Chapter 31
Akka Actors

31.1 Introduction

This chapter introduces the concept of an Actor as an approach to the development of concurrent programs. In particular we focus on the Scala *Akka* implementation of the Actor model based on release 2.5.x (the current release at the time of writing). Note that older versions of Scala also had their own *scala.actor* implementation of the Actor model—however this is now deprecated and the recommendation is to use the Akka implementation.

31.2 The Actor Model

The Actor model of concurrency (which dates from 1973) is based on the idea of having independent concurrent *Actors* that receive and send asynchronous messages and that perform some behaviour based on these message requests. These actors can hold their own state and behaviour. However, ideally only immutable data is exchanged between them. Therefore each actor is independent of all other actors and only performs some computation or processing based on a message sent to it. This is illustrated in Fig. 31.1.

The key idea underpinning the Actor model is that most of the problems with concurrency, from deadlocks to data corruption, result from having shared state. Thus in the Actor world there is no shared state (such as a concurrent producer–consumer queue). Instead, messages are sent between actors and these messages are queued in an *inbox* in a similar manner to email messages. The actors then respond to the messages as they get to them.

The Actor model has been used as the basis of a number of implementations including those in Erlang, the scala.actor framework and the Scala and Java Akka libraries. It is also used with the education system scratch.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_31

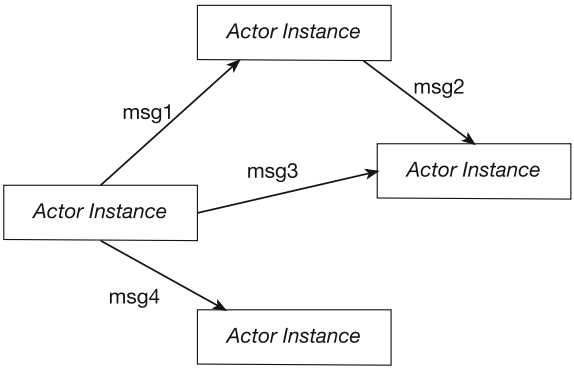

Fig. 31.1 Actors communicating via asynchronous messages

The actor model consists of a few key principles:

- No shared state.
- Lightweight processes.
- Asynchronous message passing.
- Buffering for incoming messages via an a *inbox* in which it receives messages.
- Message processing with pattern matching with each message being processed one at a time.
- No shared mutable data.
- No blocking operations.
- Any process can send a message to an actor.
- An actor does not do anything unless/until it receives a message.

Thus an actor is an independent flow of control running in its own thread or process. In many ways you can think of an actor as a *thread of execution* with extra features.

The fact that Actors do not share state means that you implement an actor as if it was a simple sequential process. This avoids a large number of the problems that result from shared state. It should be noted that in most Actor implementations it is actually possible to share state; it is just a very bad idea!

Within Scala there are actually two distinct implementations of the Actor model, these are the *scala.actor* framework and the *akka.actor* framework. In the rest of this chapter we will look at the latter of these two.

31.3 Some Terminology

The world of concurrent programming is full of terminology that you may not be familiar with. Some of those terms and concepts are outlined below:

- *Asynchronous versus synchronous invocations*. Most of the method, function or procedure invocations you will have seen in programming represent

synchronous invocations. A synchronous method or function call is one which blocks the calling code from executing until it returns. Such calls are typically within a single thread of execution. Asynchronous calls are ones where the flow of control immediately returns to the *callee* and the *caller* is able to execute in its own thread of execution, allowing both the caller and the call to continue processing.

