Extracted from:

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

This PDF file contains pages extracted from *Programming Concurrency on the JVM*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com .

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

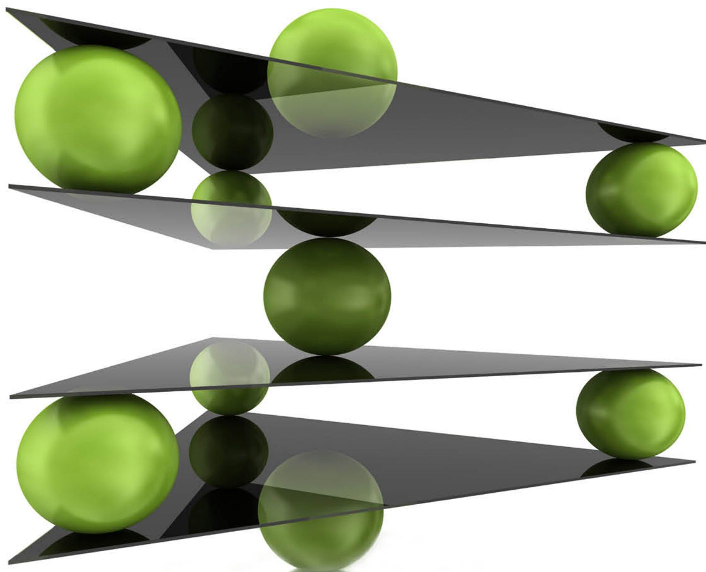
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors



Venkat Subramaniam

edited by Brian P. Hogan

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

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

Copyright © 2011 Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

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

“If it hurts, stop doing it” is a doctor’s 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, in languages like Java, we must still be very careful to avoid unmanaged mutable variables and side effects. 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, we treat tasks as lightweight processes, internal to the application/JVM. Instead of letting them grab the data, we pass immutable messages to them. Once these asynchronous tasks complete, they pass back or pass on their immutable results to other coordinating task(s). We design applications 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. The 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 fold of the JVM when Scala was introduced in 2003 (see *Programming in Scala* [OSV08] and *Programming Scala* [Sub09]).

In Java, we get to choose from more than half a dozen libraries[2] that provide actor-based concurrency: ActorFoundary, Actorom, Actors Guild, Akka, FunctionalJava, Kilim, Jetlang, and so on. Some of these libraries use aspect-oriented bytecode weaving. Each of them is at a different level of maturity and adoption.

In this chapter, we’ll learn how to program actor-based concurrency. For the 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. We can use it for both actor-based concurrency and for STM (see Chapter 6, *Introduction to Software Transactional Memory*, on page ?).

1. Someone asked me 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.
2. Imagine how boring it would be if we had just one good solution to pick.

• CLICK HERE to purchase this book now. discuss

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, we 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 we favor, it's clear we must avoid shared mutability.

Shared mutability—the root of concurrency problems—is where multiple threads can modify a variable. Isolated mutability—a nice compromise that removes most concurrency concerns—is where only one thread (or actor) can access a mutable variable, ever.

In OOP, we 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 the actor-based programming model, we allow only one actor to manipulate the state of an object. While the 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.

We take a different design approach when programming with actors compared to programming merely with objects. We divide the problem into asynchronous computational tasks and assign them to different actors. Each actor's focus is on performing its designated task. We confine any mutable state to within at most one actor, period (see Figure 11, *Actors isolate mutable state and communicate by passing immutable messages.*, on page 7). We also ensure that the messages we pass between actors are totally immutable.

In this design approach, we let each actor work on 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. We can think of this as taking OOP to the next level where select objects—mutable and active—run in their own threads. The only way we're allowed to manipulate these objects is by sending messages to them and not by directly calling methods.

3. Java had other partners in this crime, so it doesn't deserve all the blame.

• CLICK HERE to purchase this book now. discuss

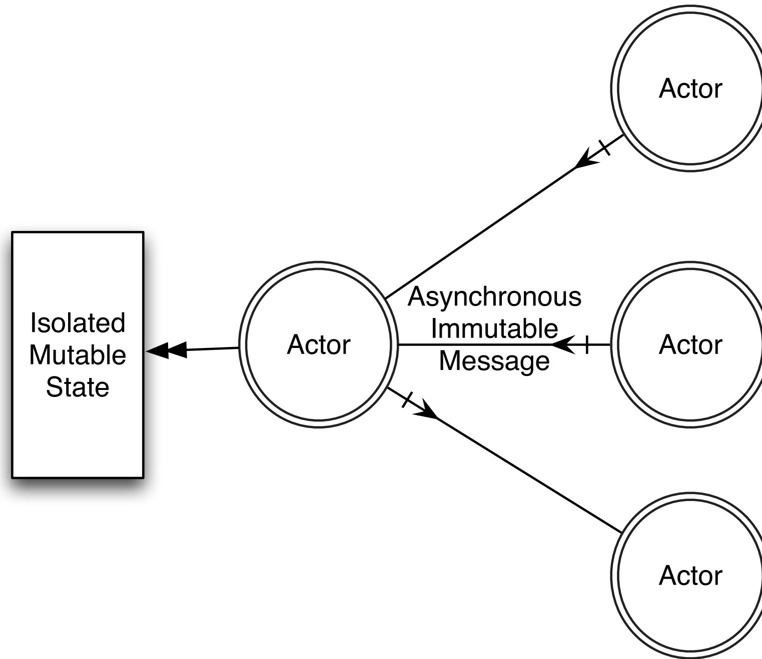


Figure 11—Actors isolate mutable state and communicate by passing immutable messages.

