Java version. We eliminated the getNames() and getScore methods and instead provided a foreach() internal-iterator to step through the score values. The Scala version of the UseScores should be a pretty straightforward translation from the Java version.

Download usingTransactionalMemory/scala/collections/UseScores.scala

```
object UseScores {
  def main(args : Array[String]) : Unit = {
    val scores = new Scores()

    scores.updateScore("Joe", 14)
    scores.updateScore("Sally", 15)
    scores.updateScore("Bernie", 12)

    println("Number of updates: " + scores.getNumberOfUpdates())

    try {
      scores.updateScore("Bill", 13)
    } catch {
      case _ => println("update failed for score 13")
    }

    println("Number of updates: " + scores.getNumberOfUpdates())

    scores.foreach { mapEntry =>
      val (name, score) = mapEntry
      println("Score for " + name + " is " + score)
    }
  }
}
```

The output of this version should be the same as the Java version as you'd expect.

```
Number of updates: 3
update failed for score 13
Number of updates: 3
Score for Joe is 14
Score for Bernie is 12
Score for Sally is 15
```

6.14 Dealing with Write Skew Anomaly

In Section 6.6, *Handling Write Skew Anomaly*, on page 109 we discussed write skew and how Clojure STM handles it. Akka also has support for dealing with write skew, but you have to configure it. OK, that word may sound scary, but it's real simple. Let's first see the default behavior without any configuration.

Create a Portfolio class that holds a checking account balance and a savings account balance, and suppose these two have a constraint of the total balance not running below $1000. This class along with the withdraw() method is shown next. In this method, you obtain the two balances first, compute their total, and after a intentional delay (introduced to set transactions on collision course), you subtract the given amount from either the checking or the savings balance if the total is not below $1000. The withdraw() method does its operations within a transaction configured using default settings.

Download usingTransactionalMemory/java/writeSkew/Portfolio.java

```java
public class Portfolio {
  final private Ref<Integer> checkingBalance = new Ref<Integer>(500);
  final private Ref<Integer> savingsBalance = new Ref<Integer>(600);

  public int getCheckingBalance() { return checkingBalance.get(); }
  public int getSavingsBalance() { return savingsBalance.get(); }

  public void withdraw(final boolean fromChecking, final int amount) {
    new Atomic<Object>() {
      public Object atomically() {
        int totalBalance = checkingBalance.get() + savingsBalance.get();
        try { Thread.sleep(1000); } catch(InterruptedException ex) {}
        if(totalBalance - amount >= 1000) {
          if(fromChecking)
            checkingBalance.swap(checkingBalance.get() - amount);
          else
            savingsBalance.swap(savingsBalance.get() - amount);
        }
        else
          System.out.println(
            "Sorry, can't withdraw due to constraint violation");
        return null;
      }
    }.execute();
  }
}
```

Let's set two transactions in concurrent motion to change the balances.

Download usingTransactionalMemory/java/writeSkew/UsePortfolio.java

```java
public class UsePortfolio {
  public static void main(String[] args) throws InterruptedException {
    final Portfolio portfolio = new Portfolio();

    int checkingBalance = portfolio.getCheckingBalance();
```

```
        int savingBalance = portfolio.getSavingsBalance();
        System.out.println("Checking balance is " + checkingBalance);
        System.out.println("Savings balance is " + savingBalance);
        System.out.println("Total balance is " + (checkingBalance + savingBalance));

        final ExecutorService service = Executors.newFixedThreadPool(10);
        service.execute(new Runnable() {
          public void run() { portfolio.withdraw(true, 100); }
        });
        service.execute(new Runnable() {
          public void run() { portfolio.withdraw(false, 100); }
        });

        service.shutdown();

        Thread.sleep(4000);

        checkingBalance = portfolio.getCheckingBalance();
        savingBalance = portfolio.getSavingsBalance();
        System.out.println("Checking balance is " + checkingBalance);
        System.out.println("Savings balance is " + savingBalance);
        System.out.println("Total balance is " + (checkingBalance + savingBalance));
        if(checkingBalance + savingBalance < 1000)
          System.out.println("Oops, broke the constraint!");
      }
    }
```

By default Akka does not avoid write skew, and the two transactions will proceed running your balances into constraint violation as you see in the output.

```
Checking balance is 500
Savings balance is 600
Total balance is 1100
Checking balance is 400
Savings balance is 500
Total balance is 900
Oops, broke the constraint!
```

It's time to fix this. Let's reach out to the TransactionFactory that will help us configure transactions programmatically. Modify line 9 in the Portfolio class to accept an instance of the factory. That is, change

```
new Atomic<Object>() {
```

to

```
akka.stm.TransactionFactory factory =
  new akka.stm.TransactionFactoryBuilder()
  .setWriteSkew(false)
  .setTrackReads(true)
  .build();
```

```
new Atomic<Object>(factory) {
```

We created a TransactionFactoryBuilder and set the writeSkew and trackReads properties to false and true, respectively. This tells the transaction to keep track of reads within a transaction and also to set a read lock on the reads until the commit begins, just as Clojure STM handles ensure.

The rest of the code in Portfolio and the code in UsePortfolio remains unchanged. Set the transactions in motion and watch the output.

```
Checking balance is 500
Savings balance is 600
Total balance is 1100
Sorry, can't withdraw due to constraint violation
Checking balance is 400
Savings balance is 600
Total balance is 1000
```

Due to the non-deterministic nature of concurrent execution, you can't predict which of the two transactions will win. You can see the difference between the output here, with the ending balances of the two accounts being different, and the output in Section 6.6, *Handling Write Skew Anomaly*, on page 109, where the ending balance was equal. You may notice differences in both examples when you run them several times.

The previous example was in Java, if you're using Scala you can configure the transaction properties writeSkew and trackReads using the syntax you saw in Section 6.10, *Configuring Transactions in Scala*, on page 131.

6.15 Limitations of STM

STM eliminates explicit synchronization. You no longer have to worry if you forgot to synchronize or if you synchronized at the wrong level. There are no issues of failing to cross the memory barrier or race conditions. I can hear the shrewd programmer in you asking "what's the catch?" Yes, STM has limitations—otherwise this book would've ended right here. It's suitable only when write collisions are infrequent. If your application has a lot of write contention you should look beyond STM.

Let's discuss this limitation further. STM provides a lock-free programming model. It allows transactions to run concurrently, and they all complete without a glitch where there are no conflicts between them.

This provides for greater concurrency and thread safety at the same time. When transactions collide on write access to the same object or data, one of them is allowed to complete and the others are automatically retried. The retries delay the execution of the colliding writers but provide for greater speed for readers and the winning writer. The performance takes little hit when you have infrequent concurrent writers to the same object. As the collisions increase, things get worse, however.

If you have a high rate of write collision to the same data, in the best case your writes are slow. In the worst case, your writes may fail due to too many retries. The examples you saw so far in this chapter showed how easy it is to use STM and how you can benefit from it. While STM is quite easy, not all uses will yield good results as you'll see in the next example.

In Section 4.2, *Coordination using CountDownLatch*, on page 70, we used AtomicLong to synchronize concurrent updates to the total file size when multiple threads explored directories. In that code, we used an active **while** loop to manage attempts to change the shared mutable variable. Furthermore, we would resort to using synchronization if we had more than one variable to change at the same time. It looks like a nice candidate for STM, but the high contentions don't favor it. Let's see if that's true by modifying the file size program to use STM.

Instead of using AtomicLong we'll use Akka managed references for the fields in our file size finder.

Download usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```java
public class FileSizeWSTM {

  private ExecutorService service;
  final private Ref<Long> pendingFileVisits = new Ref<Long>(0L);
  final private Ref<Long> totalSize = new Ref<Long>(0L);
  final private CountDownLatch latch = new CountDownLatch(1);
```

The pendingFileVisits needs to be incremented or decremented within the safe haven of a transaction. In the case of AtomicLong, we used a simple incrementAndGet() or decrementAndGet(). However, since the managed reference is generic and does not specifically deal with numbers, we have to put some more effort. It's easier if we isolate that into a separate method.

Download usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```java
private long incrementOrDecrementPendingFileVisits(final boolean increment) {
  return new Atomic<Long>() {
    public Long atomically() {
      if(increment)
        pendingFileVisits.swap(pendingFileVisits.get() + 1);
      else
        pendingFileVisits.swap(pendingFileVisits.get() - 1);

      return pendingFileVisits.get();
    }
  }.execute();
}
```

The method to explore directories and find file sizes should now be pretty easy to implement. It's a simple conversion from using the AtomicLong to using the managed references.

Download usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
private void findTotalSizeOfFilesInDir(final File file) {
  try {
    if (!file.isDirectory()) {
      new Atomic() {
        public Object atomically() {
          totalSize.swap(totalSize.get() + file.length());
          return null;
        }
      }.execute();
    } else {
      final File[] children = file.listFiles();

      if (children != null) {
        for(final File child : children) {
          incrementOrDecrementPendingFileVisits(true);
          service.execute(new Runnable() {
            public void run() {
              findTotalSizeOfFilesInDir(child); }
          });
        }
      }
    }

    if(incrementOrDecrementPendingFileVisits(false) == 0) latch.countDown();
  } catch(Exception ex) {
    System.out.println(ex.getMessage());
    System.exit(0);
  }
}
```

Finally we need code to create the executor service pool and get it running.

Download usingTransactionalMemory/java/filesize/FileSizeWSTM.java

```
  private long getTotalSizeOfFile(final String fileName)
    throws InterruptedException {
    service  = Executors.newFixedThreadPool(100);
    incrementOrDecrementPendingFileVisits(true);
    try {
     findTotalSizeOfFilesInDir(new File(fileName));
     latch.await(100, TimeUnit.SECONDS);
     return totalSize.get();
    } finally {
      service.shutdown();
    }
  }

  public static void main(final String[] args) throws InterruptedException {
    final long start = System.nanoTime();
    final long total = new FileSizeWSTM().getTotalSizeOfFile(args[0]);
    final long end = System.nanoTime();
    System.out.println("Total Size: " + total);
    System.out.println("Time taken: " + (end - start)/1.0e9);
  }
}
```

I suspect this code will run into trouble, so at the sign of an exception indicating failure of a transaction we terminate the application.

If a value changes before the transaction is committed, the transaction will be retried automatically. Several threads compete to modify these two mutable variables, and your results may vary between slow running code to outright failure. Go ahead and run the code to explore different directories. I report here the output on my system for the /etc and /usr directories.

```
Total file size for /etc
Total Size: 2266408
Time taken: 0.537082

Total file size for /usr
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
Too many retries on transaction 'DefaultTransaction', maxRetries = 1000
...
```

The STM version gave the same file size for the /etc directory as the earlier version that used AtomicLong. However, the STM version was much slower, by about an order of magnitude, due to several retries. Exploring the /usr directory turned out to be much worse—quite a few transactions exceeded the default maximum retries limit. Even though we asked the application to be terminated, since several transactions are running concurrently, you may notice more failures before the first failure gets a chance to terminate the application.

The file size program has a high frequency of write conflicts as several threads try to update the total size. So, STM is not suitable for this problem. STM will serve you well and removes the need to synchronize when you have highly frequent reads and very infrequent write conflicts. If your problem has high frequent write collisions don't use STM. Instead, to avoid synchronization, use actors.

6.16 Recap

STM is a very powerful model that eliminates the need for explicit synchronization. It provides a nice lock-free programming model with good thread-safety and high concurrent performance. STM is suitable for concurrent reads and highly infrequent write collisions to the same data. This is quite an effective way to deal with shared mutability if your application data access fits that pattern. If you have greater write collisions, you may want to lean towards the actor based model which we discuss in Chapter 8, *Favoring Isolated Mutability*, on page 172. Before that let's see how to use STM from different JVM languages in the next chapter.

A language that doesn't have everything is actually easier to program in than some that do

► Dennis Ritchie

Chapter 7

STM in Clojure, Groovy, Java, JRuby, and Scala

If you've ever tried to mix JVM languages, you've faced the challenges of integration. In addition to crossing library boundaries, you face the hurdles of crossing language boundaries. You pick a particular language for its strengths and capabilities. You certainly don't want the little annoyances to ruin your efforts.

If STM is the right choice for your application, the language you're using on the JVM shouldn't stop you from putting it to good use. Whether STM really is the right choice is decided by the data access patterns of your application and not the language you're using to code it. You can use STM from any language on the JVM.

You have a few options: You can use Clojure STM from your favorite language if you like. Alternately, you could use the Multiverse library directly or through the Akka library.

If you want to use Clojure STM, it couldn't be easier. The two key components used in Clojure STM, `ref` and `dosync`, are sugar-coated Clojure APIs for `clojure.lang.Ref` and `LockingTransaction`'s `runInTransaction()` method. So, when you define a `ref`, you're creating an instance of the `Ref` class, and when you invoke `dosync()`, you're calling the `runInTransaction()` method. Now you have direct access to the Clojure STM from Groovy, Java, JRuby, and any other JVM language.

If you want to use Multiverse, you can use its Java API, its Groovy API, or its JRuby integration API. Alternately, you can use it through the Akka library.

In this chapter, we'll see how to use STM in different JVM languages. Focus on the sections related to the languages you're interested in, and feel free to skip the others.

7.1 Clojure STM

Although STM has been around for a while, Clojure brought it to the limelight with its revolutionary separation of identity and state, as well as efficient persistent data structures. Since state is entirely immutable in Clojure, it makes it very easy to use STM—you don't have to worry about ensuring immutability, because that's already Clojure's modus-operandi. Clojure also disallows changes to mutable identity from outside of transactions, which makes it very safe. You can enjoy lock-free programming with good concurrent performance and thread-safety in applications with access pattern appropriate for STM. Refer to the examples earlier in Chapter 6, *Introduction to Software Transactional Memory*, on page 99, and the Clojure documentation and books if Clojure is in the mix of your language choices.

7.2 Groovy Integration

You can use Clojure STM from Groovy or you can use Multiverse STM directly or through Akka. In this section I will show you how to use Clojure STM and how to use Multiverse STM through Akka. If you want to use Multiverse directly, refer to the Multiverse documentation.

Using Clojure STM in Groovy

Using Clojure STM is as simple as using instances of clojure.lang.Ref for managed references and invoking the runInTransaction() method of LockingTransaction to place a piece of code in a transaction. Let's give it a shot using the account transfer example. Create a class Account to hold a managed reference to its immutable state, represented by the variable currentBalance.

Download polyglotSTM/groovy/clojure/Transfer.groovy

```
class Account {
  final private Ref currentBalance

  public Account(initialBalance) {
    currentBalance = new Ref(initialBalance)
  }
  def getBalance() { currentBalance.deref() }
```

In the constructor, we initialize the reference with the starting balance. In the getBalance() method, we call the deref() method. This is equivalent to the dereferencing you saw in Clojure using the @ prefix. Creating the managed reference was quite simple. Now let's see how you can update it in a transaction. To run a piece of code in a transaction, you have to send it to the runInTransaction() method. This method accepts as a parameter an instance that implements the java.util.concurrent.Callable interface. Groovy provides a rich syntax to implement interfaces, especially interfaces with only one method. Simply create a closure—a code block—and call the as operator on it. Under the covers Groovy will implement the required method of that interface with the provided code block as the implementation. Let's write the deposit() method of the Account class, which needs to run its operations in a transaction.

Download polyglotSTM/groovy/clojure/Transfer.groovy

```
def deposit(amount) {
  LockingTransaction.runInTransaction({
      if(amount > 0) {
        currentBalance.set(currentBalance.deref() + amount)
        println "deposit ${amount}... will it stay"
      } else {
        throw new RuntimeException("Operation invalid")
      }
  } as Callable)
}
```

The deposit modifies the current balance in a transaction if the amount is greater than 0. Otherwise, it causes the transaction to fail by throwing an exception. Implementing withdraw() is not much different. You have merely an extra check to perform.

Download polyglotSTM/groovy/clojure/Transfer.groovy

```
  def withdraw(amount) {
    LockingTransaction.runInTransaction({
      if(amount > 0 && currentBalance.deref() >= amount)
        currentBalance.set(currentBalance.deref() - amount)
      else
        throw new RuntimeException("Operation invalid")
    } as Callable)
  }
}
```

That's all the code in the Account class. Let's write a separate function that will perform the transfer, so we can see nested transactions in action. In the transfer() method, first make a deposit and then a withdrawal. This way you can see the effect of deposit discarded if the withdrawal fails.

Download polyglotSTM/groovy/clojure/Transfer.groovy

```
def transfer(from, to, amount) {
  LockingTransaction.runInTransaction({
    to.deposit(amount)
    from.withdraw(amount)
  } as Callable)
}
```

It's time to exercise all that code, so write a simple call sequence to transfer some money between the accounts. Let the first transfer be for an amount that will succeed. Then make a large transfer that will fail due to insufficient funds. This way you can see the effects of the transactions.

Download polyglotSTM/groovy/clojure/Transfer.groovy

```
def transferAndPrint(from, to, amount) {
  try {
    transfer(from, to, amount)
  } catch(Exception ex) {
    println "transfer failed $ex"
  }

  println "Balance of from account is $from.balance"
  println "Balance of to account is  $to.balance"
}

def account1 = new Account(2000)
def account2 = new Account(100)

transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

Run the code and study the output.

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is  600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is  600
```

The first transfer succeeded, and the balances displayed reflects that. The second transfer completed the deposit first, but the withdrawal failed. The total effect of the failed transaction was to leave the balance of the two accounts unaffected.

It's pretty straightforward to use STM transactions in Groovy. However, you must ensure that you are dealing with immutable state and only mutable managed references. Clojure is not there to protect you against changes to state. You must also ensure that the transactional code has no side effects.

Using Akka STM in Groovy

Using the Akka API in Groovy requires just a little bit more effort and some patience.

In Akka the transactional method atomic() is exposed through what's called a *package object*. Scala 2.8 package objects manifest themselves as a pair of classes in bytecode—the package and package$ classes. Groovy is not thrilled to see these lower case names and names with $ for classes. So you'll run into trouble if you try to use them directly.

There's a simple workaround, thankfully. Get a class reference to the package object using Class.forName(). The dynamic nature of Groovy will take you through the rest of the way quite easily. Simply invoke the atomic() method on the package reference and pass it a closure that implements Scala's closure represented by scala.Function0. You will also send a reference to the transaction factory. The atomic() method defines the second parameter, the transaction factory, as an implicit optional parameter. However, that's not recognized from Groovy and so you're required to send it. You can obtain the default factory, to use default configuration settings, from the package object or you can create your own instance of the factory with the configuration parameters you desire. Let's write the Account class to put all that to use.

Download polyglotSTM/groovy/akka/Transfer.groovy

```
class Account {
  final private Ref currentBalance
  private final stmPackage = Class.forName('akka.stm.package')

  public Account(initialBalance) { currentBalance = new Ref(initialBalance) }

  def getBalance() { currentBalance.get() }

  def deposit(amount) {
    stmPackage.atomic({
      if(amount > 0) {
        currentBalance.set(currentBalance.get() + amount)
        println "deposit ${amount}... will it stay"
      }
    } as scala.Function0, stmPackage.DefaultTransactionFactory())
  }

  def withdraw(amount) {
    stmPackage.atomic({
      if(amount > 0 && currentBalance.get() >= amount)
        currentBalance.set(currentBalance.get() - amount)
      else
        throw new RuntimeException("Operation invalid")
        } as scala.Function0, stmPackage.DefaultTransactionFactory())
  }
}
```

We pass the Java class name of the Scala package object to the forName() method and get a reference stmPackage to its Class meta-object. We then call the atomic() method of this class simply as stmPackage.atomic() and pass the two parameters, a closure and the stmPackage.DefaultTransactionFactory(). The code within our closure will run in a transaction managed by Akka/Multiverse.

The rest of the code for the transfer() method and running a pair of transfers is pretty straightforward.

Download polyglotSTM/groovy/akka/Transfer.groovy

```
def transfer(from, to, amount) {
  def stmPackage = Class.forName('akka.stm.package')
  stmPackage.atomic({
    to.deposit(amount)
    from.withdraw(amount)
    } as scala.Function0, stmPackage.DefaultTransactionFactory())
}

def transferAndPrint(from, to, amount) {
  try {
    transfer(from, to, amount)
  } catch(Exception ex) {
    println "transfer failed $ex"
  }

  println "Balance of from account is $from.balance"
  println "Balance of to account is  $to.balance"
}


def account1 = new Account(2000)
def account2 = new Account(100)

transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

Run the Groovy code and see the transactions in action.

```
deposit 500... will it stay
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is  600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is  600
```

7.3 Java Integration

You can use Clojure STM or Multiverse STM from within Java.

Using Clojure STM in Java

You can use Clojure STM for Java quite easily as the Ref and LockingTransaction are exposed as simple classes. The runInTransaction() method takes as a parameter an instance that implements the Callable interface. So, wrapping code in a transaction is as simple as wrapping it in the Callable interface's call() method.

We'll implement the account transfer example using Clojure STM in Java. First let's create a class Account that will hold a managed reference to its immutable state represented by the variable balance.

Download polyglotSTM/java/clojure/Account.java

```java
public class Account {
  final private Ref balance;

  public Account(int initialBalance) throws Exception {
    balance = new Ref(initialBalance);
  }

  public int getBalance() { return (Integer) balance.deref(); }
```

In the constructor, we initialize the reference with the starting balance. In the getBalance() method, we call the deref() method. This is equivalent to the dereferencing you saw in Clojure using the @ prefix. To run a piece of code in a transaction, you'd have to send it to the runInTransaction() method. This method accepts as a parameter an instance that implements the java.util.concurrent.Callable interface. Let's take a look at the deposit() method that needs to perform its actions in a transaction.