- *Non-blocking versus blocking code*. Blocking code is a term used to describe the code running in one thread of execution, waiting for some activity to complete which causes one of more separate threads of execution to also be delayed. For example, if one thread is the producer of some data and other threads are the consumers of that data, then the consumer treads cannot continue until the producer generates the data for them to consume. In contrast, non-blocking means that no thread is able to indefinitely delay others.
- *Concurrent versus parallel code*. Concurrent code and parallel code are similar, but different in one significant aspect. Concurrency indicates that two or more activities are both making progress even though they might not be executing at the same point in time. This is typically achieved by continuously swapping competing processes between execution and non-execution. This process is repeated until at least one of the threads of execution (Threads) has completed their task. This may occur because two threads are sharing the same physical processor with each is being given a short time period in which to progress before the other gets a short time period to progress. The two threads are said to be sharing the processing time using a technique known as time slicing. Parallelism on the other hand implies that there are multiple processors available allowing each thread to execute on their own processor simultaneously.

31.4 Scala Threads

Underpining the Akka Actor model is the notation of a Thread. A Scala Thread is a pre-emptive lightweight process. Every thread (process) has an associated priority, and a Scala thread runs to completion unless a higher priority process attempts to gain control. Scala does not necessarily share the processor time amongst processes of the same priority (as this is the responsibility of the underlying JVM). Threads with a higher priority are executed before threads with a lower priority. A thread with a higher priority may interrupt a thread with a lower priority. By default, a process inherits the same priority as the process that spawned it.

A thread is a "lightweight" process because it does not possess its own address space and it may not be treated as a separate entity by the host operating system. Instead, it typically exists within a single virtual machine process using the same address space.

It is useful to get a clear idea of the difference between a thread (running within a single virtual machine process) and a multi-process system. The thread that is

currently executing is termed the active thread. A thread can also be suspended (i.e. waiting to use the processor) or stopped (waiting for some resource).

You will find when reading about Akka Actors this term thread is used when considering how an Actor manages the requests it receives. For example, an Actor utilizes a single thread thus allowing it to process a single request at a time, to completion, before processing the next request. Alternatively, multiple Actors may use a pool of threads, so that they can process multiple requests via the multiple underling threads managed in the pool.

31.5 Akka Scala Actor Library

The Akka Actor library is an implementation of the Actor model. Note that there is both a version of this library for Java and for Scala. You therefore need to be careful if you are researching this library on the Web that you are looking at material for the Scala version.

At the time of writing the current version of this library is the Akka 2.2.3 release. Note that the 2.2 family of distributions are significantly different to those that predate them and you therefore also need to be careful that you are not looking at documentation related, for example, to the 2.1 release of the Akka library.

The Akka library is open source and is available under the Apache 2 licence. You can obtain it from the Akka home page (http://akka.io). If you wish to download the Akka library then you can do so from the Download page, which is available from the site shown in Fig. 31.2.

Note that there are two downloads available: one is the LightBend (aka Typesafe) akka-quickstart-scala.zip distribution which includes the Akka library as well as other resources and templates.

