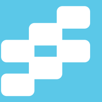


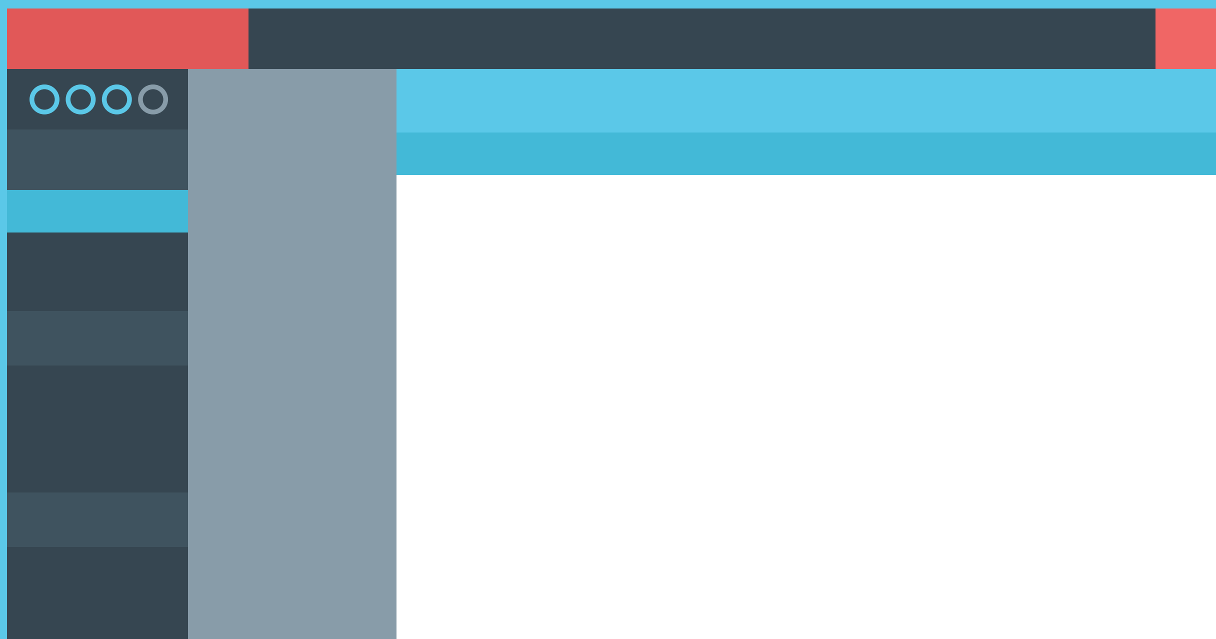
Go Reactive with

Play Framework, Akka and **Scala**

DOWNLOAD THE

Typesafe Reactive Platform

typesafe.com/platform/getstarted



Get More Refcardz! Visit Refcardz.com

CONTENTS

» The Actor Model
» Defining an Actor
» Creating Actors
» Defining Messages
» Fault Tolerance... and more!

Reactive Programming with Akka

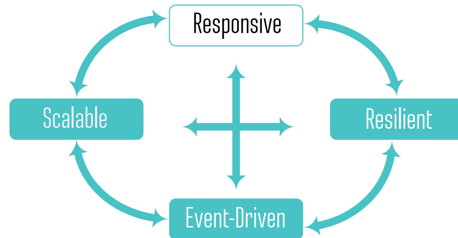
By Ryan Knight

INTRODUCTION

Akka is a toolkit and runtime for building Reactive Applications on the JVM. The Reactive Manifesto (reactivemanifesto.org) defines a reactive application as having four key properties:

1. Event-driven
2. Scalable
3. Resilient
4. Responsive

Event-driven means the application reacts to events by using an event-driven programming model. This allows the application to more effectively share resources by doing work only in response to outside events and messages. **Scalable** means the application is able to react to increasing load by making the architecture highly concurrent and distributed. When an application is **Resilient**, it can easily deal with a failure and recover. Instead of the application simply dying, it manages the failure through fault isolation so other parts of the application can keep running. The final property, **Responsive**, means the application is real-time, engaging, rich, and continues to respond even when there are application failures.