Download polyglotSTM/java/clojure/Account.java

```java
public void deposit(final int amount) throws Exception {
  LockingTransaction.runInTransaction(new Callable<Boolean>() {
    public Boolean call()  {
      if(amount > 0) {
        int currentBalance = (Integer) balance.deref();
        balance.set(currentBalance + amount);
        System.out.println("deposit " + amount + "... will it stay");
        return true;
      } else throw new RuntimeException("Operation invalid");
    }
  });
}
```

The deposit modifies the current balance in a transaction if the amount is greater than 0. Otherwise, it fails the transaction by throwing an exception. Implementing withdraw() is not much different, just an extra check to perform.

Download polyglotSTM/java/clojure/Account.java

```
  public void withdraw(final int amount) throws Exception {
    LockingTransaction.runInTransaction(new Callable<Boolean>() {
      public Boolean call() {
        int currentBalance = (Integer) balance.deref();
        if(amount > 0 && currentBalance >= amount) {
          balance.set(currentBalance - amount);
          return true;
        } else throw new RuntimeException("Operation invalid");
      }
    });
  }
}
```

That's all the code in the Account class. Let's write a separate class for transfer, so we can see nested transactions in action here. In the transfer() method first perform the deposit and then the withdrawal. This way you can see the effect of deposit discarded if the withdraw fails.

Download polyglotSTM/java/clojure/Transfer.java

```
public class Transfer {
  public static void transfer(
    final Account from, final Account to, final int amount) throws Exception {
    LockingTransaction.runInTransaction(new Callable<Boolean>() {
      public Boolean call() throws Exception {
        to.deposit(amount);
        from.withdraw(amount);
        return true;
      }
    });
  }
```

To exercise all that code write a simple call sequence to transfer some money between the accounts. Let the first transfer be an amount that will succeed. Then make a large transfer that will fail due to insufficient funds. This way we can see the effects of the transactions.

Download polyglotSTM/java/clojure/Transfer.java

```
public static void transferAndPrint(
  final Account from, final Account to, final int amount) {
  try {
    transfer(from, to, amount);
  } catch(Exception ex) {
    System.out.println("transfer failed " + ex);
  }

  System.out.println("Balance of from account is " + from.getBalance());
  System.out.println("Balance of to account is " + to.getBalance());
}
```

Let's create a main() to set these in motion.

Download polyglotSTM/java/clojure/Transfer.java

```
  public static void main(String[] args) throws Exception {
    Account account1 = new Account(2000);
    Account account2 = new Account(100);

    transferAndPrint(account1, account2, 500);
    transferAndPrint(account1, account2, 5000);
  }
}
```

Run the code and study the output.

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

The first transfer succeeded and the balances displayed reflect that. The second transfer completed the deposit first, but the withdrawal failed. The total effect of the failed transaction was to leave the balance of the two accounts unaffected.

It's pretty straightforward to use STM transactions in Java. However, you must ensure that you are dealing with immutable state and only mutable managed references. Clojure is not there to protect you against changes to state. You also need to ensure that the transactional method has no side effects.

Using Multiverse/Akka STM in Java

If you have no intention of using actors (see Chapter 8, *Favoring Isolated Mutability*, on page 172) and your only interest is in using STM, you may want to look at using Multiverse directly. If you plan to use actors or mix them with STM, you may consider using Akka. With Multiverse you can use Java based annotation syntax and APIs. Even though Akka was built on Scala, they've done a wonderful job of providing a Java API. Chapter 6, *Introduction to Software Transactional Memory*, on page 99 is filled with Akka examples in Java to help you get started if that's the route you plan to take. Remember to ensure the state is immutable and only identity is mutable.

7.4 JRuby Integration

You can use Clojure STM from JRuby or you can use Multiverse STM directly or through Akka. In this section I will show you how to use Clojure STM and how to use Multiverse STM through Akka. If you'd like to use Multiverse directly, refer to the Multiverse documentation.

Using Clojure STM in JRuby

Using Clojure STM is as simple as using instances of clojure.lang.Ref for managed references and invoking the runInTransaction() method of LockingTransaction to place a piece of code in a transaction. Let's give that a try using the account transfer example. Create a class Account that will hold a managed reference to its immutable state represented by the field @balance.

Download polyglotSTM/jruby/clojure/Account.rb

```
require 'java'
java_import 'clojure.lang.Ref'
java_import 'clojure.lang.LockingTransaction'

class Account
  def initialize(initialBalance)
    @balance = Ref.new(initialBalance)
  end

  def balance
    @balance.deref
  end
```

In the initialize() method, we initialize the reference with the starting balance. In the balance() method, we call the deref() method. This is equivalent to the dereferencing you saw in Clojure using the @ prefix. Creating the managed reference was quite simple. Let's see how you can update it in a transaction. To run a piece of code in a transaction, you have to send it to the runInTransaction() method. This method accepts as a parameter an instance that implements the java.util.concurrent.Callable interface. JRuby quietly will implement interfaces, especially interfaces with only one method, when you pass a closure. So, simply create a closure, a code block. Let's write the deposit() method of the Account class, which needs to run its operations in a transaction.

Download polyglotSTM/jruby/clojure/Account.rb

```ruby
def deposit(amount)
  LockingTransaction.run_in_transaction do
    if amount > 0
      @balance.set(@balance.deref + amount)
      puts "deposited $#{amount}... will it stay"
    else
      raise "Operation invalid"
    end
  end
end
```

The deposit modifies the current balance in a transaction if the amount is greater than 0. Otherwise, it fails the transaction by throwing an exception. Implementing withdraw() is not much different, you have just an extra check to perform.

Download polyglotSTM/jruby/clojure/Account.rb

```ruby
  def withdraw(amount)
    LockingTransaction.run_in_transaction do
      if amount > 0 && @balance.deref >= amount
        @balance.set(@balance.deref - amount)
      else
        raise "Operation invalid"
      end
    end
  end
end
```

That's all the code in the Account class. Let's write a separate function that will perform transfer, so we can see nested transactions in action here. In the transfer() method first perform the deposit and then the withdraw. This way you can see the effect of deposit discarded if the withdraw fails.

Download polyglotSTM/jruby/clojure/Account.rb

```ruby
def transfer(from, to, amount)
```

```
  LockingTransaction.run_in_transaction do
    to.deposit(amount)
    from.withdraw(amount)
  end
end
```

It's time to exercise all that code, so write a simple call sequence to transfer some money between the accounts. Let the first transfer be for an amount that will succeed. Then make a large transfer that will fail due to insufficient funds. This way you can see the effects of the transactions.

Download polyglotSTM/jruby/clojure/Account.rb

```
def transfer_and_print(from, to, amount)
  begin
    transfer(from, to, amount)
  rescue => ex
    puts "transfer failed #{ex}"
  end

  puts "Balance of from account is #{from.balance}"
  puts "Balance of to account is #{to.balance}"
end

account1 = Account.new(2000)
account2 = Account.new(100)

transfer_and_print(account1, account2, 500)
transfer_and_print(account1, account2, 5000)
```

Run the code and study the output.

```
deposited $500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposited $5000... will it stay
transfer failed Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

The first transfer succeeded and the balances displayed reflects that. The second transfer completed the deposit first, but the withdrawal failed. The total effect of the failed transaction was to leave the balance of the two accounts unaffected.

It's pretty straightforward to use STM transactions in JRuby. However, you must ensure that you are dealing with immutable state and only mutable managed references. Clojure is not there to protect you against changes to state. You must also ensure that your transactional methods have no side effects.

Using Akka STM in JRuby

Using the Multiverse STM in JRuby requires just a little bit more effort and some patience. The main reason is that Multiverse relies on exceptions to retry transactions, but JRuby wraps exceptions into NativeException—so if you're not careful, Multiverse will not see the exceptions it expects and your code will fail. You will first face this problem in the example, but we will figure out a way to workaround.

In Akka the transactional method atomic() is exposed through what's called a *package object*. Scala 2.8 package objects manifest themselves as a pair of classes in bytecode—the package and package$ classes. When you import the package class in JRuby, you will get a weird error cannot import class 'package' as 'package'. To workaround this error, redefine the package name into something other than package, like java_import 'akka.stm.package' do |pkgname, classname| "J#{classname}" end. Now when you refer to Jpackage you're referring to the Akka akka.stm.package package object.

We'll create the account transfer example using JRuby and Akka STM. Let's isolate the method that will use the Akka transaction into a separate module, so it's easier to reuse that code.

Download polyglotSTM/jruby/akka/broken/AkkaStm.rb

```
require 'java'
java_import 'akka.stm.Ref'
java_import 'akka.stm.package' do |pkgname, classname| "J#{classname}" end

module AkkaStm
  def atomic(&block)
    Jpackage.atomic Jpackage.DefaultTransactionFactory, &block
  end
end
```

We import Akka's Ref class and the package object akka.stm.package. In our AkkaStm module, we write a method atomic() that receives a block of code and passes it to the akka.stm.package's atomic() method. Since the API exposed outside of Scala for this method requires the factory as well, we obtain the default transaction factory from the package object itself. Let's use this method to place Account methods in transaction.

Download polyglotSTM/jruby/akka/Transfer.rb

```
require 'AkkaStm'
```

```
class Account
  include AkkaStm

  def initialize(initialBalance)
    @balance = Ref.new(initialBalance)
  end

  def balance
    @balance.get
  end

  def deposit(amount)
    atomic do
      if amount > 0
        @balance.set(@balance.get + amount)
        puts "deposited $#{amount}... will it stay"
      end
    end
  end

  def withdraw(amount)
    atomic do
      raise "Operation invalid" if amount < 0 || @balance.get < amount
      @balance.set(@balance.get - amount)
    end
  end
end
```

We create a managed reference for the @balance field and place the operations in the deposit() and withdraw() in transactions using our AkkaStm modules's atomic() method. We bring that method into our Account class using the **incude**—JRuby's *mixin* facility. We can write the transfer() method as a stand alone method and we can mixin the AkkaStm module into it as well to reuse the atomic() method.

Download polyglotSTM/jruby/akka/Transfer.rb

```
def transfer(from, to, amount)
  include AkkaStm
  atomic do
    to.deposit(amount)
    from.withdraw(amount)
  end
end
```

The final step is to exercise all this code by invoking a couple of transfers.

Download polyglotSTM/jruby/akka/Transfer.rb

```
def transfer_and_print(from, to, amount)
```

```
  begin
    transfer(from, to, amount)
  rescue => ex
    puts "transfer failed #{ex}"
  end

  puts "Balance of from account is #{from.balance}"
  puts "Balance of to account is #{to.balance}"
end

account1 = Account.new(2000)
account2 = Account.new(100)

transfer_and_print(account1, account2, 500)
transfer_and_print(account1, account2, 5000)
```

We can expect the first transfer to succeed since the amount is less than available funds. We'd want the second transfer to fail due to insufficient funds and leave the balance of the two accounts unaffected. Let's see what's in store when we run the code.

```
transfer failed
  org.multiverse.api.exceptions.SpeculativeConfigurationFailure: null
...
```

Not quite what we wanted to see. This error did more than fail the first transfer's transaction—in addition it cost me significant time and hair-loss. This is the error I alluded in the beginning of this section. The failure reported a SpeculativeConfigurationFailure exception which is part of org.multiverse.api.exceptions package. Let's understand where this came from and why things are failing suddenly in the JRuby version when the other language versions worked flawlessly.

By default Multiverse (and so Akka STM) uses speculative transactions, which we discussed in Section 6.9, *Nested Transactions in Java*, on page 121. At first Multiverse assumes the transaction is readonly and upon seeing the first set operation on a managed ref, it throws the SpeculativeConfigurationFailure exception. In the Multiverse layer, it handles this exception and retries the transaction, this time allowing for change to a managed ref. This is the reason you saw transactions being retried in Chapter 6, *Introduction to Software Transactional Memory*, on page 99. OK, that's Multiverse's business, and it seems to work fine so far, so you wonder what went wrong with JRuby.

If you replace

```
puts "transfer failed #{ex}"
```

with

```
puts "transfer failed #{ex.class}"
```

you'll see the exception class name is not SpeculativeConfigurationFailure but NativeException. This is JRuby placing the exception in its own wrapper exception. So, from the JRuby code, we called into Akka which in turn called into our JRuby closure. This closure tried to update the managed reference, and that caused the Multiverse exception like it's designed to do. Our JRuby closure code—unfortunately—internally wrapped that exception into a NativeException. So, instead of seeing the familiar SpeculativeConfigurationFailure, Multiverse sees this strange unexpected NativeException and simply fails the transaction and propagates the exception up the chain without retrying the transaction.

So, what's the fix? Not a pleasant one: we have to handle it explicitly. We need to check if the exception belongs to Multiverse and if so, unwrap it up the chain. It's an ugly workaround, but not too much code. Let's modify the AkkaStm's atomic() method to compensate for the JRuby behavior.

Download polyglotSTM/jruby/akka/AkkaStm.rb

```
require 'java'
java_import 'akka.stm.Ref'
java_import 'akka.stm.package' do |pkgname, classname| "J#{classname}" end

module AkkaStm
  def atomic(&block)
    begin
      Jpackage.atomic Jpackage.DefaultTransactionFactory, &block
    rescue NativeException => ex
      raise ex.cause if ex.cause.java_class.package.name.include? "org.multiverse"
    end
  end
end
```

We wrap the call to akka.stm.package.atomic around a begin-rescue block and if the exception thrown at us is a NativeException, and if the embedded real cause is a Multiverse exception, then we throw the unwrapped exception so Multiverse can handle it and take the appropriate action.

Now that we have fixed it, let's see if the two transactions behave as expected, the first one succeeding and the second one failing, leaving the balance unaffected.

```
deposited $500... will it stay
deposited $500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposited $5000... will it stay
transfer failed Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

After the fix, the JRuby version behaves just like the versions in other languages.

7.5 Choices in Scala

You again have at least three options for STM in Scala. You may use the Clojure STM from Scala. You may also use Multiverse STM from Scala. Alternately, you can use it through the Akka library. Let's explore using Clojure STM here.

Using Clojure STM in Scala

Using Clojure STM in Scala is going to follow along pretty much the lines of Java. You can use the Ref class and the runInTransaction() method of the LockingTransaction class. You create an instance that implements Callable to wrap the method that should run in a transaction. Let's implement the account transfer example using Scala and Clojure STM. It's pretty much a direct translation of the code from Java to Scala.

Download polyglotSTM/scala/clojure/Transfer.scala

```
class Account(val initialBalance : Int) {
  val balance = new Ref(initialBalance)

  def getBalance() = balance.deref
```

In the primary constructor, which is part of the class definition in Scala, we initialized the reference with the initial balance value. In the getBalance() method we called the deref() method. This is equivalent to the dereferencing you saw in Clojure using the @ prefix. Creating the managed reference was quite simple. To update the managed reference in a transaction we simply need to wrap it into a Callable and passed it to the runInTransaction() method. Let's write the deposit method whose operations need to run in a transaction.

Download polyglotSTM/scala/clojure/Transfer.scala

```
def deposit(amount : Int) = {
  LockingTransaction runInTransaction new Callable[Boolean] {
    def call() = {
      if(amount > 0) {
        val currentBalance = balance.deref.asInstanceOf[Int]
        balance.set(currentBalance + amount)
        println("deposit " + amount + "... will it stay")
        true
      } else throw new RuntimeException("Operation invalid")
    }
  }
}
```

The deposit modifies the current balance in a transaction if the amount is greater than 0. Otherwise, it fails the transaction by throwing an exception. Implementing withdraw() is not much different, just an extra check to perform.

Download polyglotSTM/scala/clojure/Transfer.scala

```
  def withdraw(amount : Int) = {
    LockingTransaction runInTransaction new Callable[Boolean] {
      def call() = {
        val currentBalance = balance.deref.asInstanceOf[Int]
        if(amount > 0 && currentBalance >= amount) {
          balance.set(currentBalance - amount)
          true
        } else throw new RuntimeException("Operation invalid")
      }
    }
  }
}
```

That's all the code in the Account class. We can write the transfer() method as a stand alone method, since we don't need a class for that in Scala if you run it as a script. If you plan to compile the code, then you'd have to wrap it into a object and provide main().

Download polyglotSTM/scala/clojure/Transfer.scala

```
def transfer(from : Account, to : Account, amount : Int) = {
  LockingTransaction runInTransaction new Callable[Boolean] {
    def call() = {
      to.deposit(amount)
      from.withdraw(amount)
      true
    }
  }
}
```

In the transfer() method, we first make the deposit and then the withdrawal. This way you can see the effect of the deposit discarded if the withdrawal fails. To exercise the code, write a simple call sequence to transfer some money between the accounts. Let the first transfer be for an amount that will succeed. Then make a large transfer that will fail due to insufficient funds. This way you can see the effects of the transactions.

Download polyglotSTM/scala/clojure/Transfer.scala

```
def transferAndPrint(from : Account, to : Account, amount : Int) = {
  try {
    transfer(from, to, amount)
  } catch {
    case ex => println("transfer failed " + ex)
  }

  println("Balance of from account is " + from.getBalance())
  println("Balance of to account is " + to.getBalance())
}

val account1 = new Account(2000)
val account2 = new Account(100)

transferAndPrint(account1, account2, 500)
transferAndPrint(account1, account2, 5000)
```

Run the code and study the output.

```
deposit 500... will it stay
Balance of from account is 1500
Balance of to account is 600
deposit 5000... will it stay
transfer failed java.lang.RuntimeException: Operation invalid
Balance of from account is 1500
Balance of to account is 600
```

The first transfer succeeded and the balances displayed reflects that. The second transfer completed the deposit first, but the withdrawal failed. The total effect of the failed transaction was to leave the balance of the two accounts unaffected.

It's pretty straightforward to use STM transactions in Scala. However, you must ensure that you are dealing with immutable state and only mutable managed references. Clojure is not there to protect you against changes to state. Also ensure that the transactional methods don't have any side effects.

Using Akka/Multiverse STM in Scala

Akka is certainly a good choice, because it was written in Scala and the APIs it exposes will feel natural to you. If you're programming in Scala, you'd most likely use Akka actors as well. Refer to the Scala examples in Chapter 6, *Introduction to Software Transactional Memory*, on page 99 to get started on using STM with Akka. Scala, being a hybrid functional programming language, allows you to create both mutable (**var**) variables and immutable (**val**) values. As good practice, it's better to promote immutability: use vals more than vars. Do a quick grep/search to see if you have vars and examine them closely. Ensure that the state is immutable and only the identity Refs are mutable.

7.6 Recap

For applications with frequent reads and infrequent write collisions you can make use of STM from any JVM language of your choice, as you learned in this chapter. You can enjoy the double bonus of the language features of your choosing and the benefits of STM. But when greater write collisions creep in, STM will not be suitable. In that case, to avoid synchronization nightmares, you can benefit from the actor based model we discuss in the next chapter.

Part IV

Actor-based Concurrency

There is always a better way.
► Thomas Alva Edison

Chapter 8

Favoring Isolated Mutability

"If it hurts, stop doing it" is doctors' good advice. In concurrent programming, shared mutability is "*it*."

With the JDK threading API it's easy to create threads but it soon becomes a struggle to prevent them from colliding and messing up. The STM eases that pain quite a bit, however, it poorly handles high frequency write collisions. Surprisingly, the struggles disappear when shared mutability disappears.

Letting multiple threads converge and collide on data is an approach we've tried in vain. Fortunately there's a better way—event-based message passing. In this approach you treat tasks as lightweight processes, internal to your application/JVM. Instead of letting them grab the data, you pass immutable messages to them. Once these asynchronous tasks complete they pass back or pass on their immutable results to other coordinating task(s). You design your application with coordinating actors[1] that asynchronously exchange immutable messages.

This approach has been around for a few decades but is relatively new in the JVM arena. Actor-based model is quite successful and popular in Erlang (see *Programming Erlang: Software for a Concurrent World* [Arm07] and *Concurrent Programming in Erlang* [VWWA96]). Erlang's actor based model was adopted and brought into the folds of the JVM when Scala was introduced in 2003 (see *Programming in Scala* [OSV08] and *Programming Scala* [Sub09]).

1. Recently someone asked what these *actors* have to do with actors in use cases—nothing. These actors *act* upon messages they receive, perform their dedicated tasks, and pass response messages for other actors... to act upon in turn.

In Java, you get to choose from over half-a-dozen libraries[2] that provide actor-based concurrency: ActorFoundary, Actorom, Actors Guild, Akka, FunctionalJava, Kilim, Jetlang,... Some of these libraries use aspect-oriented bytecode weaving. Each of them are at different levels of maturity and adoption.

In this chapter you'll learn how to program actor-based concurrency. For most part we'll use Akka as a vehicle to drive home the concepts. Akka is a high performing Scala based solution that exposes fairly good Java API. You can use it for both actor based concurrency and for STM (see Chapter 6, *Introduction to Software Transactional Memory*, on page 99).

8.1 Isolating Mutability using Actors

Java turned OOP into mutability driven development[3] while functional programming emphasizes immutability; both extremes are problematic. If everything is mutable, you have to tackle visibility and race conditions. In a realistic application, everything can't be immutable, even pure functional languages provide restricted areas of code that allow side-effects and ways to sequence them. Whichever programming model you favor, it's clear you want to avoid shared mutability.

Shared mutability—where multiple threads can modify a variable—is the root of concurrency problems. Isolated mutability—where only one thread (or actor) can access a mutable variable, ever—is a nice compromise that removes most concurrency concerns.

In OOP, you encapsulate so only the instance methods can manipulate the state of an object. However, different threads may call these methods, and that leads to concurrency concerns. In actor-based programming model, you allow only one actor to manipulate the state of an object. While your application is multithreaded, the actors themselves are single threaded, and so there are no visibility and race condition concerns. Actors request operations to be performed, but they don't reach over the mutable state managed by other actors.