Many projects will access the Akka libraries using a dependency management system such as Maven or Ivy. The examples in this chapter were written with Akka 2.5.4, and so this dependency can be added to your project:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.12</artifactId>
  <version>2.5.4</version>
</dependency>
```

The Akka library is actually a very modular library. This is because it is comprised of different library jar files representing different features of the Akka system. If you are not using those features then you do not need to incorporate those library files. Some of the modules you might decide to use are presented below:

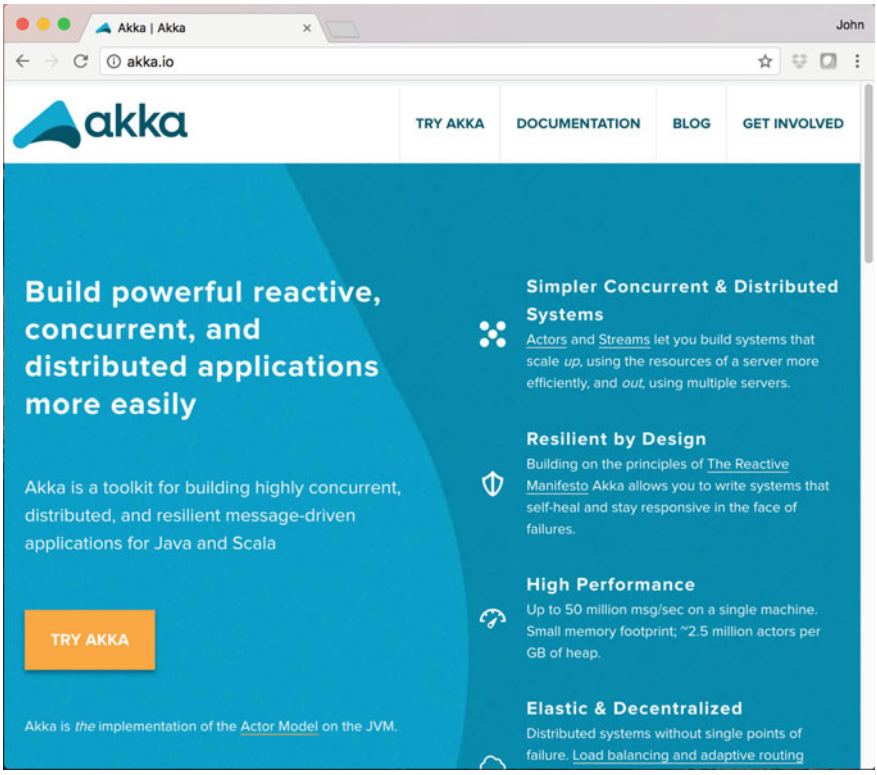

Fig. 31.2 Obtaining the Scala Akka library

- akka-actor—Classic Actors, Typed Actors, IO Actor, etc.
- config—used to handle configuration type tasks within the framework.
- akka-cluster—cluster membership management, elastic routers.
- akka-mailboxes-common—common infrastructure for implementing durable mailboxes.

The two parts of the library that we will be working with are the akka-actor and the config jars. It is these two jars that you need to find and add to your projects. Note that the names of the files are derived from the release of Scala that is targeted and the release number of Akka; for example in Fig. 31.3 the akka-actor file is actually called akka-actor_2.12-2.5.4.jar, that is it applies to Scala 2.12 and is release 2.5.4 of Akka.

The library itself can also be downloaded from the Maven repository if required. If you do this directly then you will need to add the jars to your build path of your project within the Scala IDE as shown in Fig. 31.4.

Fig. 31.3 Akka library jar files

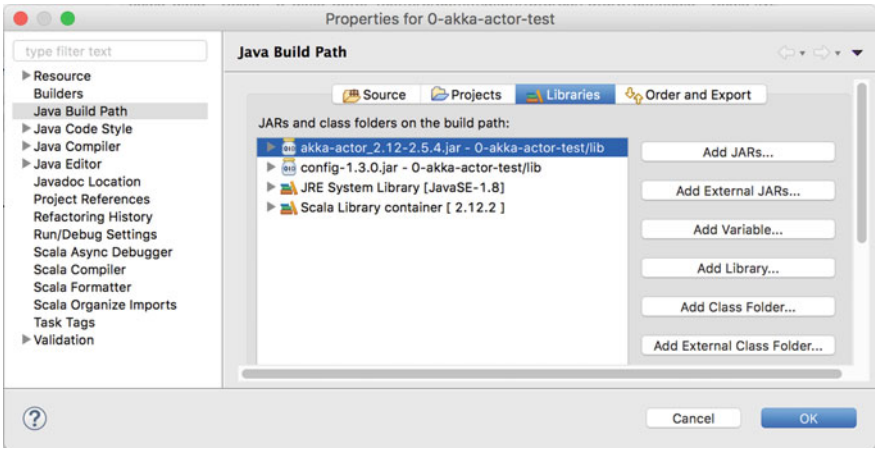

Fig. 31.4 Setting the classpath properties for an Akka project in the Scala IDE

31.6 Concurrent Hello World

In this section we will look at two simple Akka-based programs. The first of these programs creates a simple Actor that can be sent messages. If the message sent is the string "Hello" then it will print out the message "Hello World"; if anything else (of any type) is sent to the actor then it will print out the string "Hello Whoever".

To implement an Actor you must mix in the `Actor` trait. This trait requires that a single method `receive` is implemented. This method will be called when a message is available to be processed in the actor's inbox. Locally the inbox is queue in which messages are added to the back of the queue and taken from the front of the queue. Each message is processed to completion before the next message is taken. This is shown in Fig. 31.5.

The import statements for the example being discussed here are:

```
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorSystem
```

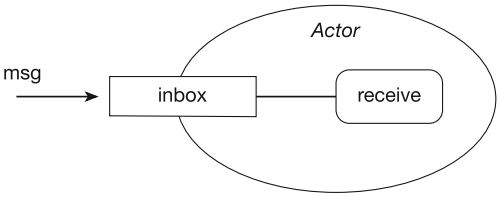

Fig. 31.5 Actor with an inbox and a receive method

It is important to take note of the package names as there is also an Actor type in the `scala.actor` package. If you import that type you will get compilation problems as it is not compatible with the Akka library.

The following listing illustrates how a simple actor called `Greeter` can be implemented:

```
class Greeter extends Actor {
  def receive = {
    case "hello" => println("Hello World")
    case _ => println("Hello Whoever")
  }
}
```

In this case, when the string "hello": is received it will print out "Hello World", while anything else (as indicated by the wild card '_') will print out "Hello Whoever". Notice that the `receive` method is implemented using *case pattern matching*, and thus, the Actor can handle different messages, different message types, etc., in different ways. If you want to be able to handle unknown messages then you need to have a default case as in the example above.

The test application that uses this `Greeter` actor is shown below:

```
object HelloworldAkkaTest extends App {
  val props = Props[Greeter]
  val system = ActorSystem("mysystem")
  val actor1 = system.actorOf(props)
  actor1 ! "hello"
  actor1 ! "Goodbye"
  actor1 ! "hello"
  actor1 ! 42
  actor1 ! true
  system.terminate
}
```

This test application goes through three steps in order to obtain a reference to the actor held in the val `actor1`. The steps are:

1. **Create a Props instance**. `Props` is a configuration class used to specify the options to be used to create an actor. In this case it is specifying the class that defines the Actor (i.e. the `Greeter` class). This Props type can also be used to pass data to any constructor defined on the class (we will see this later).

2. **Access the ActorSystem**. The Actor system is the element within the Akka framework that is responsible for creating top-level actors. These actors can create child actors, can respond to messages and can be stopped, restarted, etc. The `ActorSystem` is given a name so that multiple such systems can be created if required. However, note that the `ActorSystem` is a heavyweight object and you should try to only create one per application unless absolutely necessary.
3. **Create a new actor using actorOf**. Using the actor system reference obtained in the previous step, the application creates a new actor based on the properties in props. The `actorOf` method is a factory method that creates new actors of the type specified by the props object. The type returned by the `actorOf` method is an `ActorRef`.

Once it has successfully created a new actor a series of message are sent to the actor. These mix strings, within an `Int` and a `Boolean`. They are handled by the `receive` method in turn. The method decides what to do with each message based on the pattern-matching case statements defined within it.

It should be noted that actor1 does not hold a reference to the actual actor instance. Instead it holds a reference to the *Actor Incarnation*. This is a wrapper around the current *Actor Instance*. For the most part this is transparent to the programmer. This structure is used to allow an actor that has exited its own underlying thread (or process) to resume processing. This is done within the Akka library by creating a new Actor instance and *restarting* it. However, as this is hidden inside the Actor Incarnation the programmer is not affected by this (Fig. 31.6).

The output generated by running the HelloworldAkkaTest is shown in Fig. 31.7.

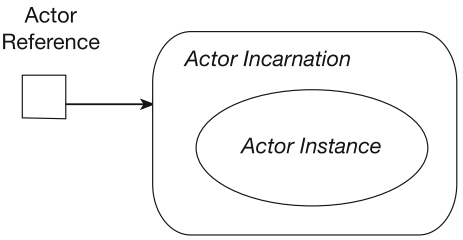

Fig. 31.6 Role of the actor reference

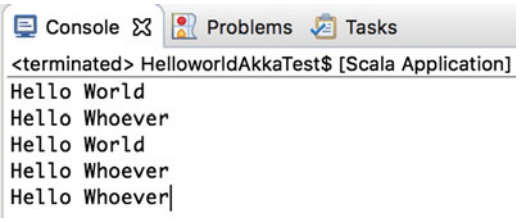

Fig. 31.7 Output form the HelloworldAkkaTest application

31.7 Concurrent Actors

The example discussed in the previous section illustrates how an Akka Actor can be defined and how messages can be sent to an actor. However, it does not really capture the concurrency inherent in Actors. To illustrates this idea see the following listing presenting the `Printer` Actor. This Actor prints out different strings depending upon the messages it receives. However, it does each string 500 times. This means that if we have multiple instances of the `Printer` running concurrently, we should be able to observe a mixture of A, B, C and _ being printed out.