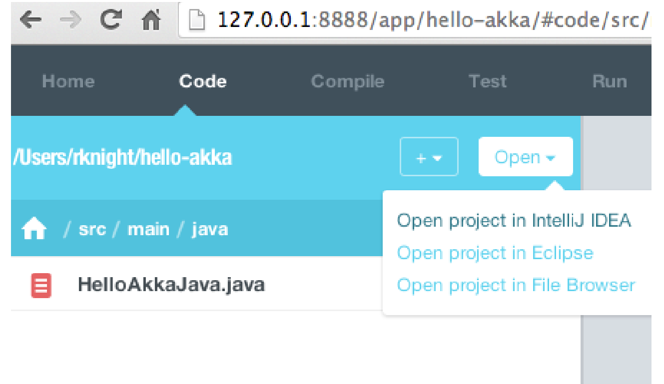


Akka was designed to enable developers to easily build reactive applications using a high level of abstraction. It does so in a very natural and simple way, without having to deal with low-level concepts like thread pools, mutexes, and deadlocks. It does so by leveraging the Actor Model of concurrency and fault-tolerance. This is a powerful model that allows the behavior and state of the application to be encapsulated and modeled as an **actor**. The key principle behind an **actor** is that the application only interacts with it through messages and never talks with it directly. This isolation allows Akka to manage the currency of the actor. There will be more discussion on this subject later.

CREATING A NEW AKKA APPLICATION

To create a new Akka application we will use the open source Typesafe Activator tool.

1. Download the Typesafe Activator: http://typesafe.com/platform/getstarted
2. Launch Typesafe Activator's UI.
3. Create a new app using the "Hello Akka!" template, which is a good place to begin learning as well as a good boilerplate project to start from.
4. Select **Run** to verify the app is working.
5. Optionally, open the new app in an IDE by selecting **Code** then **Open**, and then either **Open in Eclipse** or **Open in IntelliJ** depending on which IDE you are using:



As you make changes to your code, it will be recompiled and you will see any compile errors in the Compile tab of the activator:

This Refcard introduces Akka by modeling a simple robot named *AkkaBot*. The robot walks on four legs like a dog and has basic head controls. This tutorial can be used from either Java or Scala, and sample code has been provided in both Java and Scala. The AkkaBot will be modeled as an actor, with each leg and the head modeled as child actors. We will make the AkkaBot fault tolerant so that it reacts to failure and can deal with obstacles that come its way!

WHAT IS AKKA AND THE ACTOR MODEL?

The Actor Model was originally invented by Carl Hewitt in 1973 and has seen a recent resurgence in popularity for concurrency. In the Actor Model, an application is broken up into small units of execution called actors. These actors encapsulate the behavior and state of a part of the application. The first part of the definition of actors makes them sound very much like objects in a standard object-oriented design, which is true. However, what distinguishes an actor from a standard object is a third property: communication.

Actors never interact directly with each other and instead communicate via messages. The behavior of an actor is primarily defined by how it handles messages as they are received. Since the state of an actor is isolated from the rest of the system, an

Go Reactive with
Play Framework, Akka and Scala
DOWNLOAD THE
Typesafe Reactive Platform
typesafe.com/platform/getstarted

REACTIVE PROGRAMMING WITH AKKA

actor never has to worry about synchronization issues, such as thread locking, when modifying its state.

Where the Actor Model becomes even more interesting is when you combine actors into a supervisory hierarchy. By organizing the actors into a hierarchy, you can have parent actors that manage child actors and delegate computations to child actors. By leveraging the actor hierarchy, it becomes very easy to make an application fault-tolerant and scalable.