You take a different design approach when programming with actors compared to programming merely with objects. You divide the problem into asynchronous computational tasks and assign them to different

2. Imagine how boring it would be if you had just one good solution to pick.
3. Java had other partners in this crime, so doesn't deserve a single-handed blame.

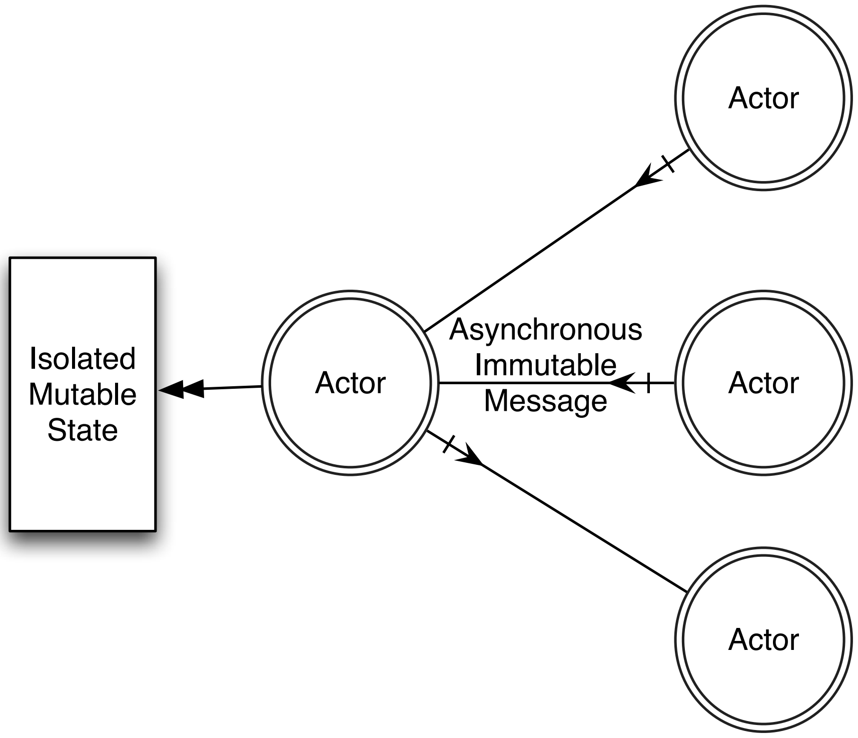


Figure 8.1: Actors isolate mutable state and communicate by passing immutable messages

actors. Each actor's focus is on performing its designated task. You confine any mutable state to within at most one actor, period (see Figure 8.1). You also ensure that the messages you pass between actors are totally immutable.

In this design approach, you let each actor work on a part of the problem. They receive the necessary data as immutable objects. Once they complete their assigned task, they send the results, as immutable objects, to the calling actor or another designated post processing actor. You can think of this as taking OOP to the next level where select objects are mutable and active—run in their own threads. The only way you're allowed to manipulate these objects is by sending messages to them and not by calling methods.

8.2 Actor Qualities

An actor is a free running activity that can receive messages, process requests, and send responses. Actors are designed to support asynchrony and efficient messaging.

Each actor has a built-in message queue much like the message queue behind your cell phone. Both Sally and Sean may leave a message for you at the same time and your cell phone provider saves both their messages for you to retrieve at convenience. Similarly the actor library allows multiple actors to send messages concurrently. The senders are non-blocking by default, they send off a message and proceed to take care of their business. The library lets the designated actors synchronously pick their messages to process. Once an actor processes a message or delegates to another actor for concurrent processing, it's ready to receive the next message.

The life cycle of an actor is shown in Figure 8.2, on the next page. Upon creation, an actor may be started or stopped. Once started, it prepares to receive messages. In the active states, the actor is either processing a message or waiting for a new message to arrive. Once it's stopped, it no longer receives any messages. How much time an actor spends waiting vs. processing a message depends on the dynamic nature of your application.

If actors play a major role in your design, you'd expect many of them to float around during the execution of your application. However, threads are limited resources and so tying actors to their threads will be very limiting. To avoid that, actor libraries in general decouple actors from threads. Threads are to actors as cafeteria seats are to office employees. You don't have a designated seat at the cafeteria (you need to find another job if you do), and each time you go for a meal an available seat is provided for you. Much like that when an actor has a message to process or a task to run, it's provided an available thread to run. Good actors don't hold threads when they're not running a task. This allows for a greater number of actors to be active in different states and provides for efficient use of limited available threads.

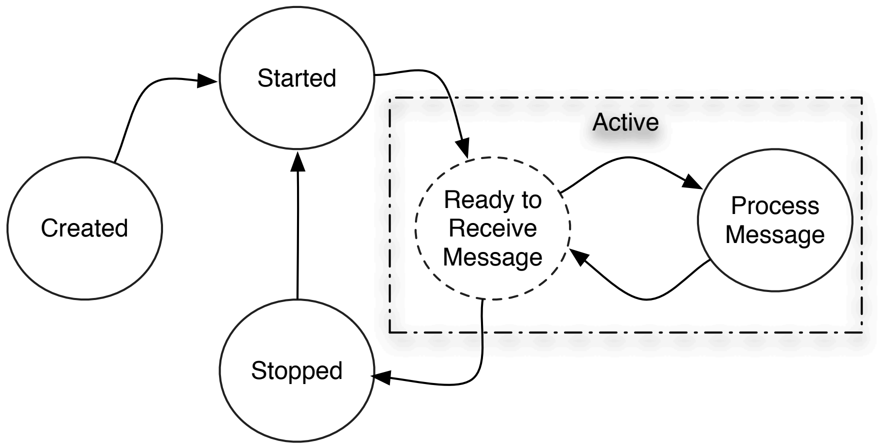


Figure 8.2: Life cycle of an actor

8.3 Creating Actors

You have quite a few choices of actor libraries to pick from, as I mentioned before. In this book I use Akka, a Scala based library[4] with pretty good performance and scalability, and with support for both actors and STM. You can use it from multiple languages on the JVM. In this chapter we'll stick to Java and Scala. In the next chapter we'll take a look at using Akka actors with other languages.

Akka was written in Scala so you will find it quite simple and more natural to create and use actors from Scala. Scala conciseness and idioms shine in the Akka API. At the same time, they've done quite a wonderful job of exposing a traditional Java API so you can easily create and use actors in your Java code. We'll first take a look at using it in Java and then see how that experience simplifies and changes when we use it in Scala.

Creating Actors in Java

Akka's abstract class akka.actor.UntypedActor represents an actor. Simply extend this and implement the required onReceive() method—this

4. In addition to Akka, there are at least two more Scala based libraries—Scala actors library and the Lift actors.

method is called whenever a message arrives for the actor. Let's give it a shot, we'll create an actor, how about a HollywoodActor that'll respond to requests to play different roles.

Download favoringIsolatedMutability/java/create/HollywoodActor.java

```
public class HollywoodActor extends UntypedActor {
  public void onReceive(final Object role) {
    System.out.println("Playing " + role +
      " from Thread " + Thread.currentThread().getName());
  }
}
```

The onReceive() method takes an Object as parameter. In this example we're simply printing it out along with the details of the thread that's processing the message. We'll learn how to deal with different types of messages later.

Our actor is all set and waiting for us to say action. We need to create an instance of the actor and send play-role messages, so let's get to that.

Download favoringIsolatedMutability/java/create/UseHollywoodActor.java

```
public class UseHollywoodActor {
  public static void main(final String[] args) throws InterruptedException {
    ActorRef johnnyDepp = Actors.actorOf(HollywoodActor.class).start();

    johnnyDepp.sendOneWay("Jack Sparrow");
    Thread.sleep(100);
    johnnyDepp.sendOneWay("Edward Scissorhands");
    Thread.sleep(100);
    johnnyDepp.sendOneWay("Willy Wonka");

    Actors.registry().shutdownAll();
  }
}
```

In Java we'd generally create objects using new, but Akka actors are not simple objects—they're active objects. So, we create them using a special method actorOf(). Alternately, we could create an instance using new and wrap it around a call to actorOf() to get an actor reference, but let's get to that later. As soon as we create the actor, we start it by calling the start() method. When we start an actor, Akka puts it into a registry; the actor is accessible through the registry until the actor is stopped. In the example, now johnnyDepp is a reference to our actor instance and it's of type ActorRef.

Next we send a few messages to the actor with roles to play using the sendOneWay() method. Once a message is sent, we really don't have to wait. However, in this case, the delay will help us learn one more detail, how actors switch threads as you'll see soon. In the end we ask to close down all running actors. Instead of calling the shutdownAll() method, you may call the stop() method on individual actors or send them a kill message as well.

Alright, it's time to run the example. Compile the code using javac compiler and remember to specify the classpath to Akka library files. You can simply run the program as you would run regular Java programs. Again, remember to provide the necessary jars in the classpath. Here's the command I used on my system.

```
javac -d . -classpath $AKKA_JARS HollywoodActor.java UseHollywoodActor.java
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseHollywoodActor
```

where AKKA_JARS is defined as:

```
export AKKA_JARS="$SCALA_HOME/lib/scala-library.jar:\
$AKKA_HOME/dist/akka-actor-1.0.jar:\
$AKKA_HOME/dist/akka-typed-actor-1.0.jar:\
$AKKA_HOME/lib_managed/compile/aspectwerkz-2.2.3.jar:\
$AKKA_HOME/lib_managed/compile/uuid-3.2.jar:\
$AKKA_HOME/lib_managed/compile/configgy-2.0.2-nologgy.jar:\
$AKKA_HOME/lib_managed/compile/slf4j-api-1.6.0.jar:\
$AKKA_HOME/lib_managed/compile/logback-classic-0.9.24.jar:\
$AKKA_HOME/lib_managed/compile/logback-core-0.9.24.jar:\
$AKKA_HOME/config:\
."
```

Define the AKKA_JARS environment variable appropriately for your operating system to match the location where you have Scala and Akka installed. You may use the scala-library.jar that comes with Akka or you may use it from your local Scala installation, if you have it installed.

By default Akka prints extra log messages on the standard output. You can silence those messages, if you like, by simply creating a file named logback.xml in the $AKKA_HOME/config directory with just the element <configuration /> in it. Since this directory is in the classpath the logger will know to silence the messages. Instead of silencing the log you may also configure it with useful options, for details see instructions at http://logback.qos.ch/manual/configuration.html.

Compile and run the code to watch our actor responding to the messages.

```
Playing Jack Sparrow from Thread akka:event-driven:dispatcher:global-1
Playing Edward Scissorhands from Thread akka:event-driven:dispatcher:global-2
Playing Willy Wonka from Thread akka:event-driven:dispatcher:global-3
```

The actor responded to the messages one at a time. The output also lets us peek at the thread that's running the actor and it's not the same thread each time. It's possible that the same thread handles multiple messages or it could be different like in this sample output—but in any case only one message will be handled at any time. The key point is that the actors are single threaded but don't hold their threads hostage. They gracefully release their threads when they wait for a message, the delay we added helped introduce this wait and illustrate this point.

The actor we created did not take any parameters at the construction time. If you desire, we can send parameters during actor creation. For example, if you want to initialize the actor with the Hollywood actor's name, we can do that.

Download favoringIsolatedMutability/java/params/HollywoodActor.java

```
public class HollywoodActor extends UntypedActor {
  private final String name;

  public HollywoodActor(final String theName) { name = theName; }

  public void onReceive(final Object role) {
    if(role instanceof String)
      System.out.println(String.format("%s playing %s", name, role));
    else
      System.out.println(name + " plays no " + role);
  }
}
```

The new version of the class HollywoodActor takes a value name of type String as the constructor parameter. While we're at it, let's take care of handling the unrecognized incoming message format. In this example we simply print a message saying the Hollywood actor does not play that unrecognized message. You can take other actions such as returning an error code, logging, calling the user's mom to report,... Let's see how we can pass the actual argument for this constructor parameter.

Download favoringIsolatedMutability/java/params/UseHollywoodActor.java

```
public class UseHollywoodActor {
  public static void main(final String[] args) throws InterruptedException {

    ActorRef tomHanks = Actors.actorOf(new UntypedActorFactory() {
        public UntypedActor create() { return new HollywoodActor("Hanks"); }
      }).start();
    tomHanks.sendOneWay("James Lovell");
    tomHanks.sendOneWay(new StringBuilder("Politics"));
    tomHanks.sendOneWay("Forrest Gump");
    Thread.sleep(1000);
    tomHanks.stop();
  }
}
```

We communicate with actors by sending messages and not invoking methods directly. Akka wants to make it hard to get a direct reference to actors and wants us to get only a reference to ActorRef. This allows Akka to ensure that we don't add methods to actors and interact with them directly, as that would takes us back to the evil land of shared mutability that we're trying so hard to avoid. This controlled creation of actors also allows Akka to garbage collect the actors appropriately. So, if we try to create an instance of an actor class directly, we'll get a runtime exception akka.actor.ActorInitializationException with a message "You can not create an instance of an actor explicitly using 'new'."

Akka allows us to create an instance in a controlled fashion, within a create() method. So, let's implement this method in an anonymous class that implements the UntypedActorFactory interface and within this method create our actor instance sending the appropriate construction time parameters. The subsequent call to actorOf() turns the regular object that extends from UntypedActor into an Akka actor. We can then pass messages to this actor like before.

Our HollywoodActor only accepts messages of type String, but in the example, we're sending an instance of StringBuilder with value Politics. The runtime type checking we performed in the onReceive() takes care of this. Finally we stop the actor by calling the stop() method. The delay introduced keeps the program alive while the actor responds to messages—the actors run in a non-daemon thread pool and will not keep your JVM alive if all daemon threads are gone. Take it for a ride to see the output.

```
Hanks playing James Lovell
Hanks plays no Politics
Hanks playing Forrest Gump
```

Creating Actors in Scala

To create an Akka actor in Scala, instead of the UntypedActor we extended in the Java version, we'll extend from the Actor trait and implement the required receive() method. Let's implement in Scala the HollywoodActor actor class that we wrote earlier in Java.

Download favoringIsolatedMutability/scala/create/HollywoodActor.scala

```
class HollywoodActor extends Actor {
  def receive = {
    case role =>
    println("Playing " + role +
      " from Thread " + Thread.currentThread().getName())
  }
}
```

The receive() method implements a PartialFunction and takes the form of Scala pattern matching, but don't let those details distract you now. This method is called when a message arrives and, if it helps, think of the receive() as a glorified switch statement for now. The implementation is much the same as in the Java version.

You saw how to define an actor, let's turn our attention to using this actor.

Download favoringIsolatedMutability/scala/create/UseHollywoodActor.scala

```
object UseHollywoodActor {
  def main(args : Array[String]) :Unit = {
    val johnnyDepp = Actor.actorOf[HollywoodActor].start()

    johnnyDepp ! "Jack Sparrow"
    Thread.sleep(100)
    johnnyDepp ! "Edward Scissorhands"
    Thread.sleep(100)
    johnnyDepp ! "Willy Wonka"

    Actors.registry().shutdownAll()
  }
}
```

There are a few flavors of the actorOf() method and here we're using the version that takes as parameter an actor class manifest, presented as [HollywoodActor]. As soon as we create the actor, we start it by calling the start() method. In the example, now johnnyDepp is a reference to our actor instance and it's of type ActorRef, however, since Scala has type inference we didn't have to specify the type.

Next we send a few messages to the actor with the roles to play. Oh wait, there one other detail, we use a special method ! to send the message. When you see actor ! message, read it right-to-left, as sending the message to the actor. Scala's conciseness again is at play here, instead of calling actor.!(message), you can simply drop the dot and parenthesis and write actor ! message. If you prefer, you can use the Java style methods with Scala conciseness as in actor sendOneWay message. The rest of the code in the example is similar to the Java example.

Alright, it's time to run the example. Compile the code using scalac compiler and remember to specify the classpath to Akka library files. You can simply run the program as you would run regular Java programs. Again, remember to provide the necessary jars in the classpath. Here's the command I used on my system, substitute the paths as appropriate for your system based on where you've installed Scala and Akka.

```
scalac -classpath $AKKA_JARS HollywoodActor.scala UseHollywoodActor.scala
java -classpath $AKKA_JARS com.agiledeveloper.pcj.UseHollywoodActor
```

If you see log messages on the standard output and want to silence them see the details I presented for that after the Java actor example. Once you compile and run the code, the output you'll see will be similar to the one the Java version produced.

```
Playing Jack Sparrow from Thread akka:event-driven:dispatcher:global-1
Playing Edward Scissorhands from Thread akka:event-driven:dispatcher:global-2
Playing Willy Wonka from Thread akka:event-driven:dispatcher:global-3
```

If you want to pass parameters, like the Hollywood actor's name, to the actor, it is a lot simpler in Scala than in the Java version. Let's first change the class HollywoodActor to accept a constructor parameter.

Download favoringIsolatedMutability/scala/params/HollywoodActor.scala

```
class HollywoodActor(val name : String) extends Actor {
  def receive = {
    case role : String => println(String.format("%s playing %s", name, role))
    case msg => println(name + " plays no " + msg)
  }
}
```

The new version of the class HollywoodActor takes a value name of type String as the constructor parameter. While we're at it let's take care of handling the unrecognized incoming message format. Rather than using instanceof the **case** statements take care of matching the message with various patterns—type of the message, in this example.

Creating an actor that accepts constructor parameter took some effort in Java, but that fizzles down to something quite simple in Scala.

Download favoringIsolatedMutability/scala/params/UseHollywoodActor.scala

```
object UseHollywoodActor {
  def main(args : Array[String]) : Unit = {

    val tomHanks = Actor.actorOf(new HollywoodActor("Hanks")).start()
    tomHanks ! "James Lovell"
    tomHanks ! new StringBuilder("Politics")
    tomHanks ! "Forrest Gump"
    Thread.sleep(1000)
    tomHanks.stop()
  }
}
```

We instantiated the actor using the **new** keyword and then passed the instance to the actorOf() method. This turns the regular object that extends from Actor into an Akka actor. We then pass messages like we did before. Rest of the code is similar to the Java version. Run the code and ensure the output is similar to the Java version's output.

```
Hanks playing James Lovell
Hanks plays no Politics
Hanks playing Forrest Gump
```

8.4 Sending and Receiving Messages

You can send just about any type of message to an actor—String, Integer, Long, Double, List, Map, tuples, Scala case classes,... all of which are immutable. I have a special liking for tuples, not because it's amusing when we mispronounce it as two-ples, but they're lightweight, immutable, and one of the easiest instances to create. For example, to create a tuple of two numbers in Scala you simply write (number1, number2). Scala's case classes are ideal to serve as message types—they're immutable, works well with pattern matching, and are quite easy to make copy of. In Java you could pass an unmodifiable Collection as messages to send more than one object in a message. When you pass a message to an actor, by default, you're passing the message's reference when both sender and receiver are within the same JVM[5]. It's your responsibility to ensure that the message you pass is immutable, especially if you decide to send instances of your own classes. You may also ask Akka to serialize the message so a copy of the message is delivered instead of a reference.

The simplest way to communicate with an actor is to send a message and move on. The send is non-blocking and the calling actor/thread

5. Akka also supports remote actors, allowing you to message between discrete processes running on different machines.

proceeds with its work. You use the sendOneWay() method, or the ! method in Scala, to send an one-way message.

Akka also provides two-way communication where you can send a message and expect a response from the actor. The calling thread, in this case, will block until it receives a response or exceeds timeout. Let's take a look at how to send and receive messages first in Java and then in Scala.

Send/Receive in Java

Use the sendRequestReply() method to send a message and wait for a response. This method return a Future of an Option Scala type. Think of Option as a union which may or may not have a value. No value is represented by the special Scala type None. Once you call the sendRequestReply method, you block for a response. If a response does not arrive within a (configurable) timeout, you're given a None. Let's take a look at an example of two-way messaging.

Download favoringIsolatedMutability/java/2way/FortuneTeller.java

```java
public class FortuneTeller extends UntypedActor {
  public void onReceive(final Object name) {
      getContext().replyUnsafe(String.format("%s you'll rock", name));
  }

  public static void main(final String[] args) {
    ActorRef fortuneTeller = Actors.actorOf(FortuneTeller.class).start();

    try {
      Object response = fortuneTeller.sendRequestReply("Joe");
      System.out.println(response);
    } catch(ActorTimeoutException ex) {
      System.out.println("Never got a response before timeout");
    } finally {
      fortuneTeller.stop();
    }
  }
}
```

We have a FortuneTeller actor that wants to respond to the message it receives. It responds back to the message sender by calling the replyUnsafe() methods on the call context obtained through getContext(). The replyUnsafe() method sends off, without blocking, a response to the calling actor. But, there's no calling actor in the code. In the main() we invoked the sendRequestReply() method. This method internally creates a Future and waits on that until it gets a result, an exception, or a timeout. Let's check Joe's fortune by running the code.

```
Joe you'll rock
```

There's one thing a bit unfortunate with this FortuneTeller, it relies on the sender to be available. We called the sendRequestReply() method and so there was a sender's Future waiting for a response. If we had called sendOneWay() instead, the replyUnsafe() method would fail. To avoid that we need to check if a sender is available before we call the replyUnsafe() method. We can do this by obtaining the sender reference from the context. Alternately, we can use the replySafe() method which will return a true if the sender was present and a false if no sender is available. So here's the modified FortuneTeller that'll handles the case when sender is not known.