```
class Printer extends Actor {
  def receive = {
    case "A" => for (i <- 1 to 500) print("A")
    case "B" => for (i <- 1 to 500) print("B")
    case "C" => for (i <- 1 to 500) print("C")
    case _ => for (i <- 1 to 500) print("_")
  }
}
```

The sample program that uses this `Printer` Actor is shown below:

```
object AkkaPrinterTest extends App {
  val props = Props[Printer]
  val system = ActorSystem("mysystem")
  val actor1 = system.actorOf(props, "actor1")
  val actor2 = system.actorOf(props, "actor2")
  val actor3 = system.actorOf(props, "actor3")
  val actor4 = system.actorOf(props, "actor4")
  println("Sending messages to Actors")
  actor1 ! "A"
  actor2 ! "B"
  actor3 ! "C"
  actor4 ! "X"
  println("Done setup - now sleeping")
  Thread.sleep(1000)
  println("Terminating the ActorSystem")
  system.terminate
}
```

In this program we again obtain a `Props` instance but this time for the class `Printer`. We then obtain the `ActorSystem` and then use this to crate the actors. In this case we create four actors. Each actor is an independent instance of the `Printer` Actor.

As you can see the version of `actorOf` used here takes the *Props* instance and a name string. The *name* parameter is optional. However as the name is used with

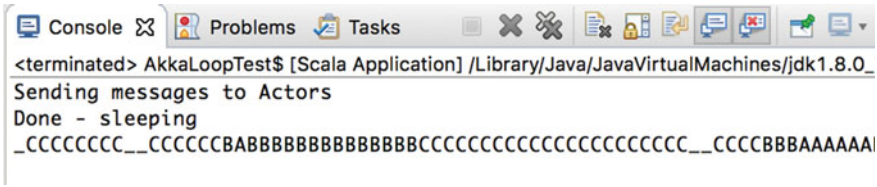

Fig. 31.8 Output from multiple concurrent Akka Actors

log messages it may be helpful during debugging. Note if a given name is already in use then an exception will be thrown.

Once we have created the four actors each is sent a message. As the main application will run concurrently to the actors themselves, all four messages should be sent before the first actor has a chance to complete the printout 500 strings. We should therefore see a mixture of different strings being output. This is shown in Fig. 31.8.

From this we can see that the four Actors run concurrently, sharing the available processor, with output coming from Actors 1, 2 and 3 clearly marked by the "A's, "B's" and "C's". The output from actor 4 is indicated by the "_"s.

31.8 The Akka Actor API

The `Actor` trait we looked at in the last couple of sections only defines a single abstract method, `receive`, which must be implemented to define the behaviour of the Actor. However, several other facilities are provided by the trait which are presented below:

- **Self**—a reference to the `ActorRef` of the actor.
- **Sender**—a reference to the sender of the message currently being processed by the Actor. Useful for returning results.
- **Context**—a reference to the actors execution context. This allows access to various facilities including the ability to create child actors (actors that execute under the control of the parent actor) and lifecycle operations.
- **supervisorStrategy**—this is used to allow a strategy to be defined that will control what should happen to child actors when various situations occur. This is discussed in the next chapter.