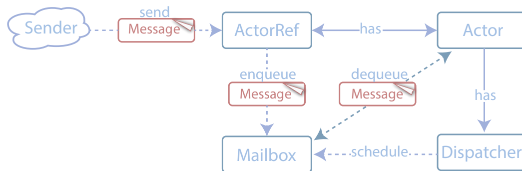
Actors in Akka are extremely lightweight since they are standard objects that only consume a few hundred bytes each. The only constraint in size is the amount of memory of they consume. This means that a single application can easily create thousands or even millions of concurrent actors.

An important part of the Actor programming model is that you never create actors directly, Instead, they are created by the **Actor System**. The returned value is not the actual actor; instead it is a reference to that actor called an **ActorRef**.

Each actor is then referenced through this **ActorRef** and it is this **ActorRef** that you send messages to. There are three advantages to using this level of indirection and never accessing the actor directly:

1. The actor can be transparently restarted after failure.
2. The location of the actor is transparent, allowing Akka to manage when and where the actor runs.
3. The actor is able to maintain mutable state without worrying about concurrent access to this state.

Shared mutable state is a cause of many concurrency problems. Akka actors only process a single message at a time. Accessing or mutating the internal state of an actor is fully thread safe since the thread is protected by the Actor Model.



The strong isolation principles of actors, together with the event-driven model and location transparency, make it easy to solve hard concurrency and scalability problems in an intuitive way. It abstracts away the underlying threading model and actor scheduling so you can focus on building your application and not infrastructure.

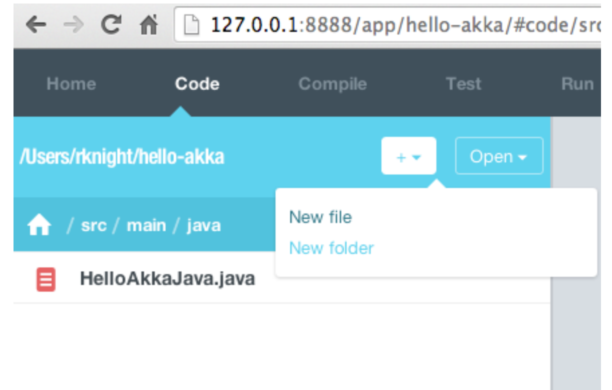
In Java, an actor is created by defining a class that extends **UntypedActor** and implements the **onReceive** method. In Scala, extend the actor trait and implement the receive method. It is in the **onReceive** method where you define the behavior, or how the actor should react to the different messages it receives.

To start building our robot, we now create a Robot Actor, the *AkkaBot*, which contains information about whether the robot is moving and its direction. It has an **onReceive** method that defines its behavior for how it should react upon receiving a **Move** message.

DEFINING AN ACTOR

You can define an actor in either your favorite IDE or in the Typesafe Activator. To do this in the activator:

1. Select the **Code** tab.
2. Navigate to either *src/main/java or src/main/scala*.
3. Click the plus sign (+) to add a new file:



4. Create the files **JavaBotMain.java** and **JavaAkkaBot.java** or **ScalaBotMain.scala** and **ScalaAkkaBot.scala** in the source directory. In those files, enter the Java or Scala code below:

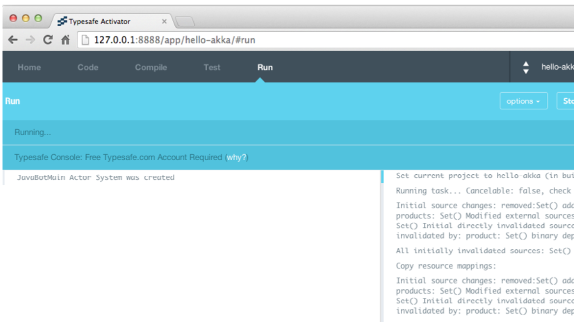
```
// Java code for JavaBotMain.java
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

public class JavaBotMain {

    public static void main(String[] args) {
        // Create the 'helloakka' actor system
        final ActorSystem system = ActorSystem.
create("helloakka");

        // Create the 'AkkaBot' actor
        final ActorRef akkaBot = system.actorOf(Props.
create(AkkaBot.class), "akkaBot");

        System.out.println("JavaBotMain Actor System was
created");
    }
}


// Java code for JavaAkkaBot.java
import akka.actor.UntypedActor;

public class JavaAkkaBot extends UntypedActor {
    boolean moving = false;

    public void onReceive(Object message) {
        unhandled(message);
    }
}


//Scala code for ScalaBotMain.scala
import akka.actor.{Props, ActorSystem}

object ScalaBotMain extends App {
  // Create the 'helloakka' actor system
  val system = ActorSystem("helloakka")

  // Create the 'akkaBot' actor
  val akkaBot = system.actorOf(Props[ScalaAkkaBot], "akkaBot")

  println("ScalaBotMain Actor System was created")
}

//Scala code for ScalaAkkaBot.scala

import akka.actor.Actor

class ScalaAkkaBot extends Actor {
  var moving: Boolean = false
  def receive = {
    case msg => unhandled(msg)
  }
}
```

Now we can test the initial implementation by running the code in the activator:

1. In Typesafe Activator, select the **Run** tab.

Typesafe

2. Change the **Main Class** to be the main class you define (either **JavaBotMain** or ScalaBotMain).
3. In the output on the left hand side, you should see the output from application:



CREATING ACTORS

Now we have the defined the actor, let's create an instance of this actor. In Akka, you can't create an instance of an actor the normal way using "new". Instead, you have to do so through a factory – in this case, an **ActorSystem**. What is returned from this factory is not an instance of the actor itself, but an **ActorRef** pointing to our actor instance.

This level of indirection adds a lot of power and flexibility. It enables location transparency meaning that the **ActorRef** can, while retaining the same semantics, represent an instance of the running actor in-process or on a remote machine (i.e. location doesn't matter). This also means that the runtime can if needed optimize the system by changing an actor's location or the application's topology while it is running. Another thing that this level of indirection enables is the "let it crash" model of failure management in which the system can heal itself by crashing and restarting faulty actors.

The factory in Akka to create actors is the **ActorSystem**, and is to some extent similar to Spring's **BeanFactory**. It also acts as a container for your actors, managing their life cycles. You create also an actor through the **actorOf** factory method. This method takes a configuration object called Props and a name. Actor (and **ActorSystem**) names are important in Akka; you use them when looking actors up and configuring them in the configuration file.

The Actor System in Akka is also much more than just a factory for creating actors. It manages the entire lifecycle of the actor, maintains the execution context (thread pool) in which actors run, a scheduling service, an event stream of what is happening and more.

Previously, we added the basic code for creating an **ActorSystem** and the *AkkaBot* in **JavaBotMain.java**:

```
final ActorSystem system = ActorSystem.create("helloakka");
final ActorRef akkaBot = system.actorOf(Props.
create(JavaAkkaBot.class), "akkaBot");
```

If you are using Scala, the code in **ScalaBotMain.scala** looks similar:

```
val system = ActorSystem("helloakka")
val akkaBot = system.actorOf(Props[ScalaAkkaBot], "akkaBot")
```

Creating an actor using the Actor System directly creates the actor at the top of the hierarchy. Actors can also be created as children of other actors using an actor's local **Actor Context**. The Actor Context contains information about the Actor System relevant to each actor, such as who its parent and children are. When an actor uses this context to create another actor, the new actor becomes a child actor. In this way the actor hierarchy gets built out.

DEFINING MESSAGES

An actor does not have a public API in terms of methods that you can invoke. Instead, its public API is defined through messages that the actor handles. Messages can be of arbitrary type (any subtype of object in Java or any in Scala). This means that we can send boxed primitive values (such as **String, Integer, Boolean**, etc.) as messages or plain data structures like arrays, collection types, and value objects.