Download favoringIsolatedMutability/java/2waysafe/FortuneTeller.java

```
public class FortuneTeller extends UntypedActor {
  public void onReceive(final Object name) {
    if(getContext().replySafe(String.format("%s you'll rock", name)))
      System.out.println("Mesage sent for " + name);
    else
      System.out.println("Sender not found for " + name);
  }

  public static void main(final String[] args) {
    ActorRef fortuneTeller = Actors.actorOf(FortuneTeller.class).start();

    try {
      fortuneTeller.sendOneWay("Bill");
      Object response = fortuneTeller.sendRequestReply("Joe");
      System.out.println(response);
    } catch(ActorTimeoutException ex) {
      System.out.println("Never got a response before timeout");
    } finally {
      fortuneTeller.stop();
    }
  }
}
```

The new version of FortuneTeller will not fail if a sender is not known, it gracefully handles the misfortune.

```
Sender not found for Bill
Mesage sent for Joe
Joe you'll rock
```

The call to sendRequestReply() blocks while waiting for a response, but the call to sendOneWay() is non-blocking and yields no response. If you want to receive a response but don't want to wait for it, you can use a more elaborate method sendRequestReplyFuture() which will return to you a Future object. You can go about doing your work until you want the response at which time either you can block or query the future object to see if response is available. Similarly on the side of the actor, you can get the senderFuture from the context reference and communicate through that right away or later on when you have a response ready. We'll take a look at using these later in an example.

Exercise caustion when you use the sendRequestReply() and sendOneWay() method as call to these methods block and can have negative impact on performance and scalability.

Send/Receive in Scala

If you'd like to use send/receive in Scala, be ready to see some differences from the Java API:

- You can directly use the self property to get access to your actor. Using this property, you can call the reply() method, which is the unsafe equivalent on the Scala side, or you can use the replySafe() method.
- You could call sendRequestReply() method or you could call a more elegant !! method—they say beauty is in the eyes of the beholder. Similarly, !!! can be used in place of sendRequestReplyFuture() method.
- Instead of returning an Object, the sendRequestReply() method returns a Scala Option. If the response arrived, this would be an instance of Some(T) and None in the case of a timeout. So, unlike the Java version, there's no exception in the case of a timeout.

Let's implement the FortuneTeller in Scala using the unsafe reply() method first.

Download favoringIsolatedMutability/scala/2way/FortuneTeller.scala

```scala
class FortuneTeller extends Actor {
  def receive = {
    case name : String =>
      self.reply(String.format("%s you'll rock", name))
  }
}

object FortuneTeller {
  def main(args : Array[String]) : Unit = {
    val fortuneTeller = Actor.actorOf[FortuneTeller].start()

    val response = fortuneTeller !! "Joe"
    response match {
      case Some(responseMessage) => println(responseMessage)
      case None => println("Never got a response before timeout")
    }


    fortuneTeller.stop()
  }
}
```

In the actor code we can see the two differences, one related to self instead of getContext() and the other is reply() instead of replyUnsafe(). On the caller side, we apply pattern matching on the response we received from the call to !!, that is the sendRequestReply() method. The first case is exercised if an actual response arrived and the second case with None is used in case of a timeout. The output of this code is the same as the Java version, as you'd expect.

```
Joe you'll rock
```

Other than the changes we discussed, using the safe version of reply is not much different from the Java version, you can use the reply_?() or replySafe().

Download favoringIsolatedMutability/scala/2waysafe/FortuneTeller.scala

```
class FortuneTeller extends Actor {
  def receive = {
    case name : String =>
      if(self.reply_?(String.format("%s you'll rock", name)))
        println("Message sent for " + name)
      else
        println("Sender not found for " + name)
  }
}

object FortuneTeller {
  def main(args : Array[String]) : Unit = {
    val fortuneTeller = Actor.actorOf[FortuneTeller].start()

    fortuneTeller ! "Bill"

    val response = fortuneTeller !! "Joe"
    response match {
      case Some(responseMessage) => println(responseMessage)
      case None => println("Never got a response before timeout")
    }


    fortuneTeller.stop()
  }
}
```

The new version of FortuneTeller will not fail if a sender is not known.

```
Sender not found for Bill
Message sent for Joe
Joe you'll rock
```

It is quite convenient that Akka passes the sender references under the covers when a message is sent. This eliminates the need to pass the sender explicitly as part of the message and removes so much noise and effort in the code.

If method names like !, !!, !!!, and reply_?() bother your eyes, you can use alternate names like sendOneWay(), sendRequestReply(), sendRequestReplyFuture(), and replySafe(), respectively.

8.5 Working with Multiple Actors

You know how to create an actor and send messages to it. Let's get a feel for putting multiple actors to work. In Chapter 2, *Division of Labor*, on page 29 we created a concurrent program to count primes in a range. In the example in Section 2.4, *Concurrent Computation of Prime Numbers*, on page 42 we used ExecutorService, Callable, and Future and filled up a little over a page with code. Let's see how that example shapes up with Akka actors first in Java and then in Scala.

Multiple Actors in Java

Given a number, like ten million, we divided the computation of primes into different ranges and distributed these ranges over several threads. Here we're going to use actors. Let's start with the actor's onReceive() method.

Download favoringIsolatedMutability/java/primes/Primes.java

```
public class Primes extends UntypedActor {
  public void onReceive(final Object boundsList) {
    List<Integer> bounds = (List<Integer>) boundsList;
    int count = PrimeFinder.countPrimesInRange(bounds.get(0), bounds.get(1));
    getContext().replySafe(new Integer(count));
  }
```

To compute the number of primes in a range, we need the lower and upper bounds of the range. Our actor receives this bundled as a List in the message parameter to onReceive(). We invoke the countPrimesInRange() method of yet to be written PrimeFinder and send back the result to the caller using the replySafe() method.

Given a number we need to divide it into the given number of parts and delegate the finding of primes in each of these parts to different actors. Let's do that in a countPrimes() static method.

Download favoringIsolatedMutability/java/primes/Primes.java

```
public static int countPrimes(
  final int number, final int numberOfPartitions) {
  int chunksPerPartition = number / numberOfPartitions;
  List<Future<?>> results = new ArrayList<Future<?>>();
  for(int index = 0; index < numberOfPartitions; index++) {
    final int lower = index * chunksPerPartition + 1;
    final int upper = (index == numberOfPartitions - 1) ? number :
        lower + chunksPerPartition - 1;
    List<Integer> bounds = Collections.unmodifiableList(
      Arrays.asList(new Integer[] {lower, upper}));
    ActorRef primeFinder = Actors.actorOf(Primes.class).start();
    results.add(primeFinder.sendRequestReplyFuture(bounds));
  }

  int count = 0;
  for(Future<?> result : results)
    count += (Integer)(result.await().result().get());

  Actors.registry().shutdownAll();
  return count;
}
```

Once we determined the bounds for each part, we wrapped that into an unmodifiable collection—remember messages have to be immutable. We then call sendRequestReplyFuture() so we can send requests to all the partitions without being blocked. We save away the Future (here it's akka.dispatch.Future and not the JDK's java.util.concurrent.Future) returned by this method so we can query later for the number of primes result from each part. We call await() on the Future and invoke the result() method on the Future instance returned by await(). This gives us an instance of Scala Option—think of this as a fancy union that holds the value if available or a None otherwise. You get the Integer value from that object finally by calling the get() method.

OK, it's time to drive the code using the command-line parameters for the number and parts.

Download favoringIsolatedMutability/java/primes/Primes.java

```
  public static void main(final String[] args) {
    if (args.length < 2)
      System.out.println("Usage: number numberOfPartitions");
    else {
      long start = System.nanoTime();
      int count = countPrimes(
        Integer.parseInt(args[0]), Integer.parseInt(args[1]));
      long end = System.nanoTime();
      System.out.println("Number of primes is " + count);
      System.out.println("Time taken " + (end - start)/1.0e9);
    }
  }
}
```

The main() is pretty straightforward. Our last step is to write the PrimeFinder that will do the actual work of computing the primes in a range.

Download favoringIsolatedMutability/java/primes/PrimeFinder.java

```
public class PrimeFinder {
  public static boolean isPrime(final int number) {
    if (number <= 1) return false;
    int limit = (int) Math.sqrt(number);
    for(int i = 2; i <= limit; i++) if(number % i == 0) return false;
    return true;
  }

  public static int countPrimesInRange(final int lower, final int upper) {
    int count = 0;
    for(int index = lower; index <= upper; index++)
      if(isPrime(index)) count += 1;

    return count;
  }
}
```

Go ahead and exercise the example code with a large number like ten million and a hundred parts.

```
Number of primes is 664579
Time taken 3.890996
```

Compare the code and output in this section with the code and output in Section 2.4, *Concurrent Computation of Prime Numbers*, on page 42. In both versions you set the number of partitions to 100. There is no pool size to set in the Akka version of primes counting. This is a computation intensive problem and setting the pool size for the ExecutorService version above the number of cores made little difference. So they're fairly close in performance and there is slightly less ceremony in the Akka version than in the ExecutorService. This difference will become more prominent when we need more coordination between the threads/actors, as we progress further in this chapter.

Multiple Actors in Scala

If you'd like to use Scala to implement the primes example, you can enjoy the Scala conciseness for implementing the actor and interacting with it. Let's look at the Scala version of Primes.

Download favoringIsolatedMutability/scala/primes/Primes.scala

```
class Primes extends Actor {
  def receive = {
    case (lower : Int, upper : Int) =>
      val count = PrimeFinder.countPrimesInRange(lower, upper)
      self.replySafe(new Integer(count))
  }
}

object Primes {
  def countPrimes(number : Int, numberOfPartitions : Int) = {
    val chunksPerPartition : Int = number / numberOfPartitions

    val results = new Array[Future[Integer]](numberOfPartitions)
    var index = 0

    while(index < numberOfPartitions) {
      val lower = index * chunksPerPartition + 1
      val upper = if (index == numberOfPartitions - 1)
        number else lower + chunksPerPartition - 1
      val bounds = (lower, upper)
      val primeFinder = Actor.actorOf[Primes].start()
      results(index) = (primeFinder !!! bounds).asInstanceOf[Future[Integer]]
      index += 1
    }

    var count = 0
    index = 0
    while(index < numberOfPartitions) {
      count += results(index).await.result.get.intValue()
      index += 1
    }
    Actors.registry().shutdownAll()
    count
  }

  def main(args : Array[String]) : Unit = {
    if (args.length < 2)
      println("Usage: number numberOfPartitions")
    else {
      val start = System.nanoTime
      val count = countPrimes(args(0).toInt, args(1).toInt)
      val end = System.nanoTime
      println("Number of primes is " + count)
      println("Time taken " + (end - start)/1.0e9)
    }
  }
}
```

There are a few differences between the Java version and this one. The message format is a simple tuple instead of an unmodifiable list. The case within the receive() was able to easily match for this. The for loop in Java has turned into a while loop here. Scala does have quite an elegant for loop, however, it will incur object to primitive conversion overhead. To get a decent performance comparison, I've avoided that elegance here.

Similarly in the PrimeFinder we'll use a while loop instead of a Scala for loop.

Download favoringIsolatedMutability/scala/primes/PrimeFinder.scala

```
object PrimeFinder {
  def isPrime(number : Int) : Boolean = {
    if (number <= 1) return false

    var limit = scala.math.sqrt(number).toInt
    var i = 2
    while(i <= limit) {
      if(number % i == 0) return false
      i += 1
    }
    return true
  }

  def countPrimesInRange(lower : Int, upper : Int) : Int = {
    var count = 0
    var index = lower
    while(index <= upper) {
      if(isPrime(index)) count += 1
      index += 1
    }
    count
  }
}
```

The performance of this version of primes example is comparable to the earlier one you saw.

```
Number of primes is 664579
Time taken 3.88375
```

8.6 Coordinating Actors

The real benefit and fun is when actors coordinate with each other to solve your problems. To make use of concurrency, you'd divide your problem into parts. Different actors may take on different parts and you need to coordinate the communication between them. That's what we'll learn here by using the file size program as an example.

In Section 4.2, *Coordinating Threads*, on page 64 we wrote a program to find the total size of all files under a given directory. In that we launched a hundred threads, each of which explored different subdirectories. We then totaled the sizes that were computed asynchronously. You saw different approaches to doing that, with AtomicLongs and queues. We can sum up those approaches as hard work to deal with shared mutability.

We can save quite a bit of effort and trouble by using isolated mutability with actors to solve that problem. We can do that without compromising performance compared to the shared mutability solutions in that chapter. As an added bonus to synchronization free coding, we'll have a lot simpler code as well, as you'll see soon.

The first step is to create a design for the problem with multiple coordinating actors—we can use two types of actors as in Figure 8.3, on the following page. We'll isolate mutable state within the SizeCollector actor. It'll receive a handful of messages, to keep track of directories that needs to be visited, to keep a total of the file size, and to feed requesting FileProcessor actors with directories to explore. The main code will set these actors in motion and wait for the work to be completed. We'll have a hundred FileProcessor actors to navigate the files under the given directory.

It's time to implement this design using Akka actors first in Java, then in Scala.

Coordinating Actors in Java

Let's first define the messages the SizeCollector will receive.

Download favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```java
class RequestAFile {}

class FileSize {
  public final long size;
  public FileSize(final long fileSize) { size = fileSize; }
}
```

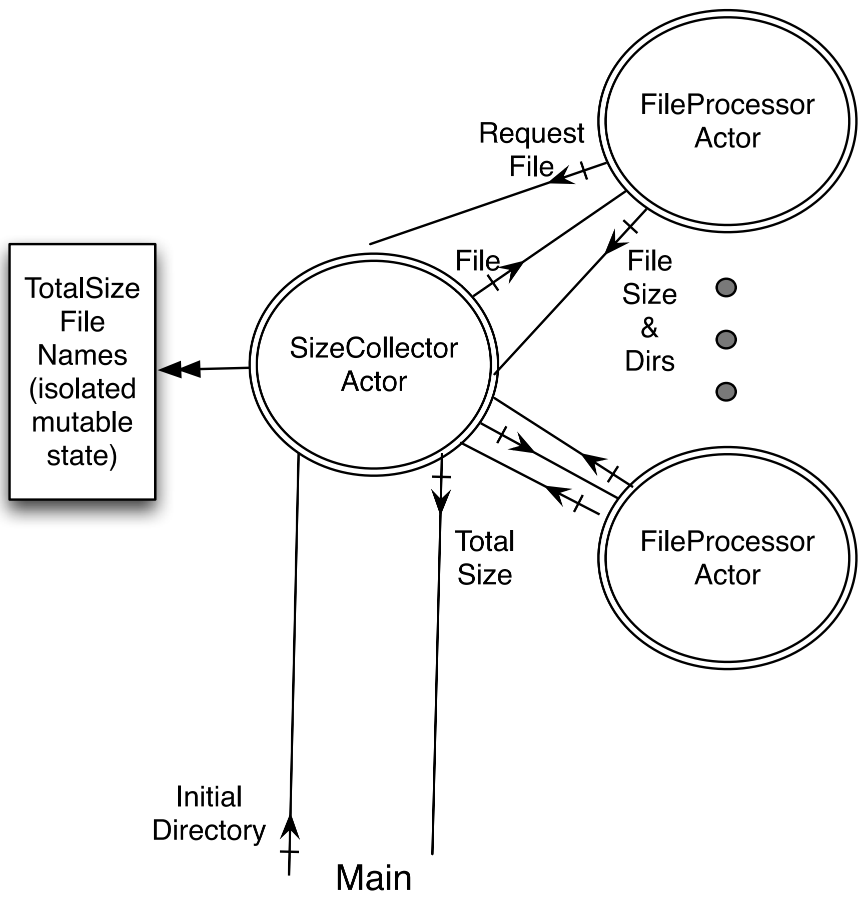


Figure 8.3: Design of the total file size problem using Actors

```
class FetchTotalSize {
  public final String fileName;
  public FetchTotalSize(final String name) { fileName = name; }
}

class FileToProcess {
  public final String fileName;
  public FileToProcess(final String name) { fileName = name; }
}
```

The four messages are represented by immutable classes. Each FileProcessor will use messages of type RequestAFile to place itself on a list with the SizeCollector. FetchTotalSize is a message that the main code will send to the SizeCollector to request for the total file size, the final result. FileSize is a message from FileProcessors that will carry the size of files in directories they explore. Finally FileToProcess is a message that carries the name of the file that needs to be explored.

FileProcessors are the workers with the job to explore a given directory and send back total size of files and names of subdirectories they find. Once they finish that task, they send the RequestAFile to let the SizeCollector know they're ready to take on the task of exploring another directory. You also need to register with the SizeCollector in the first place to receive the first directory to explore. A great place for this is the preStart() method which is called when an actor is started. Let's implement the FileProcessor, remember to receive a reference to the SizeCollector as constructor parameter.

Download favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
class FileProcessor extends UntypedActor {
  private final ActorRef sizeCollector;

  public FileProcessor(final ActorRef theSizeCollector) {
    sizeCollector = theSizeCollector;
  }

  @Override public void preStart() { registerToGetFile(); }

  public void registerToGetFile() {
    sizeCollector.sendOneWay(new RequestAFile(), getContext());
  }

  public void onReceive(Object message) {
    FileToProcess fileToProcess = (FileToProcess) message;
      final File file = new java.io.File(fileToProcess.fileName);
      long size = 0L;
      if(file.isFile()) {
        size = file.length();
      } else {
        File[] children = file.listFiles();
        if (children != null)
          for(File child : children)
            if (child.isFile())
              size += child.length();
            else
              sizeCollector.sendOneWay(new FileToProcess(child.getPath()));
      }

      sizeCollector.sendOneWay(new FileSize(size));
      registerToGetFile();
    }
  }
```

In the onReceive() method we discover subdirectories and send them to the SizeCollector using the sendOneWay() method. For the files in this subdirectory, we total their sizes and in the end of the task send that to the SizeCollector. The final step at the end of the task is to register the FileProcessor to get the next directory to explore.

The FileProcessor is all set and ready to explore directories. The SizeCollector is the mastermind that manages the isolated mutable state and keeps the FileProcessors busy with directories to explore until the final result is computed. It handles four types of messages. Let's first look at the code and then discuss the actions for each of these messages.

Download favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
class SizeCollector extends UntypedActor {
  private final List<String> toProcessFileNames = new ArrayList<String>();
  private final List<ActorRef> fileProcessors = new ArrayList<ActorRef>();
  private akka.dispatch.CompletableFuture<Object> fetchSizeFuture;
  private long pendingNumberOfFilesToVisit = 0L;
  private long totalSize = 0L;

  public void sendAFileToProcess() {
    if(!toProcessFileNames.isEmpty() && !fileProcessors.isEmpty())
      fileProcessors.remove(0).sendOneWay(
        new FileToProcess(toProcessFileNames.remove(0)));
  }

  public void onReceive(Object message) {
    if (message instanceof FetchTotalSize) {
      fetchSizeFuture = getContext().getSenderFuture().get();
      toProcessFileNames.add(((FetchTotalSize)(message)).fileName);
      pendingNumberOfFilesToVisit = 1;
      sendAFileToProcess();
    }
```

```
      if (message instanceof RequestAFile) {
        fileProcessors.add(getContext().getSender().get());
        sendAFileToProcess();
      }

      if (message instanceof FileToProcess) {
        toProcessFileNames.add(((FileToProcess)(message)).fileName);
        pendingNumberOfFilesToVisit += 1;
        sendAFileToProcess();

      }

      if (message instanceof FileSize) {
        totalSize += ((FileSize)(message)).size;
        pendingNumberOfFilesToVisit -= 1;
        if(pendingNumberOfFilesToVisit == 0)
          fetchSizeFuture.completeWithResult(totalSize);
      }
    }
  }
```

The SizeCollector keeps two lists, one for the directories to visit and the other of idling FileProcessors. It also maintains a reference to the future object that represents a call from the main eagerly waiting for the final total. The two long variables are used to keep track of how many directories are being explored at anytime and the evolving total file size.

The sendAFileToProcess() message is used to dispatch directories to idling FileProcessors to explore.

The SizeCollector expects to receive four types of messages in the onReceive() message handler. Each of these messages have distinct purpose.

The FetchTotalSize will be sent by the main code with the name of the initial directory to find the size of. The SizeCollector saves a reference to the sender, the main, and dispatches a FileProcessor to explore the given directory.

As FileProcessors become idle, they send the RequestAFile message and the SizeCollector saves the actors' references in the idling processors list.

FileToProcess is a message that the SizeCollector will send and also receive. It will receive this message from FileToProcesss as they discover subdirectories. It promptly sends off, in the sendAFileToProcess() method using the same message format, for idling FileToProcesss to explore.

The final message handled by the SizeCollector is FileSize which is sent by the FileProcessors and holds the size of files in a directory explored.

Let’s implement the last piece of the design, the main code that should request for the total file size for a directory from the SizeCollector actor.

Download favoringIsolatedMutability/java/filesize/ConcurrentFileSizeWAkka.java

```
public class ConcurrentFileSizeWAkka {
  public static void main(String[] args) {
    long start = System.nanoTime();
    final ActorRef sizeCollector =
      Actors.actorOf(SizeCollector.class).start();
    sizeCollector.setTimeout(100000);
    new Thread(new Runnable() {
      public void run() {
        for(int i = 0; i < 100; i++)
          Actors.actorOf(new UntypedActorFactory() {
              public UntypedActor create() {
                return new FileProcessor(sizeCollector); }
            }).start();
      }
    }).start();

    long totalSize =
      (Long) sizeCollector.sendRequestReply(new FetchTotalSize(args[0]));
    Actors.registry().shutdownAll();

    long end = System.nanoTime();
    System.out.println("Total size is " + totalSize);
    System.out.println("Time taken is " + (end - start)/1.0e9);
  }
}
```

The main code launches a thread that will asynchronously create a hundred FileProcessor actors. It then sends a two-way message to the SizeCollector actor with the directory to find the size for and waits for the total size. If the result arrives within the timeout, configured to 100 seconds, the result is displayed along with the time it took. Otherwise, an exception will be thrown abruptly terminating the program.

Let’s get these actors rolling and ask them to explore the /usr directory.

