Reference


Reading

Chapters 30
import scala.actors.Actor

class Example(name: String) extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println(name + " " + k)
    }
  }
}

val a = new Example("a")
a.start

Output

  a 1
  a 2
  a 3
  a 4
  a 5
  a 6
  a 7
  a 8
  a 9
  a 10

act is sort of like run in thread. One does not call it directly. One call start which registers the actor with the scheduler and act is called from a different thread.
Showing Concurrency

scala> new Example("a").start; new Example("b").start
a 1
a 2
a 3
a 4
a 5
a 6
a 7
a 8

scala> b 1
b 2
a 9
b 3
a 10
b 4
b 5
etc

Yes the two actors are running in different threads.
Multiple Starts allowed

```scala
import scala.actors.Actor

class Example extends Actor {
  def act = println("run")
}

val test = new Example

Output
run
run
```

Which is unlike the run method in a thread
Other Concurrent Examples

```scala
import scala.actors.Actor

class Example(name: String) extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println(name + " " + k)
    }
  }
}

object Main extends Application {
  val a = new Example("a")
  val b = new Example("b")
  a.start
  b.start
}
```

Run as Application
scalac example.scala
scala Main
Output is interleaved

Run in interpreter
scala
scala> :load example.scala
scala> Main
Output is interleaved first time
Output is not interleave on second load & run
import scala.actors.Actor

object SampleActor extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println("A " + k)
    }
  }
}

SampleActor.start
Utility actor Method

```scala
import scala.actors.Actor

val x = Actor.actor {
  for (k <- 1 to 10) {
    println("Hello " + k)
  }
}
```

The contents of the actor method are the act method of the new actor. The new actor is automatically started.
Utility actor Method

```scala
import scala.actors.Actor._

val x = actor {
    for (k <- 1 to 10) {
        println("Hello " + k)
    }
}
```

Shorter version

```scala
val x = actor {
    for (k <- 1 to 10) {
        println("Hello " + k)
    }
}
```

Commonly used
Messages

Asynchronous
  One-way

Synchronous
  Futures

Filtering

Mailbox
Each actor has a mailbox. The outside world can send the actor messages, which are placed in the mailbox. The actor then can remove and read messages in the mailbox.
import scala.actors.Actor

class Basic extends Actor {
  def act = {
    receive {
      case mail =>
        println("I got mail " + mail)
    } //receive reads one message from the actors mailbox. act only runs once so we only read one message from the mailbox. The second message remains in the mailbox.
  }
}

val a = new Basic
a.start
a ! "hi" //send a message
a ! 12 //another message

Output
I got mail hi
import scala.actors.Actor

class Basic extends Actor {
    def act = {
        while (true) {
            receive {
                case mail =>
                    println("I got mail " + mail)
            }
        }
    }
}

val a = new Basic
a.start
a ! "hi"
a ! 12
a ! List(1,2,3)

Output
I got mail hi
I got mail 12
I got mail List(1, 2, 3)

We can send anything in the message.
Infinite Loop Shortcut

```scala
import scala.actors.Actor
import scala.actors.Actor._

class Basic extends Actor {
  def act = {
    Actor.loop {
      loop {
        receive {
          case mail =>
            println("I got mail " + mail)
        }
      }
    }
  }
}
```
import scala.actors.Actor

class Basic extends Actor {
    def act = {
        receive {
            case mail => {
                println("I got mail " + mail)
                act
            }
        }
    }
}
exit

import scala.actors.Actor
import scala.actors.Actor._

class Basic extends Actor {
    def act = {
        loop {
            receive {
                case mail =>
                    println("I got mail "+ mail)
            }
        }
    }
}

val a = new Basic
a.start
a ! "hi"
a.exit
a ! "are you there?"

I got mail hi
scala.actors.ExitActorException
I got mail are you there?

exit does "kill" the actor, but it has to be called in the thread running the actor. So the above code does not really work. The actor continues to run.
How to use exit

```scala
import scala.actors.Actor
import scala.actors.Actor._

class Basic extends Actor {
  def act = {
    loop {
      receive {
        case "die" => exit
        case mail =>
          println("I got mail " + mail)
      }
    }
  }
}

val a = new Basic
a.start
a ! "hi"
a ! "die"
a ! "are you there?"

Output
I got mail hi
```
val partialFunction: PartialFunction[Any,Unit] =
  {case mail => println("I got mail " + mail)}
receive (partialFunction)

receive is a method. I for one am glad for the syntactic sugar of the top version
import scala.actors.Actor
import scala.actors.Actor._

class MailboxExample extends Actor {
  def act = {
    loop {
      receive {
        case "size" => println(mailboxSize)
        case "quit" => {
          println("goodby")
          exit
        }
      }
    }
  }
}

val test = new MailboxExample
test.start

// Initial message

// Second message

// Sending command "size" 2 times

// Sending command "quit"

// Output: 2
// Output: 3
// Output: goodby
Asynchronous - One Way

The message is sent. No reply is sent to the sender and the sender does not wait for a reply.
Asynchronous - One Way - !

import scala.actors.Actor
import scala.actors.Actor._

class Basic extends Actor {
  def act = {
    loop {
      receive {
        case "die" => exit
        case mail =>
          println("I got mail "+ mail)
          println("I got mail "+ mail)
      }
    }
  }
}

val a = new Basic
a.start
a ! "hi"
a ! "die"
a ! "are you there?"