However, since the messages are the actor's public API, you should define messages with good names and rich semantic and domain specific meaning, even if it's just wrapping your data type. This makes it easier to use, understand, and debug actor-based systems. In Java, this is typically done using public static classes. In Scala, this is done with case objects or classes. Then when someone is looking at the actor code, they can easily see what messages are handled for the Actor. This also makes the messages part of the auto-generated Java or Scala API docs.

Now we want to define three different messages:

- **Move** starts the robot moving
- **Stop** stops the robot
- **GetRobotState** gets the robot's state
- **RobotState** holds the robot's current state

Let's start by defining the messages in Java by putting them inside the **AkkaBot** class. It is very important that the messages we create are immutable (meaning that they cannot be changed). If not, we run the risk of accidentally sharing mutable state between two different actors, which will violate the Actor Model.

```
// Inside the AkkaBot.java code add the following inner
classes:
public enum Direction { FORWARD, BACKWARDS, RIGHT, LEFT }
public static class Move {
    public final Direction direction;
    public Move(Direction direction) { this.direction =
direction; }
}
public static class Stop {}
public static class GetRobotState {}
public static class RobotState {
    public final Direction direction;
    public final boolean moving;
    public RobotState(Direction direction, boolean moving) {
        this.direction = direction;
        this.moving = moving;
    }
}
```

In Scala, case classes and case objects make excellent messages since they are immutable and have support for pattern matching – something we will take advantage of in the actor when processing the messages the actor receives.

```
// Add the following to the ScalaAkkaBot.scala file
object ScalaAkkaBot {
  sealed abstract class Direction
  case object FORWARD extends Direction
  case object BACKWARDS extends Direction
  case object RIGHT extends Direction
  case object LEFT extends Direction
  case class Move(direction:Direction)
  case object Stop
  case object GetRobotState
  case class RobotState(direction: Direction, moving: Boolean)
}
```

TELL THE ACTOR (TO DO SOMETHING)

All communication with actors is done through asynchronous message passing. This is what makes actors reactive. An actor doesn't do anything unless it's been told to do something, and you tell it to do something by sending the message. Sending a message asynchronously means that the sender does not stick around waiting for the message to be processed by the recipient actor. Instead, the actor hands the message off by putting it on the recipient's mailbox and is then free to do something more important than waiting for the recipient to react on the message. The actor's mailbox is essentially a message queue and has ordering semantics. This guarantees that the ordering of multiple

Typesafe

messages sent from the same actor is preserved, while the same messages can be interleaved with the messages sent by another actor.

When the actor is not processing messages, it is in a suspended state in which it does not consume any resources apart from memory.

You tell an actor to do something by sending a message into the tell method on the **ActorRef**. This method puts the message on the actor's mailbox and then returns immediately.

Internally, an actor responds to messages by overriding the **onReceive** method. That method takes a single parameter - a message of type **Object** for Java or **Any** for Scala. In the **onReceive** method, the primary behavior of an actor is defined. This behavior can be any standard logic such as modifying the internal state of an actor, creating or calling other actors, business logic or any other behavior.

First let's define the **onRecieve** for the **Move** and **Stop** messages. If you are doing this in Java make the following changes to the **JavaAkkBot.java:**

```
// Add the following field:
Direction direction = Direction.FORWARD

// Change onReceive method to be
public void onReceive(Object message) {
    if (message instanceof Move) {
        direction = ((Move) message).direction;
        moving = true;
    }
    else if (message instanceof Stop) {
        moving = false;
    }
    else {
        unhandled(message);
    }
}
```

For Scala, make the following changes in **ScalaAkkaBot:**

```
/// add the following import inside the ScalaAkkaBot class:
import ScalaAkkaBot._

// Add the following field
var direction: Direction = FORWARD

//Change the onReceive method to be:

def receive = {
  case Move(newDirection) =>
    moving = true
    direction = newDirection
    println(s"I am now moving $direction")
  case Stop =>
    moving = false
    println(s"I stopped moving")
  case msg => unhandled(msg)
}
```

If you are using Akka from Scala, then you can also use the ! alias, called the bang operator.

We can now test out the actors by sending them some simple commands from the Bot Main app.