```
Total size is 3793911517
Time taken is 8.599308
```

Compare the output of this code which uses isolated mutability with the versions of code that used shared mutability in Section 4.2, *Coordinating Threads*, on page 64. All the versions produced the same file size for the /usr directory and had comparable performance. The biggest difference in the actor based version is that there’s no synchronization involved in the code, no latches, queues, or AtomicLongs to mess with. The result—comparable performance with greater simplicity—*no worries*.

Coordinating Actors in Scala

We got the file size program working in Java using Akka actors. You can implement that design in Scala and benefit from its conciseness. The first difference from the Java version is in the messages. Scala has **case** classes that provide highly expressive syntax to create immutable classes. These serve real well for message types. So, let's implement the four message types using case classes.

Download favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
case class RequestAFile()
case class FileSize(val size : Long)
case class FetchTotalSize(val fileName : String)
case class FileToProcess(val fileName : String)
```

The FileProcessor is a direct translation from the Java version to Scala, there is nothing new other than things we've discussed so far.

Download favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
class FileProcessor(val sizeCollector : ActorRef) extends Actor {
  override def preStart = registerToGetFile

  def registerToGetFile = { sizeCollector ! RequestAFile }

  def receive = {
    case FileToProcess(fileName) =>
      val file = new java.io.File(fileName)
      var size = 0L
      if(file.isFile()) {
        size = file.length()
      } else {
        val children = file.listFiles()
        if (children != null)
          for(child <- children)
            if (child.isFile())
              size += child.length()
            else
              sizeCollector ! new FileToProcess(child.getPath())
      }

      sizeCollector ! new FileSize(size)
      registerToGetFile
  }
}
```

The SizeCollector has no surprises too and so is a direct translation.

Download favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
class SizeCollector extends Actor {
  var toProcessFileNames = List.empty[String]
  var fileProcessors = List.empty[ActorRef]
  var fetchSizeFuture : Option[akka.dispatch.CompletableFuture[Any]] = None
  var pendingNumberOfFilesToVisit = 0L
  var totalSize = 0L

  def sendAFileToProcess() : Unit = {
    if(!toProcessFileNames.isEmpty && !fileProcessors.isEmpty) {
      fileProcessors.head ! new FileToProcess(toProcessFileNames.head)
      fileProcessors = fileProcessors.tail
      toProcessFileNames = toProcessFileNames.tail
    }
  }

  def receive = {
    case FetchTotalSize(fileName) =>
      fetchSizeFuture = self.getSenderFuture()
      toProcessFileNames = fileName :: toProcessFileNames
      pendingNumberOfFilesToVisit = 1
      sendAFileToProcess()

    case RequestAFile =>
      fileProcessors = self.getSender().get :: fileProcessors
      sendAFileToProcess()

    case FileToProcess(fileName) =>
      toProcessFileNames = fileName :: toProcessFileNames
      pendingNumberOfFilesToVisit += 1
      sendAFileToProcess()

    case FileSize(size) =>
      totalSize += size
      pendingNumberOfFilesToVisit -= 1
      if(pendingNumberOfFilesToVisit == 0)
        fetchSizeFuture.get.completeWithResult(totalSize)
  }
}
```

In the main() method of the Java-based ConcurrentFileSizeWAkka implementation, we asynchronously created 100 FileProcessor actors. There is a simple way to launch asynchronous threads in Akka—the spawn() method. It allows you to create an actor which will run in isolation, but you can't send any messages to it. It's a good candidate for one-off tasks that you want to run asynchronously. Let's use that as we translation the main code from the Java version.

Download favoringIsolatedMutability/scala/filesize/ConcurrentFileSizeWAkka.scala

```
object ConcurrentFileSizeWAkka {
  def main(args : Array[String]) : Unit = {
    val start = System.nanoTime()
    val sizeCollector = Actor.actorOf[SizeCollector].start()
    sizeCollector.setTimeout(100000)
    Actor.spawn {
      for(i <- 1 to 100)
        Actor.actorOf(new FileProcessor(sizeCollector)).start()
    }

    val totalSize = (sizeCollector !! FetchTotalSize(args(0))) match {
      case Some(size) => size
      case None => println("Timeout")
    }

    Actors.registry().shutdownAll()

    val end = System.nanoTime()
    println("Total size is " + totalSize)
    println("Time taken is " + (end - start)/1.0e9)
  }
}
```

Go ahead the try the Scala version of the file size program, like I've done for the /usr directory and observe the performance is comparable and the size is the same as in the Java version.

```
Total size is 3793911517
Time taken is 8.321386
```

8.7 Using Typed Actors

The actors you saw so far accepted messages. We passed different types of messages, String, tuples, case classes/custom messages, etc. Yet, passing these messages felt quite different from the regular method calls we're used to in everyday programming. Typed actors help bridge that gap by allowing you to make regular method calls and translating them to messages under the covers. Think of a typed actor as an active object, which runs in its own single lightweight event driven thread, with an intercepting proxy to turn normal looking method calls into asynchronous non-blocking messages.

To implement an actor you simply wrote a class that extended UntypedActor or Actor trait/abstract classes. To implement a typed actor, you'll create an interface-implementation pair (in Scala you don't write interfaces, instead you use traits with no implementation).

To instantiate an actor you used the actorOf() method of the Actor class. To instantiate a typed actor, you'll use TypedActor's newInstance() method.

The reference you receive from the TypedActor is the intercepting proxy that converts methods to asynchronous messages. void methods transform to the sendOneWay() or ! methods while methods that return results are transformed into sendRequestReply() or !! methods. Methods that return a Future are transformed into sendRequestReplyFuture() or !!! methods.

The EnergySource class we refactored in Chapter 5, *Taming Shared Mutability*, on page 85 with modern Java concurrency API and in Section 6.7, *Using Akka Refs and Transactions*, on page 113 using STM is a good candidate to be a typed actor. It has mutable state that we can isolate using an actor. Each instance of EnergySource will run only in a single thread, so no issue of race conditions. When multiple threads call on an instance of EnergySource, the calls will jump threads and run sequentially on the instance. Remember, actors graciously don't hold threads hostage, so they share threads across instances and provide better throughput—typed actors do the same.

The EnergySource did quite a few things, it allowed us to query for energy level, the usage count, to utilize energy, and it even replenished energy automatically in the background. You certainly want your actor based version to do all that, but let's not rush in. We'll build it incrementally so we can focus on one thing at a time. Let's build the Java version first and then the Scala version.

Using Typed Actors in Java

Typed actors need a pair of interface and implementation. So, let's start with the interface for the EnergySource.

Download favoringIsolatedMutability/java/typed1/EnergySource.java

```
public interface EnergySource {
  public long getUnitsAvailable();
  public long getUsageCount();
  public void useEnergy(long units);
}
```

The pairing implementation for this interface is the EnergySourceImpl. The only difference between this and a regular Java class is that you extend the TypedActor class to turn it into an active object.

Download favoringIsolatedMutability/java/typed1/EnergySourceImpl.java

```
public class EnergySourceImpl extends TypedActor implements EnergySource {
  private final long MAXLEVEL = 100L;
  private long level = MAXLEVEL;
  private long usageCount = 0L;

  public long getUnitsAvailable() { return level; }

  public long getUsageCount() { return usageCount; }

  public void useEnergy(long units) {
    if (units > 0 && level - units >= 0) {
      System.out.println(
        "Thread in useEnergy: " + Thread.currentThread().getName());
      level -= units;
    }
  }
}
```

TypedActor ensures these methods are all mutually exclusive, that is, only one of these methods will run at any given instance. So, there is no need to synchronize or lock access to any of the fields in our class. In order to get a feel for the threads that execute the actor, let's sprinkle a few print statements in this sample code. Finally. we're ready to use the typed actor, so let's write the code for UseEnergySource.

Download favoringIsolatedMutability/java/typed1/UseEnergySource.java

```
public class UseEnergySource {
  public static void main(String[] args) throws InterruptedException {
    System.out.println("Thread in main: " + Thread.currentThread().getName());

    final EnergySource energySource =
      TypedActor.newInstance(EnergySource.class, EnergySourceImpl.class);

    System.out.println("Energy units " + energySource.getUnitsAvailable());

    ExecutorService service = Executors.newFixedThreadPool(2);
    for(int i = 0; i < 2; i++) {
      service.execute(new Runnable() {
        public void run() { energySource.useEnergy(10); }
      });
    }

    service.shutdown();
    Thread.sleep(1000);
    System.out.println("Energy units " + energySource.getUnitsAvailable());

    TypedActor.stop(energySource);
  }
}
```

We create an instance of the typed actor using the TypedActor's newInstance() method. Then we first call the getUnitsAvailable() method and since this method returns a value, our calling thread will block for a response from the typed actor. Calls to useEnergy() are nonblocking since that's a void method returning no response. We make two concurrent calls to this method by placing each of them in a separate thread. Finally we ask for energy level again, after a delay allowing the asynchronous messages to finish, and then ask the actor to be shutdown. Let's take a look at the output from this code.

```
Thread in main: main
Energy units 100
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Energy units 80
```

The typed actor EnergySourceImpl executed only one method at a time. So, even though we invoked useEnergy() concurrently, they ran sequentially. If you like, add a small delay with Thread.sleep(1000) in the useEnergy() method to watch the concurrent calls being executed in sequence. The actor also gracefully switched execution threads. While the main() ran in the main thread, the methods of the actor ran sequentially, but, in different threads managed by Akka.

The mutable state of the energy source is isolated within the EnergySourceImpl actor—I say it's isolated not because it's encapsulated within this class, but because typed actors control access to the mutable state to only one actor with at most one thread running at anytime.

Using Typed Actors in Scala

Typed actors need a pair of interface and implementation. In Scala you don't create interface, instead you create traits with no implementation. Let's define the EnergySource as a trait.

Download favoringIsolatedMutability/scala/typed1/EnergySource.scala

```
trait EnergySource {
  def getUnitsAvailable() : Long
  def getUsageCount() : Long
  def useEnergy(units : Long) : Unit
}
```

The implementation of this interface is a pretty direct translation of the class EnergySourceImpl from the Java version.

Download favoringIsolatedMutability/scala/typed1/EnergySourceImpl.scala

```
class EnergySourceImpl extends TypedActor with EnergySource {
  val MAXLEVEL = 100L
  var level = MAXLEVEL
  var usageCount = 0L

  def getUnitsAvailable() = level

  def getUsageCount() = usageCount

  def useEnergy(units : Long) = {
    if (units > 0 && level - units >= 0) {
      println("Thread in useEnergy: " + Thread.currentThread().getName())
      level -= units
    }
  }
}
```

Our Scala version of UseEnergySource will use spawn() to invoke the methods asynchronously.

Download favoringIsolatedMutability/scala/typed1/UseEnergySource.scala

```
object UseEnergySource {
  def main(args : Array[String]) : Unit = {
    println("Thread in main: " + Thread.currentThread().getName())

    val energySource = TypedActor.newInstance(
      classOf[EnergySource], classOf[EnergySourceImpl])

    println("Energy units " + energySource.getUnitsAvailable)

    Actor.spawn { energySource.useEnergy(10) }
    Actor.spawn { energySource.useEnergy(10) }

    Thread.sleep(1000);
    println("Energy units " + energySource.getUnitsAvailable)

    TypedActor.stop(energySource)
  }
}
```

Go ahead and run the Scala version and observe the output.

```
Thread in main: main
Energy units 100
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Thread in useEnergy: akka:event-driven:dispatcher:global-5
Energy units 80
```

The Scala version enjoyed a bit of conciseness in addition to the benefit of active objects.

8.8 Typed Actors and Murmurs

The typed actor version of the EnergySource allowed us to invoke methods but ran them as asynchronous messages in sequence under the covers, providing us thread-safety without the need for synchronization. That was easy to create, but our EnergySource is half-baked at this point, missing a key feature—the energy level needs to be replenished automatically.

In the versions we implemented in the previous chapters, the replenish action did not require any user intervention, it was done automatically in the background. As soon as you started the energy source, a timer took care of appropriately increasing the energy level one unit per second.

Implementing that feature in the typed actor version is going to take some effort—we need to ensure the single-threaded nature of our typed actor is not violated by the replenish operation. Let's explore some options before we jump into the code.

We could add the replenish() method to the EnergySource interface. The user of the energy source could call this method every second. Unfortunately this places an unnecessary burden on the user of the energy source, they could forget, and also it would no longer match in functionality with the other versions of EnergySource. Strike that option.

We could create a timer within the typed actor and this timer can periodically replenish the energy level. TypedActors provide a special method called preStart() that's called as soon as the actor is created and a method postStop() that's called right after an actor is stopped or shutdown. These two methods will serve well, we can start the timer in the preStart() and cancel it in the postStop() method. Seems like a good plan, but that opens up another problem.

The glitch is the timers run in their own threads and we don't want to touch the mutable state of our actor from those threads—remember we want the state to be isolated mutable, not shared mutable. What we need is a way to cause internal method calls—I call these murmurs—to be executed properly by the actor. These murmurs will not be visible to the outside user of our typed actor, but run as asynchronous messages, like those externally invoked messages, sequenced and interleaved with other messages. Let's figure out how to get that coded.

Remember typed actors are really actors with added convenience. They do receive messages just like actors do. The base class TypedActor's receive() method receives the messages from the proxies and dispatches the appropriate method on your class. You can override this method to implement a special message for the murmurs, the internal operations. That way users of your actor invoke methods published through the interface and your class internally can use this (unpublished) message. You could even take an extra step to ensure the sender of this message is your own actor, if you desire.

It's going to take some effort to get this done in Java but it's a lot easier in Scala. Let's first take a look at the Java implementation and then we'll look at the Scala version.

Implementing Murmurs in Java

While the external users of our EnergySourceImpl communicate through the EnergySource interface, internally we'll set the actor to send itself a Replenish request message every second using a timer. The replenish() method you'll see is private and can't be called directly from outside the class, but we'll refrain from calling it from the timer also. The timer will only send a message to the actor. Let's take a look at that part of the code.

Download favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
public class EnergySourceImpl extends TypedActor implements EnergySource {
  private final long MAXLEVEL = 100L;
  private long level = MAXLEVEL;
  private long usageCount = 0L;
  private final Timer replenishTimer = new Timer(true);

  class Replenish {}

  @Override public void preStart() {
    java.util.TimerTask timerTask = new java.util.TimerTask() {
      public void run() {
        System.out.println(
          "Thread in timer: " + Thread.currentThread().getName());
        ActorRef self = optionSelf().get();
        self.sendOneWay(new Replenish());
      }
    };

    replenishTimer.scheduleAtFixedRate(timerTask, 1000, 1000);
  }

  @Override public void postStop() { replenishTimer.cancel(); }
  private void replenish() {
    System.out.println("Thread in replenish " + Thread.currentThread());
    if (level < MAXLEVEL) level += 1;
  }
```

In the preStart() method, which is called right after the actor is started, we kickoff the timer. Each second, the timer sends a Replenish message to the actor. The timer is running within the actor, you know this refers to the Java instance, but we need a handle to the actor to send a message. We obtain it by calling the optionSelf() method on the instance. When the actor is stopped, we should stop the timer, and, so we have the postStop() method. In the private replenish() method, which we've not found a way to invoke yet, we increment the level.

The proxy used by the users of typed actors convert method calls to messages. The TypedActor base class's receive() method converts those messages to method calls on the implementation. If you examine the receive() method's signature, it returns a scala.PartialFunction. Think of a partial function, for our discussion here, as a glorified switch statement. It dispatches different pieces of code based on the type of the message received. So, the base class maps the messages to our methods and we want to take up an added responsibility of mapping an additional, but private, message. In other words, we want to combine our message handling with the efforts of the base class's receive() method. The orElse() method of PartialFunction allows us to easily do that, so we'll use that.

Download favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```
@Override public PartialFunction<Object, Object> receive() {
  return processMessage().orElse(super.receive());
}
```

We override the receive() method and, in it, combine the partial function returned by a, yet to be implemented, processMessage() method with the partial function returned by the base's receive(). Now we can turn our attention to implementing the processMessage() method. This method should receive a Replenish message and call the private replenish() method. Since this is part of the message handling sequence, we've taken care of thread synchronization using the actor based communication. Load up your cup if it's low on coffee, you're gonna need that extra shot of caffeine to implement this method.

PartialFunction is a trait in Scala and you already saw in Chapter 6, *Introduction to Software Transactional Memory*, on page 99 that traits with implementations manifest in Java as a pair of interface and an abstract class. So, to implement a trait in Java you'll implement that interface and delegate calls, as appropriate, to the corresponding abstract class.

The key method we'll implement is the apply() method where we handle the Replenish message. We'll also provide the implementation for the isDefined() which will tell if our PartialFunction supports a particular message format or type. Rest of the methods of this interface can be delegated. We can avoid implementing some of the methods of the interface by extending the AbstractFunction1 which shares a common interface Function1 with PartialFunction.

Download favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```java
private PartialFunction<Object, Object> processMessage() {
  class MyClass extends AbstractFunction1 implements PartialFunction {
    public Object apply(Object message) {
      if (message instanceof Replenish) replenish();
      return null;
    }

    public boolean isDefinedAt(Object message) {
      return message instanceof Replenish;
    }

    public Function1 lift() { return PartialFunction$class.lift(this); }

    public PartialFunction andThen(Function1 fn) {
      return PartialFunction$class.andThen(this, fn);
    }

    public PartialFunction orElse(PartialFunction fn) {
      return PartialFunction$class.orElse(this, fn);
    }
  };

  return new MyClass();
}
```

In the apply() method we checked if the message is of type Replenish and invoked the private replenish(). The isDefinedAt() indicated that we support only this one message type, leaving rest of the messages at the discretion of the base class's receive(). OK, the last step is not to forget the methods that have not changed from the previous typed actor version, so let's get them over.

Download favoringIsolatedMutability/java/typed2/EnergySourceImpl.java

```java
  public long getUnitsAvailable() { return level; }
  public long getUsageCount() { return usageCount; }

  public void useEnergy(long units) {
    if (units > 0 && level - units >= 0) {
      System.out.println(
        "Thread in useEnergy: " + Thread.currentThread().getName());
      level -= units;
    }
  }
}
```

There's no change to the EnergySource interface and the UseEnergySource continues to use the actor as before. So compile your new version of the EnergySourceImpl and run it with the UseEnergySource you wrote in the previous section. In the previous section, at the end of the run, you were left with 80 units of energy. Since you have the automatic replenish now working, the units should be higher by one or two points.

```
Thread in main: main
Energy units 100
Thread in useEnergy: akka:event-driven:dispatcher:global-2
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Thread in timer: Timer-0
Thread in replenish Thread[akka:event-driven:dispatcher:global-4,5,main]
Thread in timer: Timer-0
Thread in replenish Thread[akka:event-driven:dispatcher:global-5,5,main]
Energy units 82
```

The print statements we sprinkled to print the thread information shows that while the timer is running in its own thread, the replenish request runs in the actor's thread, thus once again relieving us of the synchronization concerns.

Implementing Murmurs in Scala

The essence of the approach was to combine your own partial function implementation with the partial function returned by the base class's receive() method. It was quite an effort in Java to do that, but is a lot easier in Scala. Let's look at the part that deals with handling the murmur, the internal message.

Download favoringIsolatedMutability/scala/typed2/EnergySourceImpl.scala

```
class EnergySourceImpl extends TypedActor with EnergySource {
  val MAXLEVEL = 100L
  var level = MAXLEVEL
  var usageCount = 0L
  val replenishTimer = new java.util.Timer(true)

  case class Replenish()
  override def preStart() = {
    val timerTask = new java.util.TimerTask {
      def run() = {
        println("Thread in timer: " + Thread.currentThread().getName())
        self ! Replenish
      }
    }
    replenishTimer.scheduleAtFixedRate(timerTask, 1000, 1000)
  }

  override def postStop() = { replenishTimer.cancel() }

  override def receive = processMessage orElse super.receive

  def processMessage : scala.PartialFunction[Any, Unit] = {
    case Replenish =>
      println("Thread in replenish " + Thread.currentThread())
      if (level < MAXLEVEL) level += 1
  }
```

The preStart() and the postStop() methods are simply translation from the Java version. The message class Replenish() became a case class here. The receive() is pretty much the same except for Scala conciseness. The biggest change is in the processMessage() method. It fizzled into a simple pattern matching for the Replenish message—no messing with inheritance and delegation like on the Java side.

Remember to bring in the code that implements the EnergySource interface/trait from the previous version.

Download favoringIsolatedMutability/scala/typed2/EnergySourceImpl.scala

```
  def getUnitsAvailable() = level

  def getUsageCount() = usageCount

  def useEnergy(units : Long) = {
    if (units > 0 && level - units >= 0) {
      println("Thread in useEnergy: " + Thread.currentThread().getName())
      level -= units
    }
  }
}
```

You can use the EnergySource and UseEnergySource from the previous Scala version. So compile the new version of EnergySourceImpl and run it to compare the output.