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 a cell phone. Both Sally and Sean may leave a message at the same time on Bob's cell phone. The cell phone provider saves both their messages for Bob to retrieve at his convenience. Similarly, the actor library allows multiple actors to send messages concurrently. The senders are nonblocking by default; they send off a message and proceed to take care of their business. The library lets the designated actor sequentially pick its 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 12, *Life cycle of an actor*, on page 9. 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 the application they're part of.

If actors play a major role in our design, we'd expect many of them to float around during the execution of the 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. Bob doesn't have a designated seat at his company cafeteria (he needs to find another job if he does), and each time he goes for a meal, he gets seated in one of the available seats. 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. Although multiple actors may be active at any time, only one thread is active in an actor at any instance. This provides concurrency among actors while eliminating contention within each actor.

8.3 Creating Actors

We have quite a few choices of actor libraries to pick from, as I mentioned earlier. In this book, we use Akka, a Scala-based library[4] with pretty good performance and scalability and with support for both actors and STM. We 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 it's 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 we can easily create and use actors in 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 method is called

4. In addition to Akka, there are at least two more Scala-based libraries—Scala actors library and the Lift actors.

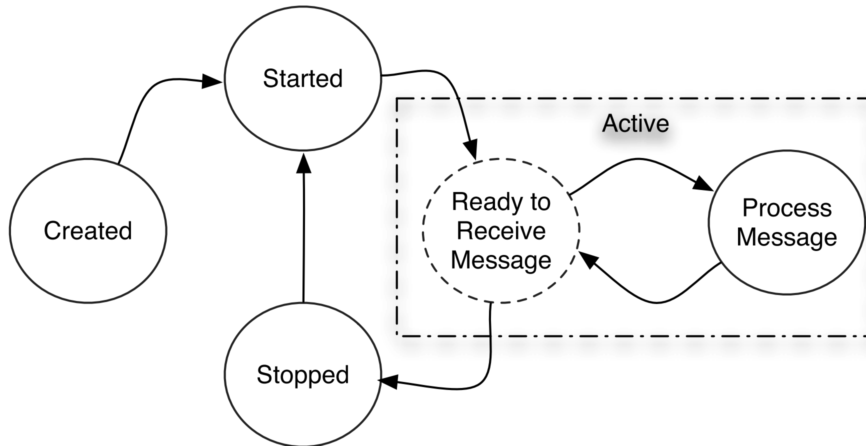


Figure 12—Life cycle of an actor

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 a 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 messages with their role, so let's get to that:

Download favoringIsolatedMutability/java/create/UseHollywoodActor.java

```
public class UseHollywoodActor {
  public static void main(final String[] args) throws InterruptedException {
    final 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, of type ActorRef, is a reference to our actor instance.

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, which is how actors switch threads, as we'll see soon. In the end, we ask to close down all running actors. Instead of calling the shutdownAll() method, we may call the stop() method on individual actors or send them a kill message as well.

All right, to run the example, let's compile the code using javac and remember to specify the classpath to Akka library files. We can simply run the program as we would run regular Java programs. Again, we must 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 follows:

```
export AKKA_JARS="$AKKA_HOME/lib/scala-library.jar:\
$AKKA_HOME/lib/akka/akka-stm-1.1.3.jar:\
$AKKA_HOME/lib/akka/akka-actor-1.1.3.jar:\
$AKKA_HOME/lib/akka/multiverse-alpha-0.6.2.jar:\
$AKKA_HOME/lib/akka/akka-typed-actor-1.1.3.jar:\
$AKKA_HOME/lib/akka/aspectwerkz-2.2.3.jar:\
$AKKA_HOME/config:\
."
```

We need to define the AKKA_JARS environment variable appropriately for our operating system to match the location where we have Scala and Akka installed. We may use the scala-library.jar file that comes with Akka, or we may use it from the local Scala installation.

By default Akka prints extra log messages on the standard output; we saw how to configure that in *Creating Transactions in Java, on page ?*.

Let's compile and run the code to watch our actor responding to 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 responds 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 construction time. If we desire, we can send parameters during actor creation. For example, to initialize the actor with the Hollywood actor's name:

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. We can take other actions such as returning an error code, logging, calling the user's mom to report, and so on. 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 {

    final 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 by 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, because that would take 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 the runtime exception akka.actor.ActorInitializationException with the 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 the 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 gives time for the actor to respond to messages before we shut it down. Let's take it for a ride to see the output:

```
Hanks playing James Lovell
Hanks plays no Politics
Hanks playing Forrest Gump
```

• CLICK HERE to purchase this book now. discuss