For Java, add the following to the **JavaBotMain.java** file:

```
akkaBot.tell(new JavaAkkaBot.Move(JavaAkkaBot.Direction.
FORWARD), ActorRef.noSender());
akkaBot.tell(new JavaAkkaBot.Move(JavaAkkaBot.Direction.
BACKWARDS), ActorRef.noSender());
akkaBot.tell(new JavaAkkaBot.Stop(), ActorRef.noSender());
```

For Scala, add the following to the **ScalaBotMain.scala** file:

```
akkaBot ! ScalaAkkaBot.Move(ScalaAkkaBot.FORWARD)
akkaBot ! ScalaAkkaBot.Move(ScalaAkkaBot.BACKWARDS)
akkaBot ! ScalaAkkaBot.Stop
```

When you run the application, you should see some basic logging of the application such as:

```
JavaBotMain Actor System was created
I am now moving FORWARD
I am now moving BACKWARDS
I stopped moving
```

THE 'SELF' REFERENCE

Sometimes the communication pattern is not just one-way, but instead lends itself towards **request-reply**. One explicit way of doing that is by adding a reference of yourself as part of the message so the receiver can use that reference to send a reply back to you. This is such a common scenario that Akka directly supports it. For every message you send, you have the option of passing along the sender reference (the actor's ActorRef). If you are sending a message from within an actor, then you have access to your own ActorRef through the **self** reference. In Java, you can access the **self** reference through the **getSelf()** method. In Scala, use **self**.

```
// From within an Actor
akkaBotRef.tell(new JavaAkkaBot.Move(JavaAkkaBot.Direction.
FORWARD), getSelf());
```

In Scala, this is simplified a bit. Scala has something called **implicit parameters**, which allows you to automatically and transparently pass parameters into methods. We can take advantage of this feature to automatically pass along the sender reference when you send a message to another actor.

This code will, if invoked from within "Actor A," automatically pass along the ActorRef of "Actor A" as the sender of this message:

```
// Within an Actor the sender is automatically the self
ActorRef
akkaBotRef ! ScalaAkkaBot.Move(ScalaAkkaBot.FORWARD)
```

In Java, if you are not inside an actor or do not want to pass the sender, use **ActorRef.noSender()** instead. In Scala, if the **tell** method is called from outside an actor, then the implicit sender parameter will not be found. In this case, the default parameter for tell is used, which is **Actor.noSender.** An example of both these cases is in the Java or Scala main classes where messages were sent to the robot without a sender.

THE 'SENDER' REFERENCE

When an actor receives and processes a message in the **onReceive** method, the sender of that message is available. Since each message is paired with its unique sender reference, the "current" sender reference will change with each new message you process. In Java, you can access it using the **getSender()** method. In Scala, just use the **sender** reference. For example, to send a message back to the sender of the message that is being handled do:

```
// Java code
getSender().tell(new Greeting(greeting), getSelf());

// Scala code
sender ! Greeting(greeting)
```

If you need to use a specific sender reference after some asynchronous processing, like after calling other actors, you will need to store a reference to the sender in a variable. The reason is that the sender might change if other messages or processing happens in the meantime.

In some cases there might not be a sender, like when a message is sent from outside an actor or the sender was **Actor.noSender**. In these cases, sending a message to sender would cause it to be sent to **dead letters**. The **dead-letter** actor is where all unhandled messages end up. For debugging and auditing purposes, you can watch for messages to the **dead-letter** actor.

Typesafe

ACTOR HIERARCHIES

The real power of the Actor Model is with actor hierarchies. An actor hierarchy is when a parent actor creates and supervises child actors. This structure helps avoid cascading failures that can take down an entire system by isolating and supervising child nodes.