```
Thread in main: main
Energy units 100
```

```
Thread in useEnergy: akka:event-driven:dispatcher:global-3
Thread in useEnergy: akka:event-driven:dispatcher:global-5
Thread in timer: Timer-0
Thread in replenish Thread[akka:event-driven:dispatcher:global-6,5,main]
Thread in timer: Timer-0
Thread in replenish Thread[akka:event-driven:dispatcher:global-7,5,main]
Energy units 82
```

It was much easier to implement this in Scala compared to Java. Even though the two pieces of code produced same logical result, the Scala version was a lot less noisy compared to the Java version.

8.9 Mixing Actors and STM

Actors nicely allow you to isolate mutable state. They work great if your problem can be divided into concurrent tasks that can run independently and communicate with each other asynchronously using messages. Actors, however, don't provide a way to manage consensus across tasks. Suppose you want the actions of two or more actors to all succeed or fail collectively, that is either all of them succeed or all of them fail. Actors alone can't do that for you, but, you can achieve that by bringing STM into the mix. In this section I assume you've read Chapter 6, *Introduction to Software Transactional Memory*, on page 99 and the discussions on actors and typed actors in this chapter.

The AccountService class that helped us transfer between two Accounts in Section 4.5, *The Lock interface*, on page 80 and Section 6.9, *Creating Nested Transactions*, on page 121 would serve as a good example to understand this interplay of actors and STM. The operations of deposit and withdraw are isolated to individual accounts. So Account can be implemented using a simple actor or a typed actor. However, the operation of transfer has to coordinate a pair of deposit and withdraw across two accounts. In other words, the operation of deposit, handled by one actor, should succeed if-and-only-if the corresponding withdraw operation, handled by another actor as part of the overarching transfer, also succeeds. Let's mix actors and STM to implement the account transfer sample.

Akka provides a few options to mix actors and STM. You can create a separate transaction coordination object and manage the sequencing of the various actors into the transaction yourself (see Akka documentation if you want that level of control). Alternately, you can rely on one of two convenience ways to manage transactions among actors. We'll take a look at these two ways: how to use their *transactors* and how to *coordinate* typed actors.

8.10 Using Transactors

Akka transactors or transactional actors brings otherwise independent execution of actors within the folds of a transaction. The net effect of the coordinated actors on Ref objects are atomic. The changes to these objects are retained only if the encompassing transaction commits, the changes are otherwise discarded.

To create a transactor extend from Akka's UntypedTransactor in Java, Transactor class in Scala. This class in turn extends from the Actor class and overrides the receive() method. The Transactor's implementation of the receive() method decides if messages to the (trans)actor run standalone or coordinated.

By default messages to a transactor run coordinated in a transaction. If you want a message to be processed standalone or uncoordinated in any transaction, provide an optional normally() method. This method looks much like the receive() method you'd write in an actor. The messages you don't handle in your normally() method (or if you don't provide this method at all) run under their own transaction. So, a message Foo to a transactor is run under its own transaction if the transactor's normally() does not handle it and runs without a transaction if it's handled in normally(). If you want a transactor to run a message under a transaction, then wrap the message in a Coordinated message, like for example, Coordinated(Foo).

Transactors also provides handles to run some code before a transaction and after the transaction. Place the messages you want to run within a transaction into the required atomically() method. Again, this method looks much like the receive() and normally() methods. The difference is that you're assured that the messages within atomically() are run within a coordinated transaction. The Transactor class intercepts the Coordinated messages like Coordinated(Foo), performs the pre-transaction operation, sends the message (Foo for example) to atomically(), and carries out the post-transaction method if the message was handled successfully.

There's one last bit of details we need to discuss before we can dive into the example. Your transactor may want to bring in other actors under the folds of its own transaction. Again you can manually manage this within the atomically() method by getting access to the transaction Coordinated object or you may override the coordinate() method. In this method you can direct the other actors to run, within your coordinated transaction, either the same messages that you received or run some other messages. That's quite a bit of details, it's time to put those concepts into an example code.

Transactors in Java

To use transactors in Java you'll extend the UntypedTransactor class and implement the required atomically() method. In addition, if you want to include other actors in your transaction, you implement the coordinate() method. Let's first start with the message classes we'll need as shown next.

Download favoringIsolatedMutability/java/transactors/Deposit.java

```
public class Deposit {
  public final int amount;
  public Deposit(final int theAmount) { amount = theAmount; }
}
```

Download favoringIsolatedMutability/java/transactors/Withdraw.java

```
public class Withdraw {
  public final int amount;
  public Withdraw(final int theAmount) { amount = theAmount; }
}
```

Download favoringIsolatedMutability/java/transactors/FetchBalance.java

```
public class FetchBalance {}
```

Download favoringIsolatedMutability/java/transactors/Balance.java

```
public class Balance {
  public final int amount;
  public Balance(final int theBalance) { amount = theBalance; }
}
```

Download favoringIsolatedMutability/java/transactors/Transfer.java

```
public class Transfer {
  public final ActorRef from;
  public final ActorRef to;
  public final int amount;

  public Transfer(final ActorRef fromAccount,
    final ActorRef toAccount, final int theAmount) {
    from = fromAccount;
    to = toAccount;
    amount = theAmount;
  }
}
```

Balance will be used by the Account transactor as a response message to the FetchBalance request. This actor will also respond to the Deposit and Withdraw messages which carry the amount in them. The Transfer message is for the AccountService transactor to handle. Let's take a look at the Account transactor now.

Download favoringIsolatedMutability/java/transactors/Account.java

```
public class Account extends UntypedTransactor {
  private Ref<Integer> balance = new Ref<Integer>(0);

  public void atomically(final Object message) {
    if(message instanceof Deposit) {
      int amount = ((Deposit)(message)).amount;
      if (amount > 0) {
        balance.swap(balance.get() + amount);
        System.out.println("Received Deposit request " + amount);
      }
    }

    if(message instanceof Withdraw) {
      int amount = ((Withdraw)(message)).amount;
      System.out.println("Received Withdraw request " + amount);
      if (amount > 0 && balance.get() >= amount)
        balance.swap(balance.get() - amount);
      else {
        System.out.println("...insufficient funds...");
        throw new RuntimeException("Insufficient fund");
      }
    }

    if(message instanceof FetchBalance) {
      getContext().replySafe(new Balance(balance.get()));
    }
  }
}
```

The Account class extends UntypedTransactor and implements the atomically() method. This method will run under the context of a transaction provided to it—this may be the transaction of the caller or a separate transaction if none is provided. If the message received is the Deposit message, we increase the balance that's stored in the STM managed Ref object. If the message is Withdraw, then we decrease the balance only if sufficient money is available. Otherwise, we throw an exception. The exception, if thrown, will trigger a rollback of the encompassing transaction. Finally, if the message is FetchBalance, we respond back to the sender with the current value of balance. Since this entire method is running within one transaction, we don't have to worry about multiple accesses to the Ref object(s) within the transactor. The changes we make to the Ref objects will be retained only if the encompassing transaction is committed—remember you're responsible to maintain immutable state.

The job of coordinating the operation of deposit on one transactor (account) and withdraw on another transactor (account) belongs to the AccountService transactor which we will write next.

Download favoringIsolatedMutability/java/transactors/AccountService.java

```java
public class AccountService extends UntypedTransactor {

  @Override public Set<SendTo> coordinate(final Object message) {
    if(message instanceof Transfer) {
      Set coordinations = new java.util.HashSet<SendTo>();
      Transfer transfer = (Transfer) message;
      coordinations.add(sendTo(transfer.to, new Deposit(transfer.amount)));
      coordinations.add(sendTo(transfer.from, new Withdraw(transfer.amount)));
      return java.util.Collections.unmodifiableSet(coordinations);
    }

    return nobody();
  }

  public void atomically(final Object message) {}
}
```

Since the only responsibility of the AccountService is to coordinate the deposit and withdraw, in the coordinate() method we direct the appropriate messages to be sent to the two accounts. We do that by grouping actors and messages for each actor into a set. When we return this set from the coordinate() method, the AccountService transactor's base implementation will send the appropriate message to each transactor in the set. Once the messages are sent, it will call its own atomically() implementation. Since we have nothing more to do here we leave this method blank.

Let's exercise these transactors using a sample code.

Download favoringIsolatedMutability/java/transactors/UseAccountService.java

```java
public class UseAccountService {

  public static void printBalance(String accountName, ActorRef account) {
    Balance balance = (Balance)(account.sendRequestReply(new FetchBalance()));
    System.out.println(accountName + " balance is " + balance.amount);
  }
  public static void main(String[] args) throws InterruptedException {
    ActorRef account1 = Actors.actorOf(Account.class).start();
    ActorRef account2 = Actors.actorOf(Account.class).start();
    ActorRef accountService =
      Actors.actorOf(AccountService.class).start();

    account1.sendOneWay(new Deposit(1000));
    account2.sendOneWay(new Deposit(1000));

    Thread.sleep(1000);

    printBalance("Account1", account1);
    printBalance("Account2", account2);

    System.out.println("Let's transfer $20... should succeed");
    accountService.sendOneWay(new Transfer(account1, account2, 20));

    Thread.sleep(1000);

    printBalance("Account1", account1);
    printBalance("Account2", account2);

    System.out.println("Let's transfer $2000... should not succeed");
    accountService.sendOneWay(new Transfer(account1, account2, 2000));

    Thread.sleep(6000);

    printBalance("Account1", account1);
    printBalance("Account2", account2);

    Actors.registry().shutdownAll();
  }
}
```

There's no difference between how you interact with an actor and a transactor. If you send a regular message (like new Deposit(1000)), it is automatically wrapped into a transaction. You can also create your own coordinated transaction by creating an instance of akka.transactor.Coordinated and wrapping your message into it (like, for example, new Coordinated(new Deposit(1000))). Since you're dealing with only one-way messages in this example, you give sufficient time for the message to be handled before making the next query. This will give time for coordinating transaction to succeed or fail, and you can see the effect of the transaction in the subsequent call.

Since the coordinating transaction can commit only if all the messages to the participating transactors complete without exception, the coordinating request waits at most for the timeout period (configurable) for the transaction to complete. The output from the code is shown next.

Download favoringIsolatedMutability/java/transactors/output/UseAccountService.output

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Withdraw request 2000
...insufficient funds...
Received Deposit request 2000
Account1 balance is 980
Account2 balance is 1020
```

The first two deposits and the first transfer completed without a glitch. The last transfer, however, was for an amount larger than the balance available in the from account. The deposit part of the transfer was carried out as you can see from the printed message in the output (the deposits and withdraws run concurrently, so the order of the print may vary each time you run). However, the withdraw did not complete and so the entire transfer was rolled back. The change made in the deposit was discarded and the second transfer did not affect the balance of the two accounts.

Transactors in Scala

To use transactors in Scala you'll extend the Transactor class and implement the required atomically() method. In addition, if you want to include other actors in your transaction, you implement the coordinate() method. Let's translation the example from java to Scala. We will start with the message classes—these can be written concisely in Scala as case classes.

Download favoringIsolatedMutability/scala/transactors/Messages.scala

```scala
case class Deposit(val amount : Int)

case class Withdraw(val amount : Int)

case class FetchBalance()

case class Balance(val amount : Int)
case  class Transfer(val from : ActorRef, val to : ActorRef, val amount : Int)
```

Next let's translation the Account transactor to Scala. We can use pattern matching to handle the three messages.

Download favoringIsolatedMutability/scala/transactors/Account.scala

```
class Account extends Transactor {
  val balance = new Ref(0)

  def atomically = {
    case Deposit(amount) =>
      if (amount > 0) {
        balance.swap(balance.get() + amount)
        println("Received Deposit request " + amount)
      }

    case Withdraw(amount) =>
      println("Received Withdraw request " + amount)
      if (amount > 0 && balance.get() >= amount)
        balance.swap(balance.get() - amount)
      else {
        println("...insufficient funds...")
        throw new RuntimeException("Insufficient fund")
      }

    case FetchBalance =>
      self.replySafe(Balance(balance.get()))
  }
}
```

The next step is to translation the AccountService transactor. Again here we will leave the atomically() method blank and mention which objects coordinate to participate in the transaction within the coordinate() method. You will notice the syntax for that is a lot more concise here on the Scala side than in the Java code.

Download favoringIsolatedMutability/scala/transactors/AccountService.scala

```
class AccountService extends Transactor {

  override def coordinate = {
    case Transfer(from, to, amount) =>
      sendTo(to -> Deposit(amount), from -> Withdraw(amount))
  }

  def atomically = { case _ => }
}
```

Let's exercise these transactors using some sample code.

Download favoringIsolatedMutability/scala/transactors/UseAccountService.scala

```scala
object UseAccountService {

  def printBalance(accountName : String, account : ActorRef) = {
    (account !! FetchBalance) match {
      case Some(Balance(amount)) =>
        println(accountName + " balance is " + amount)
      case None =>
        println("Error getting balance for " + accountName)
    }
  }

  def main(args : Array[String]) = {
    val account1 = Actor.actorOf[Account].start()
    val account2 = Actor.actorOf[Account].start()
    val accountService = Actor.actorOf[AccountService].start()

    account1 ! Deposit(1000)
    account2 ! Deposit(1000)

    Thread.sleep(1000)

    printBalance("Account1", account1)
    printBalance("Account2", account2)

    println("Let's transfer $20... should succeed")
    accountService ! Transfer(account1, account2, 20)

    Thread.sleep(1000)

    printBalance("Account1", account1)
    printBalance("Account2", account2)

    println("Let's transfer $2000... should not succeed")
    accountService ! Transfer(account1, account2, 2000)

    Thread.sleep(6000)

    printBalance("Account1", account1)
    printBalance("Account2", account2)

    Actors.registry().shutdownAll()
  }
}
```

That was a direct translation from Java, again you see the areas where Scala conciseness plays a nice role. This version should behave just like the Java version when you observe the output.

Download favoringIsolatedMutability/scala/transactors/output/UseAccountService.output

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

Using Transactors

You saw how to implement transactors in Java and Scala. Transactors combine the benefits of actors and STM and allow you to provide consensus between otherwise independently running actors. Just like in the use of STMs, transactors are useful when the write collisions are highly infrequent. Ideally if multiple actors have to participate in a quorum or voting of some sought, transactors can be quite convenient.

8.11 Coordinating Typed Actors

Typed actors you saw in Section 8.7, *Using Typed Actors*, on page 201 is a glamor child of object-oriented programming and actor-based programming—it brings together the all-too-familiar, convenient method calls and the benefits of actors. So in an OO-application, instead of using actors you may be tempted to use typed actors. However, just like actors, typed actors run in isolation and don't provide transactional coordination—you'll have to use coordinating typed actors for that.

Akka makes it real simple to make a typed actor a coordinating typed actor, you simply have to mark your corresponding interface methods with a special Coordinated annotation. To indicate that a sequence of calls should run under a coordinated transaction, simply wrap the calls into a coordinate() method. This method by default waits for all the methods you invoked within to commit or rollback before proceeding further.

There is one limitation, only void methods can be marked with the special Coordinated annotation. Void methods translate to one way calls and these can participate in transactions. Methods that return values translate to two way blocking calls and so can't participate in free running concurrent transactions.

Let's implement the transfer of money sample that we implemented using transactors in Section 8.10, *Using Transactors*, on page 213.

Coordinating Typed Actors in Java

Typed actors use a pair of interface and implementations, so let's start with the two interfaces we'll need, the Account and AccountService interfaces.

Download favoringIsolatedMutability/java/coordtyped/Account.java

```
public interface Account {
  public int getBalance();
  @Coordinated public void deposit(int amount);
  @Coordinated public void withdraw(int amount);
}
```

Download favoringIsolatedMutability/java/coordtyped/AccountService.java

```
public interface AccountService {
  public void transfer(Account from, Account to, int amount);
}
```

The @Coordinated annotation marking is the only thing special in the Account interface. By marking the methods you indicated that these should run in either their own transaction or participate in the transaction of the caller. The method in AccountService is not marked with the annotation since the implementor of this class will be managing its own transaction. Let's take a look at the implementation of the Account interface now.

Download favoringIsolatedMutability/java/coordtyped/AccountImpl.java

```
public class AccountImpl extends TypedActor implements Account {
  private final Ref<Integer> balance = new Ref<Integer>(0);

  public int getBalance() { return balance.get(); }

  public void deposit(int amount) {
    if (amount > 0) {
      balance.swap(balance.get() + amount);
      System.out.println("Received Deposit request " + amount);
    }
  }
  public void withdraw(int amount) {
    System.out.println("Received Withdraw request " + amount);
    if (amount > 0 && balance.get() >= amount)
      balance.swap(balance.get() - amount);
    else {
      System.out.println("...insufficient funds...");
      throw new RuntimeException("Insufficient fund");
    }
  }
}
```

By extending TypedActor we indicated that this is an actor. Instead of using simple local fields, we're using the managed STM Refs. We don't write any code specific to transactions here, however, all the methods in this class will run within a transaction since the relevant interface methods are marked with @Coordinated. The deposit() increases the balance if amount is greater than 0. The withdraw() method decreases the balance if sufficient money is available. Otherwise it throws an exception indicating the failure of the operation and the encompassing transaction. The methods will participate in their own transaction if one is not provided. In the case of transfer, however, we want both the operations of deposit and withdraw to run in one transaction. This is managed by the AccountServiceImpl which we write next.

Download favoringIsolatedMutability/java/coordtyped/AccountServiceImpl.java

```
public class AccountServiceImpl extends TypedActor implements AccountService {
  public void transfer(final Account from, final Account to, final int amount) {
    Coordination.coordinate(true, new Atomically() {
      public void atomically() {
        to.deposit(amount);
        from.withdraw(amount);
      }
    });
  }
}
```

The transfer() method runs the two operations, deposit and withdraw, within a transaction. The code to run within a transaction is wrapped into the atomically() method which is a method of the Atomically interface. The coordinate() runs the transactional block of code in atomically(). The first parameter, true tells the method coordinate() to wait for its transaction to complete (succeed or rollback) before returning. The methods invoked within the transaction are one-way messages, however, and the coordinate() method does not block for their results, but only for their transactions to complete.

The code to use these typed actors should not look any different from code that uses regular objects, except for the creation of these objects. Rather than using new, you use a factory to create them as shown next.

Download favoringIsolatedMutability/java/coordtyped/UseAccountService.java

```
public class UseAccountService {

  public static void main(String[] args) throws InterruptedException {
    Account account1 = TypedActor.newInstance(Account.class, AccountImpl.class);
    Account account2 = TypedActor.newInstance(Account.class, AccountImpl.class);
    AccountService accountService =
      TypedActor.newInstance(AccountService.class, AccountServiceImpl.class);

    account1.deposit(1000);
    account2.deposit(1000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    System.out.println("Let's transfer $20... should succeed");

    accountService.transfer(account1, account2, 20);

    Thread.sleep(1000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    System.out.println("Let's transfer $2000... should not succeed");
    accountService.transfer(account1, account2, 2000);

    Thread.sleep(6000);

    System.out.println("Account1 balance is " + account1.getBalance());
    System.out.println("Account2 balance is " + account2.getBalance());

    Actors.registry().shutdownAll();
  }
}
```

The actions of the above code are the same as the example in Section 8.10, *Using Transactors*, on page 213. The output from this example is shown next.

```
Received Deposit request 1000
Account1 balance is 1000
Received Deposit request 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

As you'd expect, the output produced by the coordinating typed actor version is similar to the output produced by the transactor version—any changes made within a failing transaction are discarded.

Coordinating Typed Actors in Scala

Let's translation the Java version to Scala. Scala uses traits instead of interfaces, so that's the first difference you'll see.

Download favoringIsolatedMutability/scala/coordtyped/Account.scala

```
trait Account {
  def getBalance() : Int
  @Coordinated def deposit(amount : Int) : Unit
  @Coordinated def withdraw(amount : Int) : Unit
}
```

Download favoringIsolatedMutability/scala/coordtyped/AccountService.scala

```
trait AccountService {
  def transfer(from : Account, to : Account, amount : Int) : Unit
}
```

The implementation of the Account trait is straight forward translation from Java version.

Download favoringIsolatedMutability/scala/coordtyped/AccountImpl.scala

```
class AccountImpl extends TypedActor with Account {
  val balance = new Ref(0)

  def getBalance() = balance.get()

  def deposit(amount : Int) = {
    if (amount > 0) {
      balance.swap(balance.get() + amount)
      println("Received Deposit request " + amount)
    }
  }

  def withdraw(amount : Int) = {
    println("Received Withdraw request " + amount)
    if (amount > 0 && balance.get() >= amount)
```

```
      balance.swap(balance.get() - amount)
    else {
      println("...insufficient funds...")
      throw new RuntimeException("Insufficient fund")
    }
  }
}
```

Likewise, implementation of the AccountService is pretty simple too.

Download favoringIsolatedMutability/scala/coordtyped/AccountServiceImpl.scala

```
class AccountServiceImpl extends TypedActor with AccountService {
  def transfer(from : Account, to : Account, amount : Int) = {
    coordinate {
      to.deposit(amount)
      from.withdraw(amount)
    }
  }
}
```

By default the coordinate() method waits for its transaction to complete (succeed or rollback) before returning. The methods invoked within the transaction are one-way messages, however, and the coordinate() method does not block for their results, but only for their transactions to complete. You may pass an optional parameter like coordinate(wait = false) {...} to tell the method not to wait for the transaction to complete.

Finally let's write the sample code to use the above code.

Download favoringIsolatedMutability/scala/coordtyped/UseAccountService.scala

```
object UseAccountService {

  def main(args : Array[String]) = {
    val account1 = TypedActor.newInstance(classOf[Account], classOf[AccountImpl])
    val account2 = TypedActor.newInstance(classOf[Account], classOf[AccountImpl])
    val accountService =
      TypedActor.newInstance(classOf[AccountService], classOf[AccountServiceImpl])

    account1.deposit(1000)
    account2.deposit(1000)

    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())


    println("Let's transfer $20... should succeed")

    accountService.transfer(account1, account2, 20)

    Thread.sleep(1000)
    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())

    println("Let's transfer $2000... should not succeed")
    accountService.transfer(account1, account2, 2000)

    Thread.sleep(6000)

    println("Account1 balance is " + account1.getBalance())
    println("Account2 balance is " + account2.getBalance())

    Actors.registry().shutdownAll()
  }
}
```

The Scala version should behave just like the Java version.

