Scala Actors

-Terrance Dsilva

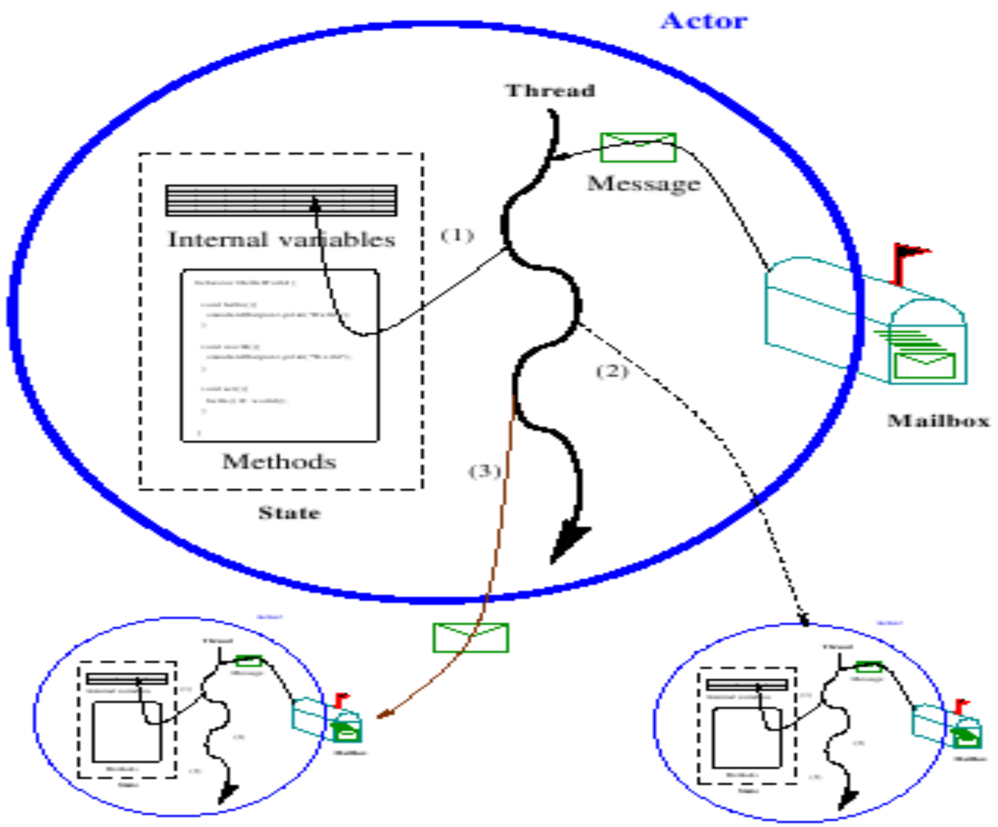
Concurrency

- Most programmers avoid by retreating to multiple independent processes that share data externally (for example, through a database or message queue)
- How do you know what your multithreaded program is doing, and when? What value does a variable hold when you have two threads running, or five, or fifty?
- How can you guarantee that your program's many tendrils aren't clobbering one another in a race to take action

Actors

- Thankfully, Scala offers a reasonable, flexible approach to concurrency
- Actors aren’t a concept unique to Scala. Actors, originally intended for use in Artificial Intelligence research, were first put forth in 1973 (see [Hewitt1973] and [Agha1987]).
- Actors have appeared in a number of programming languages, most notably in Erlang and Io.

What is an actor?



More on Actors

- Actors encapsulate state and behavior (like objects)
- Actors are logically active (unlike most objects)
- Actors communicate through asynchronous message passing (*nonblocking* send, *locking* receive)

Actors in Abstract

- Actor is an object that receives messages and takes action on those messages.
- The order in which messages arrive is unimportant to an Actor, though some Actor implementations (such as Scala’s) queue messages in order.
- An Actor might handle a message internally, or it might send a message to another Actor, or it might create another Actor to take action based on the message.

Actors in Abstract- cont

- Actors don’t enforce a sequence or ordering to their actions. This inherent eschewing of sequentiality, coupled with independence from shared global state, allow Actors to do their work in parallel.
- Actors are a very high-level abstraction.
- Enough theory. Let’s see Actors in action.