Creating child actors is very similar to creating top level actors - the only difference is the context the child actors are created in. The actor context can be thought of as where in the actor hierarchy an actor lives. As an example, lets create a Bot Master that creates several children. For Java, add the class **JavaBotMaster** that creates several children in the constructor:

```
import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;

public class JavaBotMaster extends UntypedActor {

    public JavaBotMaster() {
        for (int indx = 0; indx < 10; indx++) {
            context().actorOf(Props.create(JavaAkkaBot.class));
        }
    }

    public void onReceive(Object message) {}
}
```

For Scala, add the class **ScalaBotMaster**:

```
import akka.actor.{Props, Actor}

class ScalaBotMaster extends Actor {

  import ScalaBotMaster._
  import ScalaAkkaBot._

  for (indx <- 1 to 10) {
    context.actorOf(Props[ScalaAkkaBot])
  }


  def receive = {
    case _ =>
  }
}
```

In both Java and Scala, we get the local actor context and create new actors in that context. That makes all of the **ScalaAkkaBots** created children of the Bot Master. Now that the Bot Master has children it can interact with the children directly. To do this, lets add a simple method to start the child bots moving.

For Java, add the **StartChildBots** static class and modify the **onReceive**:

```
public void onReceive(Object message) {
    if (message instanceof StartChildBots) {
        for (ActorRef child : getContext().getChildren()) {
            System.out.println("Master started moving " +
child);
            child.tell(new JavaAkkaBot.Move(JavaAkkaBot.
Direction.FORWARD), getContext().self());
        }
        System.out.println("Master has started children bots");
    }
}

public static class StartChildBots {}
```

For Scala, add a case object **StartChildBots**:

```
object ScalaBotMaster {
  case object StartChildBots
}
```

Then modify the receive function:

```
def receive = {
  case StartChildBots =>
    context.children.foreach { child =>
      println(s"child=$child")
      child ! Move(FORWARD)
    }
    println("Master has started children bots.")
}
```

What this does is look in the Master Bot's context for all of its children. It then iterates through the children and sends them a message.

To test this out, lets modify the Akka Bots so they print out their own path to make it easier to see where trace statements are coming from. In Java, change the **println's** to be:

```
System.out.println(self().path() + " is now moving " +
direction);
```

Then modify the main class to start the Bot Master instead:

```
// Create the 'AkkaBot' actor
final ActorRef akkaBot = system.actorOf(Props.
create(JavaBotMaster.class), "akkaBotMaster");
akkaBot.tell(new JavaBotMaster.StartChildBots(), ActorRef.
noSender());
```

In Scala, change them to be:

```
System.out.println(self().path() + " is now moving " +
direction);
```

Finally, modify the Scala main class:

```
val akkaBotMaster = system.actorOf(Props[ScalaBotMaster],
"akkaBotMaster")
```

The method call to **self.path** gets the path of the current actor. This path will be something like:

```
akka://helloakka/user/akkaBotMaster/$f
```

The **$f** is the name of the actor. Since we didn't give it an explicit name, Akka gave it a randomly chosen name. From the path we can see this actor is a child of **akkaBotMaster**.

This exercise should give you a good sense for how the context is where an actor lives in the hierarchy of other actors, since you can see how an actor can get its parents and children. A child actor can be referenced via its parent.

FAULT-TOLERANT AND SELF-HEALING

The reason actor hierarchies are so powerful is they provide a way to build systems that are fault-tolerant and self-healing. This can be done by isolating the errors to child actors and monitoring those actors for failures.

Failure is a normal part of any application and can happen from a wide variety of causes. With actors the failures can happen during startup, message processing or other lifecycle events. What is unique about the actor model is a failure doesn't cause the entire application to crash; rather the fault is isolated to an individual actor.

There are two ways actors can deal with failures. The first is through customizing the supervision strategy. Each actor has a default supervisor strategy that defines how it handles failures in its children. The default supervisor strategy for an exception is to restart the actor. This supervisor strategy can be overridden to define different behavior. Besides restart, other possible actions include escalating the exception up the hierarchy, simply resuming the actor or stopping the actor.