```
Received Deposit request 1000
Received Deposit request 1000
Account1 balance is 1000
Account2 balance is 1000
Let's transfer $20... should succeed
Received Deposit request 20
Received Withdraw request 20
Account1 balance is 980
Account2 balance is 1020
Let's transfer $2000... should not succeed
Received Deposit request 2000
Received Withdraw request 2000
...insufficient funds...
Account1 balance is 980
Account2 balance is 1020
```

8.12 Limitations of Actor Based Model

The actor based model makes it easy to program with isolated mutability, but, it does have some limitations.

Actors communicate with each other through messages. In languages that do not enforce immutability, you must be very careful to ensure the messages are immutable. Passing mutable messages can lead to thread-safety concerns and eventually the perils of shared mutability. Tools may evolve to verify that the messages are immutable, until then the onus is on us to ensure immutability.

Actors run asynchronously but coordinate by passing messages. Unexpected failure of actors may result in livelocks—one or more actors may be waiting for messages that would never arrive due to the failure. Program defensively and handle exceptions within your actors and propagate error messages to actors awaiting response.

Not all applications are well suited for actor based model. Actors serve well when you can divide the problem into parts that can run quite independently and need to communicate only sporadically. If frequent interaction is required or the parts or tasks need to coordinate to form a quorum, the actor based model is not suitable. You may have to mix other concurrency models or consider significant redesign.

8.13 Recap

Actor based model relieves you of the synchronization concerns and offers quite a convenient way to manage isolated mutability. You saw how different problems can be solved using coordinating actors. You also saw how to mix actors with transactions where synchronous coordination between otherwise isolated actors is needed. You saw examples in Java and Scala in this chapter. If you're programming in Groovy or JRuby, you'll learn how to use actors in these languages in the next chapter.

In business, as in war, surprise is worth as much as force
► Paul Graham

Chapter 9
Actors in Clojure, Groovy, Java, JRuby, and Scala

You've saw actors in action and if you wonder how to use them from your favorite language we'll learn that in this chapter. Sending messages to actors does not seem like a big deal, the real concern is around creating actors—this involves extending from classes and overriding methods. There are some minor issues you'll run into, but nothing someone who has survived this far into this book can't handle.

In this chapter you'll see how to create and use actors in different JVM languages. Focus on the sections related to languages you're interested in and feel free to skip the others.

9.1 Clojure Agents

Clojure does not have direct support for Actors, but it has agents. Agents, unlike actors, don't have any designated receives. Also, unlike actors, each agent is tied to a single identity. You can send functions to agents and these are run, one at a time, in a thread from the thread pool. You use these functions to effect change to the mutable identity. You can directly read or dereference the values held by an agent, you don't need to send messages for that.

You create agents much like how you created refs. However, while Clojure refs are used for synchronized coordinated change to multiple mutable identities, agents provide asynchronous coordinated change to single identity. The state you return from the scheduled function becomes the new state of the agent. This new state is used for subsequent reads or functions that are sent to the agent.

The functions you schedule on an agent are run sequentially using threads from a thread pool. So, if you schedule multiple functions, they're run in sequence while the send itself is non-blocking. If you expect a method to be mostly CPU bound then use the send() method to schedule a function on an agent. If IO blocking is possible within the function, then use send-off() instead of send() to prevent agents starving for threads while some agents are blocked.

Agents run in non-daemon threads, so you've to be careful to shutdown the agents when you're done—otherwise your JVM won't quit.

Let's see agents in action with an example—create a hollywood actor and ask him to play different roles. You'll schedule multiple roles to play concurrently on different actors. While the agents run concurrently, you'll see that multiple messages to a single agent are run sequentially.

Let's first create a function that will ask an actor to play a role and add the role to the list of roles played. In the code shown next, act() method receives two parameters actor and role. The actor is a tuple with two elements, the name of the actor and a list of roles played. We print the actor's name and the given role. Then after a simulated delay to show the sequencing of concurrent calls to the same agent we return the new state for the actor, the name and a new list of roles with the new role conjured in. We did not modify the actor itself, the new state we return will become the state of the actor as you'll see soon.

```
(defn act [actor role]
  (let [name (first actor) roles (second actor)]
    (println name " playing role of " role)
    (. Thread sleep 2000)
    [name (conj roles role)]))
```

You may wondering where the agent is, so let's get to that now. Agents are quite simple, they don't need any fanfare. In the code shown next we define two agents, each representing a famous Hollywood actor. All that these agents contain is a tuple with name of the actor and an empty list. You use the keyword **agent** to define them.

```
(try
  (def depp (agent ["Johny Depp" ()]))
  (def hanks (agent ["Tom Hanks" ()]))
```

Our agents are geared up to receive messages, so let's send a few messages to them, asking the two actors to act some roles as shown next.

The send() method takes the agent as the first parameter followed by the function to invoke. The first parameter this function takes is the agent that preceded it in this call list. If the function takes additional parameters, like role in this example, place it after the function name.

```
(send depp act "Wonka")
(send depp act "Sparrow")
(send hanks act "Lovell")
(send hanks act "Gump")
```

The call to send() is non-blocking. If you want to wait for the agent to respond to all messages sent from your current thread, you can use the await() or the civilized version of it await-for() which accepts a timeout as shown next. The await-for() takes the timeout in milliseconds as the first parameter followed by any number of agents you'd like to wait for. You break-off from this blocking call when the agents complete your calls or the timeout occurs, whichever comes first.

```
(println "Let's wait for them to play")
(await-for 4000 depp hanks)
```

To dereference the agents, simply prefix the agent with an @ symbol as shown next.

```
(println "Let's see the net effect")
(println (first @depp) " played " (second @depp))
(println (first @hanks) " played " (second @hanks))
```

One last details to remember, you must terminate the agents since they run in non-daemon threads. Remember to perform this operation within the **finally** block to ensure release even in the face of exceptions as shown next.

```
(finally (shutdown-agents)))
```

Let's put all that together so you can see the complete sequence of code.

Download polyglotActors/clojure/clojureagent.clj

```
(defn act [actor role]
  (let [name (first actor) roles (second actor)]
    (println name " playing role of " role)
    (. Thread sleep 2000)
    [name (conj roles role)]))

(try
  (def depp (agent ["Johny Depp" ()]))
  (def hanks (agent ["Tom Hanks" ()]))

  (send depp act "Wonka")
  (send depp act "Sparrow")
  (send hanks act "Lovell")
  (send hanks act "Gump")

  (println "Let's wait for them to play")
  (await-for 4000 depp hanks)

  (println "Let's see the net effect")
  (println (first @depp) " played " (second @depp))
  (println (first @hanks) " played " (second @hanks))

  (finally (shutdown-agents)))
```

Go ahead, run it and you'll see the output similar to the one shown next. The sequence you see may slightly vary since the agents are running concurrently and the exact ordering is non-deterministic.

```
Johny Depp  playing role of  Wonka
Tom Hanks  playing role of  Lovell
Let's wait for them to play
Johny Depp  playing role of  Sparrow
Tom Hanks  playing role of  Gump
Let's see the net effect
Johny Depp  played  (Sparrow Wonka)
Tom Hanks  played  (Gump Lovell)
```

It's clear from the output that the calls to send() are non-blocking. Each of the agents executed the first message sent to them. The second messages to the agents are delayed due to the induced delay in the first messages. The main thread, in the mean time, awaits for the agents to handle these messages. When done, the roles played by the actors represented by these agents are printed.

Let's compare the Clojure agents with Akka actors. Agents don't have any specific methods or messages that they respond to. You can schedule arbitrary functions to run on agents. In contrast, Akka actors have designated onReceive() or receive() methods and have predefined messages that they respond to. The sender of the messages have to know which messages are supported. In comparison, both actors and agents serialize calls to them, allowing only one thread to run on them at any time.

By default agents don't run in any transaction. Recall that transactional code should have no side-effects since they can be rolled back and retried several times. What if you do want to perform some actions or send messages to agents from within these transactions? Clojure tactfully handles this by grouping all the message sends to agents within a transaction and dispatching them only if the transaction commits.

Let's send messages to an agent from within transactions to see this behavior. In the code shown next, the sendMessageInTXN() method will send an act message to the given actor to play a given role. This message is sent from within a transaction delineated by the dosync() method. The transaction is forced to fail if the shouldFail parameter is true. If the transaction fails the message send will be discarded; it will be dispatched otherwise. We create an agent depp and send a transactional message to play the role "Wonka" first. Then, within a failed transaction, you send message to play the role "Sparrow" next. Finally, you wait for the messages to be processed and then print the roles played by depp.

```
(defn act [actor role]
  (let [name (first actor) roles (second actor)]
    (println name " playing role of " role)
    [name (conj roles role)]))

(defn sendMessageInTXN [actor role shouldFail]
  (try
    (dosync
      (println "sending message to act " role)
      (send actor act role)
      (if shouldFail (throw (new RuntimeException "Failing transaction"))))
    (catch Exception e (println "Failed transaction"))))

(def depp (agent ["Johny Depp" ()]))

(try
  (sendMessageInTXN depp "Wonka" false)
  (sendMessageInTXN depp "Sparrow" true)
  (await-for 1000 depp)
  (println "Roles played " (second @depp))
  (finally (shutdown-agents)))
```

If it worked as expected you should see the first message being delivered to the agent, but the second message should be discarded as shown in the output.

```
sending message to act  Wonka
Johny Depp  playing role of  Wonka
sending message to act  Sparrow
Failed transaction
Roles played  (Wonka)
```

When bound by a transaction, messages sent to agents are grouped and dispatched only when the transaction succeeds. You can contrast this with the Akka transactors where you mixed actors with transactions.

The Clojure solution is very concise and does not have some of the drawbacks of transactors we discussed earlier.

9.2 Actors in Groovy with GPars

To create actors you have at least as many options in Groovy as in Java.

You can implements Akka actors in Groovy by extending Akka UntypedActor if you like. Groovy allows you to extend from classes written in other JVM languages, so it is pretty straightforward to override the onReceive() method of UntypedActor. Within this method, you can check for the type of message and perform appropriate actions.

You may also use one of the other actor frameworks available for Java. However, you may want to consider a concurrency library geared more towards Groovy—GPars.

GPars is a library written in Java that brings the love of programming concurrency to Groovy, and Java. The performance of Java and the sheer elegance of Groovy shines in GPars. It provides a number of capabilities—asynchrony, parallel array manipulations, fork/join, agents, dataflow, and actors. I'll not discuss GPars in its entirety here. My goal is to show how you can implement some of the examples you saw earlier using GPars actors. For a comprehensive discussion on GPars refer to GPars documentation.

First, the actors in GPars take care of the fundamentals, they cross the memory barrier at appropriate times, allow for only one thread to run their receiver methods at any given time, and are backed by message queues. GPars has different flavors of actors and allow you to configure various parameters like fairness of thread affinity. The API for creating and using actors is fluent as we'll see next.

Creating Actors

To create an actor in GPars, extend from DefaultActor or simply pass a closure to the actor() method of Actors. Use the latter if you have a simple message loop. If you have more complex logic or want to call into other methods of your actor class, take the route of extending DefaultActor.

You must start an actor before it can accept messages, any attempt to send a message before that will result in an exception. Let's create an actor first using the DefaultActor and then by using the actor() method.

Let's create an actor that will receive a String message and respond by simply printing the message. Extend DefaultActor and implement its act() method as shown in the next code. This method runs the message loop, so invoke a never ending loop() method and call the react() to receive a message. When your actor runs into react(), it blocks without holding a thread hostage. When a message arrives, a thread from the thread pool is scheduled to run the closure you provide to the react() method. Once the thread completes the block of code provided to react(), it reruns the containing loop(), placing the actor again on the message wait queue. In the example we simply print the message received plus the information on the thread that's processing the message.

Download polyglotActors/groovy/create/createActor.groovy

```
class HollywoodActor extends DefaultActor {
  def name

  public HollywoodActor(actorName) { name = actorName }

  void act() {
    loop {
      react { role ->
        println "$name playing the role $role"
        println "$name runs in ${Thread.currentThread()}"
      }
    }
  }
}
```

You create an instance of an actor much like how you create an instance of any class, but remember to call the start() method. Let's create a pair of actors.

Download polyglotActors/groovy/create/createActor.groovy

```
depp = new HollywoodActor("Johny Depp").start()
hanks = new HollywoodActor("Tom Hanks").start()
```

The actors are now ready to receive messages, so let's send a message to each actor using the send() method.

Download polyglotActors/groovy/create/createActor.groovy

```
depp.send("Wonka")
hanks.send("Lovell")
```

Instead of using the traditional Java style method call, you can also use an overloaded operator << to send a message. If that's too much noise for you, simply place the message next to the actor as in the next example.

Download polyglotActors/groovy/create/createActor.groovy

```
depp << "Sparrow"
hanks "Gump"
```

The actors, by default, run in daemon threads, so the code will terminate as soon as main completes. We want it to live long enough for you to see the messages being processed, so we can add a delay to the end or use the join() method with a timeout. The join() will wait for the actors to terminate, but since we did not use the terminate() call, they will not die, but the timeout will get the main thread out of the blocking call.

Download polyglotActors/groovy/create/createActor.groovy

```
[depp, hanks]*.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

Let's take this example for a ride, run groovy and provide the GPars jar in the classpath, like so,

```
groovy -classpath $GPARS_HOME/gpars-0.11.jar createActor.groovy
```

The actors should respond to the messages by printing the role we asked them to play. The two actors run concurrently, but each of them process only one message at a time. At different instance, different threads from the thread pool may run the actors' method in response to the messages send. Each time you run the code, you may see a slightly different sequence of calls in the output due to the non-deterministic nature of concurrent executions. Coincidentally, in one of the output I managed to capture, you see two actors share the same thread, at different instances, and also an actor switching threads between messages.

```
Johny Depp playing the role Wonka
Johny Depp runs in Thread[Actor Thread 3,5,main]
Tom Hanks playing the role Lovell
Tom Hanks runs in Thread[Actor Thread 3,5,main]
Tom Hanks playing the role Gump
Tom Hanks runs in Thread[Actor Thread 2,5,main]
Johny Depp playing the role Sparrow
Johny Depp runs in Thread[Actor Thread 3,5,main]
```

If you want to process messages indefinitely, or until an actor is terminated, use the loop() as you saw in the previous example. If you want to run the loop only a finite number of times or until some condition is met, you can use variations of the loop() method that gives you more control. To run for a finite number of times use loop(count) {}. If you'd like to specify a condition use loop ({-> expression }) and the actor will repeat the loop as long as the expression is true. You can also provide an optional closure to run when the actor terminates as you'll see in the next example.

Let's create an actor this time using the Actors' act() method. Call the act() method and send the message loop to it as a closure. In the previous example we used loop() with only the closure—the message processing body with call to react(). In this example we pass two additional parameters to loop(). The first parameter can be a condition or a count. Here we pass 3 to tell the actor to respond to three messages and then terminate. The second parameter, which is optional, is a piece of code (given as a closure) that's executed when the actor terminates.

Download polyglotActors/groovy/create/loop3.groovy

```
depp = Actors.actor {
  loop(3, { println "Done acting" }) {
    react { println "Johny Depp playing the role $it" }
  }
}
```

The actor is programmed to receive only three messages. Let's send one additional message than it's expecting to see what happens.

Download polyglotActors/groovy/create/loop3.groovy

```
depp << "Sparrow"
depp << "Wonka"
depp << "Scissorhands"
depp << "Cesar"

depp.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

When you run the previous example, you'll see the actor received three messages and then terminated. The last message to the actor is ignored as the actor has already terminated by this point.

```
Johny Depp playing the role Sparrow
Johny Depp playing the role Wonka
Johny Depp playing the role Scissorhands
Done acting
```

Sending and Receiving Messages

In addition to one way messages, GPars allows you to send messages and receive responses as well. Yon can make a blocking sendAndWait() with an optional timeout or you can use the non-blocking sendAndContinue() which accepts a closure that is be called when the reply arrives.

This non-blocking version will make it easier for you to message multiple actors and then wait for their responses.

To facilitate sending a reply back to the message sender, the Actor provides a convenient sender property. Call the send() method on the sender to send a reply as in the next example.

Download polyglotActors/groovy/create/fortune.groovy

```
fortuneTeller = Actors.actor {
  loop {
    react { name ->
      sender.send("$name, you have a bright future")
    }
  }
}
```

The fortuneTeller actor receives a name and replies a lame fortune message. From the user of the actor point of view, you'd call the sendAndWait() to send a name and block for a response.

Download polyglotActors/groovy/create/fortune.groovy

```
message = fortuneTeller.sendAndWait("Joe", 1, TimeUnit.SECONDS)
println message
```

While it's simple, the sendAndWait() won't help if you want to send multiple messages at once. You can send multiple messages using the non-blocking sendAndContinue() but once you send the messages, you need a way to block for all the responses to arrive. You could use the CountDownLatch for that. When replies arrive, you can count down on that latch. The main thread which blocks on the latch can continue when all responses arrive. If the timeout happens before that, you can handle that case from the return value of the await() method.

Download polyglotActors/groovy/create/fortune.groovy

```
latch = new CountDownLatch(2)

fortuneTeller.sendAndContinue("Bob") { println it; latch.countDown() }
fortuneTeller.sendAndContinue("Fred") { println it; latch.countDown() }

println "Bob and Fred are keeping their fingers crossed"

if (!latch.await(1, TimeUnit.SECONDS))
  println "Fortune teller didn't respond before timeout!"
else
  println "Bob and Fred are happy campers"
```

The output from a run of the example is shown next.

```
Joe, you have a bright future
```

```
Bob, you have a bright future
Bob and Fred are keeping their fingers crossed
Fred, you have a bright future
Bob and Fred are happy campers
```

Handling Discrete Messages

Messages are not restricted to simple String used in the GPars examples so far. You can send any object as message, but ensure they're immutable. Let's create an example actor that mimics stock trading, it would handle a couple of different types of messages. But first we need to define the messages. You can make use of the Groovy defined annotation Immutable to ensure the message is immutable. This annotation not only ensures all fields are final, it also throws in a constructor (plus a few more methods) for you.

Download polyglotActors/groovy/multimessage/stock1.groovy

```groovy
@Immutable class LookUp {
  String ticker
}

@Immutable class Buy {
  String ticker
  int quantity
}
```

The LookUp holds a ticker symbol and represents a message that requests price for a stock. The Buy holds a ticker and the number of shares you'd like to buy. Our actor needs to take different actions for these two messages, let's get to that now.

Download polyglotActors/groovy/multimessage/stock1.groovy

```groovy
trader = Actors.actor {
  loop {
    react { message ->
        if(message instanceof Buy)
          println "Buying ${message.quantity} shares of ${message.ticker}"

        if(message instanceof LookUp)
          sender.send((int)(Math.random() * 1000))
    }
  }
}
```

When the actor receives a message you check if the message is an instance of one of the message types you expect using runtime type identification (RTTI) **instanceof**. You take the necessary action based on the message type, or ignore the message if it's something you didn't expect; alternately you could throw exceptions if you receive an unrecognized message. In this example, if the message was a Buy you simply print a message and if it was a LookUp you return a random number representing price of the stock.

There's nothing special in using an actor that takes different types of messages as you can see here.

Download polyglotActors/groovy/multimessage/stock1.groovy

```
trader.sendAndContinue(new LookUp("XYZ")) {
  println "Price of XYZ sock is $it"
}

trader << new Buy("XYZ", 200)

trader.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

The output from the code is shown next.

```
Price of XYZ sock is 27
Buying 200 shares of XYZ
```

It was simple, but the way your actor handled the message probably left you desiring for something better. Rather than using conditional statements to check the type, you can leverage Groovy's typing system to do the branching. For this you'd use DynamicDispatchActor as base actor class as shown here.

Download polyglotActors/groovy/multimessage/stock2.groovy

```
class Trader extends DynamicDispatchActor {
  void onMessage(Buy message) {
    println "Buying ${message.quantity} shares of ${message.ticker}"
  }

  def onMessage(LookUp message) {
    sender.send((int)(Math.random() * 1000))
  }
}
```

We extended the actor from DynamicDispatchActor and overloaded the onMessage() method for each type of message our actor handles. The base class DynamicDispatchActor will do the checking of message type and dispatch the appropriate onMessage() of the actor. Since we defined a separate class, we must remember to call start() to initialize the actor message loop as you see next.

Download polyglotActors/groovy/multimessage/stock2.groovy

```
trader = new Trader().start()
```

```
trader.sendAndContinue(new LookUp("XYZ")) {
  println "Price of XYZ sock is $it"
}

trader << new Buy("XYZ", 200)

trader.join(1, java.util.concurrent.TimeUnit.SECONDS)
```

To provide discrete message handlers you're not forced to create a separate class. Instead you can pass a closure that handles a chain of methods to the constructor of DynamicDispatchActor as you see next.

Download polyglotActors/groovy/multimessage/stock3.groovy

```
trader = new DynamicDispatchActor({
  when { Buy message ->
    println "Buying ${message.quantity} shares of ${message.ticker}"
  }

  when { LookUp message ->
    sender.send((int)(Math.random() * 1000))
  }
}).start()
```

The closure that you send as constructor parameter has a nice fluent syntax where each type of message is presented in a **when** clause. At first glance this appears much like the conditional if-statements but there are some significant differences. First, you don't have the bad taste of RTTI. Second, each of these when clauses transform into a specific message handler under the covers. Invalid or unhandled messages result in exceptions, whereas in the case of explicit conditional statements you had to write code to handle that.