Actors in Scala

- Actors in Scala are objects that inherit from scala.actors.Actor.

```
import scala.actors.Actor
   class  Sample extends Actor {
 def test() {
 println(“Hello.")
}
 }
val objectsample = new Sample
objectsample.start
```

- ➢ Actor defined in this way must be both instantiated and started, similar to how threads are handled in Java. It must also implement the abstract method act, which returns Unit.

Factory Made Actor

- The scala.actors package contains a factory method for creating Actors that avoids much of the setup in the above example. We can import this method and other convenience methods from scala.actors.Actors._.

```
import scala.actors.Actor
import scala.actors.Actor._
val paulNewman = actor {
  println("To be an actor, you have to be a child.")
}
```

➢ While a subclass that extends the Actor class must define act in order to be concrete, a factory-produced Actor has no such limitation. In this shorter example, the body of the method passed to actor is effectively promoted to the act method from our first example

Sending Messages to Actors

Actors can receive any sort of object as a message, from strings of text to numeric types to whatever classes .
An Actor should only act on messages of familiar types; a pattern match on the class and/or contents of a message is good defensive programming, and increases the readability of Actor code.

```
val Test1 = actor {
loop {
   receive {
   case s: String => println("I got a String: " + s)
   case i: Int => println("I got an Int: " + i.toString)
        }
   }
}

Test1 ! "hi there"
Test2 ! 23
```

Sending Messages to Actors-cont

- The example prints
  - I got a String: hi there
  - I got an Int: 23

- The body of Test1 is a receive method wrapped in a loop. loop is essentially a nice shortcut for while(true); it does whatever is inside its block repeatedly. receive blocks until it gets a message of a type that will satisfy one of its internal pattern matching cases.
- The final lines of this example demonstrate use of the ! (exclamation point, or *bang*) method to send messages to our Actor. If you've ever seen Actors in Erlang, you'll find this syntax familiar.
- The Actor is always on the left-hand side of the bang, and the message being sent to said Actor is always on the right

The Mailbox

- Every Actor has a mailbox in which messages sent to that Actor are queued.

```
import scala.actors.Actor
import scala.actors.Actor._
val count = actor {
    loop {
react {
   case "how many?" => { println(mailboxSize.toString + " number
   is mailbox.")
   }
   }
   }
 }
count ! 1
count ! 2
count! 3
```

The Mailbox

- This example produces the following output.

```
3 messages in my mailbox.
```

- **Tip**

  If you see an Actor’s mailbox size ballooning unexpectedly, you’re probably sending messages of a type that the Actor doesn’t know about. Include a catchall case (_) when pattern matching received messages to find out what’s harassing your Actors.

Effective Actors

In order to get the most out of Actors, there are few things to remember. First off, note that there are several methods you can use to get different types of behavior out of your Actors. The table below should help clarify when to use each method.

Method	Returns	Description
act	Unit	Abstract, top-level method for an Actor. Typically contains one of the following methods inside it.
receive	Result of processing message	Blocks until a message of matched type is received.
receiveWithin	Result of processing message	Like receive but unblocks after specified number of milliseconds.
react	Nothing	Requires less overhead (threads) than receive.
reactWithin	Nothing	Like react but unblocks after specified number of milliseconds.

Sending and receiving messages

- To send a message, use actor ! message

  The thread sending the message keeps going--it doesn't wait for a response

- To receive a message (in an Actor), use either receive {...} or react {...}

  -Both receive and react block until they get a message that they   recognize (with case)

  -When receive finishes, it keeps its Thread

  -Statements following receive{...} +will then be executed

  -When react finishes, it returns its Thread to the thread pool

  -Statements following the react{...} statement will not be executed

  -The Actor's variable stack will not be retained

  -This (usually) makes react more efficient than receive

- Hence: Prefer react to receive, but be aware of its limitations

Doing it correctly

- Martin Odersky gives some rules for using Actors effectively
  - Actors should not block while processing a message
  - Communicate with actors only via messages
    - Scala does not prevent actors from sharing state, so it's (unfortunately) very easy to do
  - Prefer immutable messages
    - Mutable data in a message is shared state
  - Make messages self-contained
    - When you get a response from an actor, it may not be obvious what is being responded to
    - If the request is immutable, it's very inexpensive to include the request as part of the response
    - The use of case classes often makes messages more readable