Resuming and restarting might seem to be the same, however there is one key difference. When an actor is restarted a new instance of the actor is created, which resets its state. If an actor is resumed the actor state is maintained. With both strategies the message that caused

Typesafe

the fault is lost, but the actor does not lose any of its other messages waiting in the mailbox.

The second way a parent can manage the failures of its children is to add watches to the children. Then the parent receives a Terminated message when the child is terminated. As an example let's assume that if a child robot stops, we want to do some special processing. To do this we simply add a watch after creating the robot.

In Java, this would be done with the following changes. In the constructor of the Java Bot Master add a watch:

```
ActorRef child = context().actorOf(Props.create(JavaAkkaBot.
class)); context().watch(child);
```

And then handle the case of receiving a terminated message in the **onReceive** method:

```
else if (message instanceof Terminated) {
    System.out.println("Child has stopped ... starting a new
one");
    ActorRef child = context().actorOf(Props.
create(JavaAkkaBot.class));
    context().watch(child);
}
```

In Scala, we would add the following to the creation of the children:

```
val child = context.actorOf(Props[ScalaAkkaBot])
context.watch(child)
```

And then modify the **receive** function to handle the **Terminated** message:

```
case akka.actor.Terminated(ref) =>
  println("Child has stopped ... starting a new one")
  val child = context.actorOf(Props[ScalaAkkaBot])
  context.watch(child)
```

Now we need to make the child randomly fail. This can be done in Java by adding the field to the **Move** message handler in the **JavaAkkaBot:**

```
Random rand = new java.util.Random();
int nextInt = rand.nextInt(10);
if ( (nextInt % 2) == 0) {
    context().stop(self());
}
```

For Scala, add the following to the **Move** message handler in the **ScalaAkkaBot**:

```
val random = scala.util.Random

println(s"${self.path} am now moving $direction")
if ((random.nextInt(10) % 10) == 0) {
  context.stop(self)
}
```

You can now see how the actors deal with failure. Uncaught exceptions also stop the actor allowing the supervisor to decide what to do.

FURTHER LEARNING

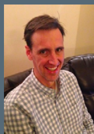
Now that you have learned the basics of Akka check other Akka Activator templates that will teach you about topics including supervision, clustering, and dependency injection. For a curated list of these templates, visit: http://akka.io/downloads/

The Akka reference documentation is a very complete resource: http://akka.io/docs/

For an in-depth slide presentation about Akka, check out the creator of Akka's slides, Introducing Akka: http://www.slideshare.net/jboner/introducing-akka

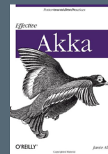
The Scala Days 2013 site has a number of recorded Akka presentations: http://scaladays.org/ny2013/

ABOUT THE AUTHOR



Ryan Knight is a consultant and trainer for Typesafe where he helps others learn and use Scala, Akka and Play. Ryan frequently does training and presentations at conferences around the world, such as JavaOne, Devoxx, and many other Java get-togethers. He has over 15 years of experience with enterprise software development. He first started consulting with Enterprise Java in 1999 with the Sun Java Center. Since then he has worked with a wide variety of companies, such as the Oracle, LDS Church, Williams Pipeline, Riot Games, Sony, T-Mobile, Deloitte and the State of Louisiana. This has given him experience with wide range of business, such as genealogy, telecommunications, finance and video games.

RECOMMENDED BOOK



Effective Akka distills years of experience building distributed, asynchronous, high-performance applications using Akka. Written by Jamie Allen, creator of Akka, this book includes deep-dives into domain-driven and work-distribution actor applications, interaction modeling, actors vs. classes, common patterns, actor monitoring, and more.

BUY NOW

BROWSE OUR COLLECTION OF **250+ FREE RESOURCES**, INCLUDING:

RESEARCH GUIDES: Unbiased insight from leading tech experts

REFCARDZ: Library of 200+ reference cards covering the latest tech topics

COMMUNITIES: Share links, author articles, and engage with other tech experts

JOIN NOW

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

Version 1.0 $7.95

Copyright © 2014 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.