Using GPars

You got a hang of creating and coordinating GPars actors. Let's put that knowledge to use to rewrite the file size program. We discussed an actor based design for that program in Section 8.6, *Coordinating Actors*, on page 193 and you saw the implementations using Scala and Java before. Let's see how the Groovy and GPars implementation of that design compares.

We need to define the message types, so let's start with that.

Download polyglotActors/groovy/filesize/findFileSize.groovy

```
@Immutable class RequestAFile {}
@Immutable class FileSize { long size }
@Immutable class FileToProcess { String fileName }
@Immutable class FetchTotalSize { String fileName }
```

In addition to the custom messages RequestAFile, FetchTotalSize, and FileSize, you'll also use the java.io.File and java.util.List of files as messages between the actors. The file size program design asked for two actors, the SizeCollector and the FileProcessor. Each FileProcessor will receive as message a directory and send as message the size of files and list of subdirectories in that directory. A DefaultActor will serve that purpose. This actor needs to register with the SizeCollector actor for receiving the directories. This is done in the afterStart() method. GPars provides event methods like afterStart() and afterStop() just like Akka preStart() and postStop() methods. The afterStart() is called right after the actor is started but before the first message is processed. Let's implement the FileProcessor actor.

Download polyglotActors/groovy/filesize/findFileSize.groovy

```
class FileProcessor extends DefaultActor {
  def sizeCollector

  public FileProcessor(theSizeCollector) { sizeCollector = theSizeCollector }

  void afterStart() { registerToGetFile() }

  void registerToGetFile() { sizeCollector << new RequestAFile() }

  void act()  {
    loop {
      react { message ->
        def file = new File(message.fileName)
        def size = 0
        if(!file.isDirectory())
          size = file.length()
        else {
          def children = file.listFiles()
          if (children != null) {
            children.each { child ->
              if(child.isFile())
                size += child.length()
              else
                sizeCollector << new FileToProcess(child.getPath())
            }
          }
        }
        sizeCollector << new FileSize(size)
        registerToGetFile()
      }
    }
  }
}
```

The SizeCollector will receive four different types of messages as it has to communicate with the FileProcessors and the main thread as well. The DynamicDispatchActor comes in handy for this purpose so let's use that. Just like the previous implementations of this design, you maintain a list of files to process, a list of file processors, a reference to the main sender, a count of files pending to be visited, and the most important total file size being computed. Each of the message types are handled by one of the onMessage() overloaded methods. As files are received and file requests arrive, you dispatch them to willing and able FileProcessors.

Download polyglotActors/groovy/filesize/findFileSize.groovy

```
class SizeCollector extends DynamicDispatchActor {
  def toProcessFileNames = []
  def fileProcessors = []
  def fetchSizeSender
  def pendingNumberOfFilesToVisit = 0
  def totalSize = 0L

  def sendAFileToProcess() {
    if(!toProcessFileNames.isEmpty() && !fileProcessors.isEmpty()) {
      fileProcessors.first() << new FileToProcess(toProcessFileNames.first())
      fileProcessors = fileProcessors.tail()
      toProcessFileNames = toProcessFileNames.tail()
    }
  }

  void onMessage(FetchTotalSize message) {
    fetchSizeSender = sender
    toProcessFileNames.add(message.fileName)
    pendingNumberOfFilesToVisit = 1
    sendAFileToProcess()
  }

  void onMessage(RequestAFile message) {
    fileProcessors.add(sender)
    sendAFileToProcess()
  }

  void onMessage(FileToProcess message) {
    toProcessFileNames.add(message.fileName)
    pendingNumberOfFilesToVisit += 1
    sendAFileToProcess()
  }

  void onMessage(FileSize message) {
    totalSize += message.size
    pendingNumberOfFilesToVisit -= 1
    if(pendingNumberOfFilesToVisit == 0) fetchSizeSender << totalSize
  }
}
```

It's time to put these actors to use and that's the job of the main code.

Download polyglotActors/groovy/filesize/findFileSize.groovy

```
start = System.nanoTime()
sizeCollector = new SizeCollector().start()

100.times { new FileProcessor(sizeCollector).start() }

totalSize = sizeCollector.sendAndWait(
  new FetchTotalSize(args[0]), 100, TimeUnit.SECONDS)

end = System.nanoTime()
println "Total size is $totalSize"
println "Time taken is ${(end - start)/1.0e9}"
```

The main code sets one SizeCollector and a hundred FileProcessors in motion and waits for a response from the SizeCollector. The final result and the time taken is printed at the end. The output from running the code for the /usr directory is shown next.

```
Total size is 3793911517
Time taken is 8.69052900
```

The file size reported by the Groovy-GPars version is the same as the result from the other versions. The time taken is pretty comparable as well. The Groovy version enjoys comparable level of conciseness in syntax with the Scala version as well. Overall that's pretty groovy.

9.3 Java Integration

You have a handful of choices for actor based concurrency in Java as we discussed earlier. You get to pick from ActorFoundary, Actorom, Actors Guild, Akka, FunctionalJava, Kilim, Jetlang,... We used Akka fairly extensively in Chapter 8, *Favoring Isolated Mutability*, on page 172. Akka was written in Scala, but they've done a fairly decent job of exposing convenient interface for you to use in Java. If you'd like to use Akka refer to Akka documentation and the examples given in this book. If you'd like to use one of the other libraries, refer to their respective documentation. You'll have to learn about their different APIs, however, conceptually they're similar to what Akka provides.

9.4 JRuby Akka integration

To use Akka in JRuby, you'll follow along the same lines as you did in Java. The main difference is you'll be able to benefit from JRuby conciseness. You're already familiar with the capabilities of Akka and its powerful API from Chapter 8, *Favoring Isolated Mutability*, on page 172. Let's jump right into implementing the design in Section 8.6, *Coordinating Actors*, on page 193 for the file size program in JRuby. Recollect you need a bunch of message types, two actor types, and a main code to exercise these. Let's start with the four message types first.

Download polyglotActors/jruby/FileSize.rb

```ruby
class RequestAFile; end

class FileSize
  attr_reader :size
  def initialize(size)
    @size = size
  end
end

class FileToProcess
  attr_reader :file_name
  def initialize(file_name)
    @file_name = file_name
  end
end

class FetchTotalSize
  attr_reader :file_name
  def initialize(file_name)
    @file_name = file_name
  end
end
```

These message representing classes are a direct translation of the respective classes from Java version you saw earlier. Writing the FileProcessor actor involves a little bit more effort. When you extend a Java class, JRuby does not allow you to change the constructor signature. The Akka actor base class UntypedActor has a no-argument constructor, but your FileProcessor needs to accept a parameter for the SizeCollector. If you proceed to write an initialize() method in this class, you'll run into runtime errors. Don't battle that, instead take a simple workaround. Write a class (static) method create_new() that will accept the parameter you'd like to create an instance with. Within this method, use the instance_eval() method to set the field in the context of the newly created instance. Except for that, the rest of this class is a simple translation of the Java code to JRuby.

Download polyglotActors/jruby/FileSize.rb

```
require 'java'
java_import java.lang.System
java_import 'akka.actor.ActorRegistry'
java_import 'akka.actor.Actors'
java_import 'akka.actor.UntypedActor'

class FileProcessor < UntypedActor
  attr_accessor :size_collector

  def self.create_new(size_collector)
    instance = self.new
    instance.instance_eval { @size_collector = size_collector }
    instance
  end

  def preStart
    register_to_get_file
  end

  def register_to_get_file
    @size_collector.send_one_way(RequestAFile.new, self.self)
  end

  def onReceive(fileToProcess)
    file = java.io.File.new(fileToProcess.file_name)
    size = 0
    if !file.isDirectory()
      size = file.length()
    else
      children = file.listFiles()
      if children != nil
        children.each do |child|
          if child.isFile()
            size += child.length()
          else
            @size_collector.send_one_way(FileToProcess.new(child.getPath()))
          end
        end
      end
    end
    @size_collector.send_one_way(FileSize.new(size))
    register_to_get_file
  end
end
```

Next let's get the second actor SizeCollector ready. When you create an instance of this actor using one of the versions of UntypedActor's actor_of() method, you'll get an error that this class lacks the create() method. So, let's go ahead and provide that factory method. Except for this addition, rest of the code is a direct translation of Java version to JRuby.

Download polyglotActors/jruby/FileSize.rb

```
class SizeCollector < UntypedActor
  def self.create(*args)
    self.new(*args)
  end

  def initialize
    @to_process_file_names = []
    @file_processors = []
    @fetch_size_future = nil
    @pending_number_of_files_to_visit = 0
    @total_size = 0
  end

  def send_a_file_to_process
    if !@to_process_file_names.empty? && !@file_processors.empty?
      @file_processors.first.send_one_way(
        FileToProcess.new(@to_process_file_names.first))
      @file_processors = @file_processors.drop(1)
      @to_process_file_names = @to_process_file_names.drop(1)
    end
  end

  def onReceive(message)
    if message.kind_of? FetchTotalSize
      @fetch_size_future = self.self.getSenderFuture
      @to_process_file_names << message.file_name
      @pending_number_of_files_to_visit += 1
      send_a_file_to_process
    end

    if message.kind_of? RequestAFile
      @file_processors << self.self.getSender.get
      send_a_file_to_process
    end

    if message.kind_of? FileToProcess
      @to_process_file_names << message.file_name
      @pending_number_of_files_to_visit += 1
      send_a_file_to_process
    end

    if message.kind_of? FileSize
      @total_size += message.size
      @pending_number_of_files_to_visit -= 1
      if @pending_number_of_files_to_visit == 0
        @fetch_size_future.get.complete_with_result(@total_size)
      end
    end
  end
end
```

The last step is to write the code that will exercise these two actors.

Download polyglotActors/jruby/FileSize.rb

```
startTime = System.nano_time()

size_collector = Actors.actor_of(SizeCollector).start()
size_collector.set_timeout(100000)

100.times do
  Actors.actor_of(
    lambda { FileProcessor.create_new(size_collector) }).start
end

total_size = size_collector.send_request_reply(FetchTotalSize.new(ARGV[0]))

Actors.registry().shutdownAll()

endTime = System.nano_time()

puts "Total size is #{total_size}"
puts "Time taken is #{(endTime - startTime)/1.0e9}"
```

UntypedActorprovides a few flavors of the actor_of() or actorOf() in Java lingo. We use the version of this method that expects a factory with a create() method to create an instance of the actor SizeCollector. The static create() method we provided in SizeCollector gets used here. Since we need to set a parameter at construction time into the instances of FileProcessor, we use the version of actor_of() that accepts a closure which internally creates an instance of the actor. JRuby allows us to fluently pass lambdas where closures are expected, so we took advantage of that here. Rest of the code is a simple translation from the Java version to JRuby.

Let's take this JRuby version of file size program for a ride across the /usr directory.

```
Total size is 3793911517
Time taken is 14.505133
```

9.5 Choices in Scala

You have two options for actor based concurrency in Scala—the scala.actor library that's provided with the Scala installation or the Akka library. Akka library provides better performance and is better suited for enterprise applications. If you'd like to use the Scala actor library, refer to Scala documentation, *Programming in Scala* [OSV08], or *Programming Scala* [Sub09]. If you decide to use Akka, refer to Akka documentation and the examples provided in Chapter 8, *Favoring Isolated Mutability*, on page 172.

9.6 Recap

You can program concurrency in any language of your choice on the JVM. If the language you're using is one we discussed in this chapter, you learned how to use actors in it. Where there were integration challenges, you saw techniques to work around them. In the next chapter we will wrap up with some recommendations for programming concurrency.

Part V

Epilogue

To understand a new idea, break an old habit.
► Jean Toomer

Chapter 10

Zen of Programming Concurrency

I hope you picked a few useful tidbits from this book for your journey across the sea of concurrency, a sail bound to be exciting and challenging at the same time. Doing it right can help safely reach your destination—deliver high performing, responsive applications that make good use of multicore processors.

Programming concurrency is not about the methods you call or the parameters you pass. It's more about the approaches you take, the efforts you put in, and the set of libraries you opt to use. You have to make the right choices to arrive at a clear concurrency solution.

The coding journey is not all about surviving the perils, it's about succeeding and having fun doing that. You'd have to decide on the number of threads you'd use and how you'd divide your application into concurrent tasks. The number of threads depends on the number of cores your application will have available at runtime and how much time your tasks will spend being blocked vs. in active computations. Dividing the application into tasks takes some effort. You have to understand the nature of the application and how to partition into tasks with uniform work load. Choosing the right number of threads and a uniform workload across the parts will help utilize the cores well throughout the application runtime as we discussed in Chapter 2, *Division of Labor*, on page 29.

10.1 Exercise Your Options

Few things affect concurrency as the approach you take to deal with state, from among shared mutability, isolated mutability, and pure immutability. Exercise your options and choose wisely.

Shared mutability is the fail-boat you want to avoid. The minute you refuse to get on it, a number of concurrency issues disappear. It takes some effort and discipline to avoid the all-too-familiar shared mutable style of programming, however, those efforts will soon pay back.

Get on board the boats of isolated mutable and pure immutability (where possible) to set on a smooth sail to your goals. These approaches remove the concerns at the root. Rather than spending your valuable time and effort preventing race conditions, these approaches completely eliminate them so you can focus on your application logic.

Choosing between these options may feel daunting, however, even in problems where there is significant state change, you can use isolated mutability and pure immutability as we saw in Chapter 3, *Design Approaches*, on page 50.

10.2 Concurrency: Programmer's Guide

The first step to programming concurrency is to upgrade to the modern JDK concurrency API and reduce your use of the old multithreading API that shipped with the first versions of Java. The java.util.concurrent API is such a breath of fresh air. You an easily manage a pool of threads, have better control over synchronization, and also benefit from high performing concurrent data structures as you saw in Chapter 4, *Scalability and Thread Safety*, on page 61. Several libraries now make use of this modern API and concurrency libraries (like Akka and GPars) provide a layer of abstraction on top of this modern API.

If your job responsibilities include "mutable state wrangler," you know the frustrations of ensuring thread-safety and achieving concurrent performance. You saw the tools and techniques to cope with these concerns in Chapter 4, *Scalability and Thread Safety*, on page 61 and Chapter 5, *Taming Shared Mutability*, on page 85. Make use of the ExecutorService to graciously manage your pool of threads and the concurrent data structures to coordinate between your threads/tasks. Utilize the Lock interface and related facilities for fine-grained synchronization, improved concurrency, and better control on lock timeouts.

Make fields and local variables final by default and relent only as an exception. Avoid shared mutability in your code. If after reasonable efforts you don't know a way to avoid mutability, then favor isolated mutability instead of shared mutability. It's much better to avoid concurrency concerns by design than struggle to handle using programatic means.

Take the time to learn, experiment, and prototype solutions using STM and actor based model.

The examples in Chapter 6, *Introduction to Software Transactional Memory*, on page 99 and Chapter 7, *STM in Clojure, Groovy, Java, JRuby, and Scala*, on page 151 will help you create your prototypes and gain experience with somewhat limited, but very powerful way to deal with shared mutability. If your application has frequent reads and infrequent write collisions, but has significant shared mutability, you may find solace in Software Transactional Memory (STM).

You can eliminate the problem at the root and isolate mutable states within actors. Actors are highly scalable as they make effective use of threads. As active objects, they internally sequence operations and therefore eliminate concurrency concerns. Spend time creating prototypes using the examples in Chapter 8, *Favoring Isolated Mutability*, on page 172 and Chapter 9, *Actors in Clojure, Groovy, Java, JRuby, and Scala*, on page 229.

If your application does not allow for such clean isolation, but have isolated tasks with interleaved coordination, explore options to mix the two models together as you've see in Section 8.9, *Mixing Actors and STM*, on page 212.

Some of these approaches may be new to you, so there may be a bit of a learning curve. Remember the first time you learned to ride a bike, a car,... it took time and effort, but once you got proficient, it helped you get around much faster than on foot. That's exactly how these unfamiliar techniques are, when you start they may take you out of your comfort zone and require some time and effort. Once you learn and master them, you can avoid the perils of concurrency and devote more of your time and effort where it's really deserved, on your application logic.

10.3 Concurrency: Architect's Guide

Concurrency and performance are most likely two issues that you have to contend with quite extensively. Fortunately there are some good options to influence the design and architecture of your application to deal with these effectively.

You'd agree it's much better to eliminate concurrency concerns than tackle them. The easiest route to that is to eliminate mutable state. Design your application around immutability or at least isolated mutability.

The two challenges you'll face as you move away from shared mutability are design and performance. We're so used to designing around shared mutability that it takes a good amount of practice and effort to do it differently. Look toward persistent and concurrent data structures for better performance.

Another area of influence you may have is the language choice. This generally is a touchy issue when you have developers who're quite attached to the languages they're comfortable with. If you're not convinced that a particular language is better, then it's hard for you to convince your team and your managers. The same applies to concurrency approaches as well. The one that's well known and widely used in your team may or may not be the right solution. If you're convinced that one approach is better than the other for your application, you have a chance to convince others.

So the first step is to convince yourself about these choices you have on hand. Take the time to prototype and create small yet significant parts of your application using the language or the concurrency model you feel is the most suitable for your application. Don't tackle the language choice and the concurrency choice at once. This may not help you or your team see a clear distinction. So, take them in turn and be objective in your comparisons.

Once you're convinced, you'll find it easier to lead your team. Your team may have to slow down as they pick up a new approach, but once they get proficient, they'll be able to benefit from the improved efficiency. The time they had spent in ceremonial activities, like managing shared mutable state or language constraints, can now be devoted to solve real problems. The examples using different concurrency models and different languages in this book may help you compare the effectiveness of one over the other. They may help you decide if the approach you want to consider is better than the one that your team may currently be using in practice.

10.4 Choose Wisely

Programming concurrency on the JVM has come a long way, you're no longer confined to one prescribed model. You have the freedom to choose from the models that make the most sense for your application.

You have at least three options:

- the beaten path of synchronize-and-suffer model
- the software transaction memory
- the actor based concurrency

You saw throughout this book that the concurrency model you choose is not forced upon you by the language you program in. It's decided by the nature of your application, the design approaches you take, and the willingness of your team to adapt to the model that's the right fit. Choose wisely and enjoy your journey.

Bon Voyage.

Appendix A

Web Resources

Akka . . . . http://akka.io
Jonas Bonér's Akka is a Scala based scalable concurrency library that can be used from multiple languages on the JVM. It provides both STM and Actor based concurrency.

Clojure Data Structures . . . . http://clojure.org/data_structures
This page discusses various data structures in Clojure, including the persistent data structures.

Values and Change—Clojure's approach to Identity and State. . .
. . . http://clojure.org/state
Rich Hickey discuses the Clojure model of separating the identify and state, and his approach to balance imperative and functional style of programming.

Designing and Building Parallel Programs. . .
. . . http://www.mcs.anl.gov/~itf/dbpp/book-info.html
Online version of the book by Ian Foster with the same title.

The Dining Philosophers Problem. . .
. . . http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF
The original paper by Edsger W. Dijkstra, "Hierarchical Ordering of Sequential Processes" Acta Informatica, 1971.

GPars . . . . http://gpars.codehaus.org
Started by Vaclav Pech as GParallelizer, GPars is a Groovy based concurrency library for use with Groovy and Java.

Ideal Hash Trees . . . . http://lamp.epfl.ch/papers/idealhashtrees.pdf
Phil Bagwell's paper that discusses persistent *Tries*.

The Java Memory Model. . .
. . . http://java.sun.com/docs/books/jls/third_edition/html/memory.html
The Java Language Specification discussing Java Threads and the Java Memory Model.

The JSR-133 Cookbook for Compiler Writers. . .
. . . http://g.oswego.edu/dl/jmm/cookbook.html
Doug Lea explains Memory Barriers in this article on the web.

The Lock Interface. . .
. . . http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Lock.html
The javadoc for the Lock interface introduced in Java 5.

The Meaning of Object-Oriented Programming. . .
. . . http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en
In an email exchange, Alan Kay discusses the meaning of OOP."

Multiverse: Software Transaction Memory for Java. . .
. . . http://multiverse.codehaus.org/overview.html
Multiverse, founded by Peter Veentijer, is a Java based implementation of STM that can be used from multiple languages on the JVM.

Polyglot Programming. . .
. . . http://memeagora.blogspot.com/2006/12/polyglot-programming.html
Neal Ford's vision of "Polyglot Programmers."

Test Driving Multithreaded Code. . .
. . . http://www.agiledeveloper.com/presentations/TestDrivingMultiThreadedCode.zip
Venkat's notes and code from his presentation entitled "Test Driving Multithreaded Code."

Appendix B

Bibliography

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2007.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[Blo08] Joshua Bloch. *Effective Java.* Addison Wesley Longman, Reading, MA, second edition, 2008.

[Goe06] Brian Goetz. *Java Concurrency in Practice.* Addison-Wesley, Reading, MA, 2006.

[Hal09] Stuart Halloway. *Programming Clojure.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

[Lea00] Doug Lea. *Concurrent Programming in Java, Second Edition: Design Principles and Patterns.* Addison-Wesley, Reading, MA, 2000.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima, Inc., Mountain View, CA, 2008.

[PJ95] Meilir Page-Jones. *What Every Programmer Should Know About Object-Oriented Design.* Dorset House, New York, 1995.

[Sub09] Venkat Subramaniam. *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

[VWWA96] Robert Virding, Claes Wikstrom, Mike Williams, and Joe Armstrong. *Concurrent Programming in Erlang*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1996.

Index