31.9 Actor Lifecycle

Each Actor can go through a number of states during its lifetime. This is illustrated in Fig. 31.9.

An actor goes through the following states during its lifetime:

- Does not exist—initially an actor does not exist.
- Actor is incarnated. When the `actorOf` method is called it assigns an incarnation of the actor described by the Props to a given path. A Path represents a route from the top-level location, via any parents, to the actor being created. If the actor is being created by the `ActorSystem` then it is a top-level actor. If it is being created by another actor then it is a child of that actor and its path is relative to the parent. An actor incarnation has a unique ID (a UID) and holds a reference to the current Actor Instance. Once the Actor Instance has been created from the actor incarnation then the `prestart` method is called on the actor instance. Note the fact that the Actor incarnation wraps the actor instance allows different actor instances to be used without the external references being aware of this.
- The lifecycle of the Actor incarnation ends when the actor is stopped. This can be done by called the `stop` method on the context. At that point the `postStop` method is called on the actor instance allowing for any clean-up operations to be performed. Once the actor has been stopped the name of the actor can be reused.

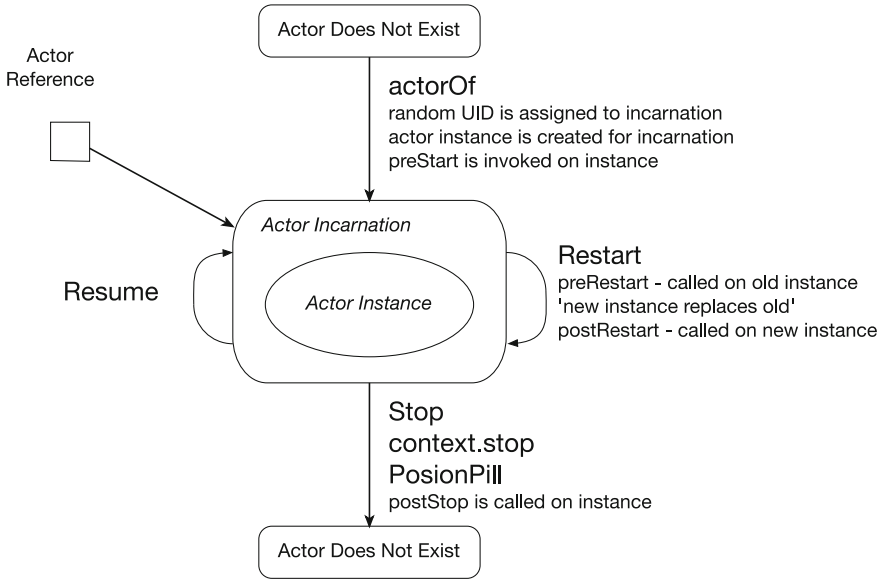

Fig. 31.9 Lifecycle of an Actor

- If an actor throws an exception it can be restarted if the exception scenario can be handled. The restart involves creating a new Actor instance to run the concurrent behaviour wrapped inside the same Actor incarnation. The lifecycle methods `preRestart` and `postRestart` are available to handle suitable processing of this scenario. However, you should note that `preRestart` is called on the old actor instance and the `postRestart` is called on the new actor instance.
- Resume is an alternative to Restart. That is, whereas Restart is used to restart a clean internal actor instance, Resume can be used to continue with the internal Actor Instance (as long as the internal state of the actor is stable).

To illustrate some of these ideas the following simple `LifecycleGreeter` actor defines overrides for the `preStart` and `postStop` lifecycle methods. These methods merely print out a string in this example.

The Actor's `receive` method is defined such that it will print out a message "Hello World" when it is sent the message "hello". It will also print out "Hello Whoever" if sent any other string the "stop". However, if sent the message string "stop", it will use the context property of the actor to stop itself.

The associated `LifecycleTest1` application then sends three messages to the actor, "hello", "Welcome" and "stop":