Output
I got mail hi

! sends an asynchronous message.
Asynchronous - With Separate Return

Time

Receiver

Message

Sender

Receiver

Message

Sender
Asynchronous - Return to sender

import scala.actors.Actor._

class Adder extends Actor {
  def act = {
    loop {
      receive {
        case x: Int => sender ! x + 1
      }
    }
  }
}

class Requester(adder: Actor) extends Actor {
  def act = {
    adder ! 3
    receive {
      case x: Int => println("Answer " + x)
    }
  }
}

val a = new Adder
a.start
val sender = new Requester(a)
sender.start

The sender method returns a reference to the actor/thread that send the current message
Asynchronous - With return address

import scala.actors.Actor._

class Adder extends Actor {
  def act = {
    loop {
      receive {
        case (x: Int, receiver: Actor) =>
          receiver ! x + 1
      }
    }
  }
}

class Receiver extends Actor {
  def act = {
    loop {
      receive {
        case x: Int =>
          println("Answer "+ x)
      }
    }
  }
}
Using the Example

val a = new Adder
a.start
val sender = new Receiver
sender.start
a ! (12, sender)
a ! 12
a ! (3, sender)
a ! "cat"

Output
Answer 13
Answer 4
Sender blocks until receiver replies
import scala.actors.Actor._
import scala.actors.Actor

class Adder extends Actor {
  def act = {
    var answer:Int = 0
    loop {
      receive {
        case x:Int => reply( x + 1)
      }
    }
  }
}

val a = new Adder
a.start
val answer: Any = a !? 3
a.exit

!? blocks until it receives an answer
Sender block when it requests a value from the future until the value is actually available.
import scala.actors.Actor._
import scala.actors.Actor

class Adder extends Actor {
  def act = {
    var answer:Int = 0
    loop {
      receive {
        case x:Int => {
          Thread.sleep(1000)
          reply( x + 1)
        }
      }
    }
  }
}

import scala.actors.Future
val a = new Adder
a.start
val answer: Future[Any] = a !! 3
val start = System.currentTimeMillis()
val value: Any = answer()
val end = System.currentTimeMillis()
a.exit
println(end - start)

Output
1005

!! returns immediately. However it returns a future object. When you try to access the value in the future the code blocks until the value is available
Future isSet

val a = new Adder
a.start

val answer: Future[Any] = a !! 3
var value: Any = 0
if (answer.isSet)
    value = answer()
else
    println("not ready")
Synchronous with timeout

val a = new Adder
a.start
val millisecondsToWait = 1500
val answer: Option[Any] = a !?(millisecondsToWait,3)
if (!answer.isEmpty)
  println(answer.get)
React & Receive

react
Reads a message from the mail box
Does not return
Allows scheduler to use one thread to handle multiple actors

receive
Reads a message from the mail box
One thread per actor
class Receiver extends Actor {
    def act = {
        println("Before receive")
        receive {
            case _ => println("receive test")
        }
        println("After receive")
    }
}

val a = new Receiver
a.start
a ! 1

Output
Before receive
receive test
After receive
class Reactor extends Actor {
    def act = {
        println("Before react")
        react {
            case _ => println("React test")
        }
        println("After react")
    }
}

val a = new Reactor
a.start
a ! 1

Output
Before react
React test
import scala.actors.Actor

class MutableExample extends Actor {
  def act = {
    receive {
      case x:Array[Int] => x(0) = 30
    }
  }
}

var data = Array(2,1)
val actor = new MutableExample
actor.start
actor ! data
println(data(0))

Output
10

The data in messages is shared between
Don't use mutable data in messages
Sieve Example - Collector

```scala
import scala.actors.Actor
import scala.actors.Actor._

class Collector extends Actor {
  def act = {
    loop {
      receive {
        case x: Int => println(x)
        case "quit" => exit
      }
    }
  }
}
```

Sieve Example - Filter

class Filter(primes:List[Int], endActor:Collector) extends Actor {
    val prime: Int = primes.head
    val next: Actor = if (primes.length > 1)
        new Filter(primes.tail, endActor)
    else endActor

    next.start
    def act = {
        loop {
            receive {
                receive {
                    case x: Int => if (x % prime != 0) next ! x
                    case "quit" => {
                        println("goodby")
                        next ! "quit"
                        exit
                    }
                }
            }
        }
    }
}
Sieve Example - Using

val smallPrimes = List(2,3,5,7,11,13,17,23)
val seive = new Filter(smallPrimes, new Collector)
seive.start
for (x <- 2 to 100)
    seive ! x
seive ! "quit"
Remote Actors

Local Actors
- Run in same JVM
- May be run in separate thread

Remote Actors
- Run in different JVM
- May be run on machines

Messages sent to Remote Actors
- Must be serializable
Remote Actor Server

```scala
import scala.actors.Actor
import scala.actors.Actor._
import scala.actors.remote.RemoteActor

class RemoteAdder(port: Int) extends Actor {
  def act() {
    RemoteActor.alive(port)
    RemoteActor.register('Adder, self)
    println("go")
    loop {
      receive {
        case n:Int => reply(n + 1)
      }
    }
  }
}
```

val port = 8888
val server = new RemoteAdder(port)
server.start()

Based on http://youshottheinvisibleswordsman.co.uk/2009/04/01/remoteactor-in-scala/
import scala.actors.remote.RemoteActor
import scala.actors.remote.Node

val remoteport = 8888
val peer = Node("10.0.1.192", remoteport)
val server = RemoteActor.select(peer, 'Adder)
val answer = server !? 10
println(answer)