```
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorSystem

class LifecycleGreeter extends Actor {

  override def preStart(): Unit =
                { println("preStart") }
  override def postStop(): Unit =
                { println("postStop") }

  def receive = {
    case "hello" => println("Hello World")
    case "stop" => context.stop(self)
    case _ => println("Hello Whoever")
  }
}

object LifecycleTest1 extends App {
  val props = Props[LifecycleGreeter]
  val system = ActorSystem("mysystem")
  val actor = system.actorOf(props)
  actor ! "hello"
  actor ! "Welcome"
  actor ! "stop"
  system.terminate
}
```

The output from this program is shown below:

```
preStart
Hello World
Hello Whoever
postStop
```

As you can see from this the `prestart` method is called before the actor starts processing the messages sent to it. It then processes each of the messages in turn. The final message causes the actor to stop, and thus, the `postStop` method is called.

31.10 Akka Configuration

The Akka system is configured via the `ActorSystem`. That is, all configuration information is provided to the `ActorSystem` object. This can be done by specifying a `Config` object or by relying on the default locations used by Akka. If you wish to use the default configuration files, then you have a choice of the following files:

- application.conf
- application.json
- application.properties

These files are found in the root of the classpath. The Actor system merges information from these locations together to determine the default configuration information.

An example of a custom `application.conf` file is shown below. This example configures the information to be used by a TockTock actor.

```
TickTock {
 howlong = 2
 Ticker {
  message : "Ping"
 }
 Tocker {
  message : "Pong"
 }
}
```

This file can be loaded via the `com.typesafe.config.ConfigFactory.load` method. This method returns a `Config` object initialised using the default application files (e.g. application.config).

Alternatively you can explicitly create a `Config` instance. This is very flexible and allows you to load or specify any configuration information in any form. There are numerous factory methods that allow you to create `config` objects in different ways defined on the `ConfigFactory` object. For example, you can use the `parseString` method to create a `config` object based on the content of a string, for example:

```
Config config =
   ConfigFactory.parseString("something=somethingElse");
ActorSystem system =
   ActorSystem.create("myname", config);
```

In the following example, the Actor System is configured using the `ConfigFactory.load` method. The system then provides access to this configuration information via the `settings.config` object. The `config` object has numerous methods available on it that allow the values held in the configuration to be retrieved. For example `getInt` can be used to retrieve an integer representing how long an application should run.

```
import akka.actor.Props
import akka.actor.ActorSystem
import akka.actor.Actor

import com.typesafe.config.ConfigFactory
import akka.actor.ActorLogging

sealed abstract class Message
case class StartTicking ( tocker: ActorRef )
                                          extends Message
case object TickMessage extends Message
case object TockMessage extends Message

object ActorApp extends App {

  val system = ActorSystem("mysystem",
                           ConfigFactory.load)
  println(system.settings.config)

  val howLong =
      system.settings.config.getInt("TickTock.howlong")

  val config = system.settings.config

  println(s"Running for $howLong seconds")

  val ticker = system.actorOf(
     Props(classOf[TickActor],
             config.getString("TickTock.Ticker.message")),
     "Ticker")
  val tocker = system.actorOf(
     Props(classOf[TockActor],
             config.getString("TickTock.Tocker.message")),
     "Tocker")
  ticker ! StartTicking(tocker)

  Thread.sleep(1000 * howLong)
  system.terminate()

}
```

When the actors are created the ticker message or the tocker message are retrieved as strings from the `config` object and used to initialise the `TickActor` or `TockActor` respectively.

The actual actor classes themselves are given below:

```
class TickActor(msg: String)
                    extends Actor with ActorLogging {
  log.info(s"Creating Tick Actor with msg: $msg")
  override def receive = {
    case StartTicking(tocker) =>
      log.info(s"Starting... Tick ($msg)");
      tocker ! TockMessage
    case TickMessage =>
      log.info(msg);
      Thread.sleep(500); sender ! TockMessage
  }
}

class TockActor(msg: String)
                    extends Actor with ActorLogging {
  log.info("Creating Tock Actor with msg: $msg")
  override def receive = {
    case TockMessage =>
      log.info(msg);
      Thread.sleep(500); sender ! TickMessage
  }
}
```

Both these actors take a `msg` constructor parameter to be displayed when they receive a message they also send message between each other using the sender pseudo variable.

The output from this simple application is given below:

```
  Config(SimpleConfigObject({"TickTock":{"Ticker":
{"message":"Ping"},"Tocker":
{"message":"Pong"},"howlong":2},"akka...))
  Running for 2 seconds
  [INFO] [...] [akka://mysystem/user/
Tocker] Creating Tock Actor with msg: $msg
  [INFO] [...] [akka://mysystem/user/
Ticker] Creating TickActor with msg: Ping
  [INFO] [...] [akka://mysystem/user/Ticker] Starting... Tick (Ping)
  [INFO] [...] [akka://mysystem/user/Tocker] Pong
  [INFO] [...] [akka://mysystem/user/Ticker] Ping
  [INFO] [...] [akka://mysystem/user/Tocker] Pong
  [INFO] [...] [akka://mysystem/user/Ticker] Ping
  [INFO] [...] [akka://mysystem/user/Tocker] Message
```

It is possible to see from this the configuration information as well as its use in the application. Note that the `config` object also contains an extensive set of default values used by the Akka runtime.

31.11 Actor DSL

To make it easier to create simple Actors, the Akka library provides a domain-specific language (or DSL) specifically for their creation. For example, one off workers can be created very concisely using the `Act` trait. The `Act` trait is defined by the `akka.actor.ActorDSL._` type and includes supporting objects.

The *implicit* actor system serves as the `ActorRefFactory` required by the `actor` function when using the `Act` trait to create the actor. The `actor` function essentially uses the `ActorOf` functionality of the `ActorSystem` (or `ActorContent` if used inside an Actor) to create the new actor based on the `Act` trait. This is done by overriding the default behaviour of the `Act` traits `become` method. The overridden version of `become` essentially swaps the created Actors' `receive` method with that defined in the `Act` trait at runtime.

The following listing provides a simple example of creating a worker Actor:

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

object TestWorker extends App {
  implicit val system = ActorSystem("demo")

  val a = actor(new Act {
    become {
      case "A" => println("hello")
    }
  })

  a ! "A"

}
```

In this example the new `Act` instance defines the behaviour which will be used for the resulting actors `receive` method in the `become` function. Here if the string "A" is received, the actor will print out the string "hello". After defining this actor the message "A" is sent to the reference held to that actor.

The output from this program is:

```
Hello
```

The DSL also supports defining lifecycle behaviours. This is done using the `whenStarting`, `whenStopping`, `whenFailing` and `whenRestarted` functions, as shown below:

```
import akka.actor.ActorDSL.Act
import akka.actor.ActorDSL.actor
import akka.actor.ActorSystem

object LifecycleTest2 extends App {

  implicit val system = ActorSystem("demo")

  val a = actor(new Act {
    become {
      case "hello" => println("hello World")
      case "stop" => context.stop(self)
      case _ => println("Hello Whoever")
    }
    whenStarting { println("starting") }
    whenStopping { println("stopping") }
    whenFailing {case m@(cause, msg) =>println("Error")}
    whenRestarted { cause => println("whenRestarted") }
  })

  a ! "hello"
  a ! "Goodbye"
  a ! "stop"

  system terminate
}
```

The output from this program is:

```
starting
hello World
Hello Whoever
stopping
```

This is because the starting and stopping behaviours have been invoked but the failing and restarted ones have not.