8
Actors

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*
>
> *Leslie Lamport*

Throughout this book, we have concentrated on many different abstractions for concurrent programming. Most of these abstractions assume the presence of shared memory. Futures and promises, concurrent data structures, and software transactional memory are best suited for shared memory systems. While the shared memory assumption ensures that these facilities are efficient, it also limits them to applications running on a single computer. In this chapter, we consider a programming model that is equally applicable to a shared-memory machine or a distributed system, namely, the **actor model**. In the actor model, the program is represented by a large number of entities that execute computations independently, and communicate by passing messages. These independent entities are called **actors**.

The actor model aims to resolve issues associated with using shared memory, such as data races or synchronization, by eliminating the need for shared memory altogether. Mutable state is confined within the boundaries of one actor, and is potentially modified when the actor receives a message. Messages received by the actor are handled serially, one after another. This ensures that the mutable state within the actor is never accessed concurrently. However, separate actors can process the received messages concurrently. In a typical actor-based program, the number of actors can be orders of magnitude greater than the number of processors. This is similar to the relationship between processors and threads in multi-threaded programs. The actor model implementation decides when to assign processor time to specific actors, to allow them to process messages.

The true advantage of the actor model becomes apparent when we start distributing the application across multiple computers. Implementing programs that span across multiple machines and devices that communicate through a computer network is called **distributed programming**. The actor model allows you to write programs that run inside a single process, multiple processes on the same machine, or on multiple machines that are connected with a computer network. Creating actors and sending messages is oblivious and independent of the location of the actor. In distributed programming, this is called **location transparency**. Location transparency allows you to design distributed systems without having the knowledge about the relationships in the computer network.

In this chapter, we will use the Akka actor framework to learn about the actor concurrency model. Specifically, we cover the following topics:

- Declaring actor classes and creating actor instances
- Modeling actor state and complex actor behaviors
- Manipulating the actor hierarchy and the life cycle of an actor
- The different message-passing patterns used in actor communication
- Error recovery using the built-in actor supervision mechanism
- Using actors to transparently build concurrent and distributed programs

We will start by studying the important concepts and terminology in the actor model, and learning the basics of the actor model in Akka.

Working with actors

In the actor programming model, the program is run by a set of concurrently executing entities called actors. Actor systems resemble human organizations, such as companies, governments, or other large institutions. To understand this similarity, we consider the example of a large software company.

In a software company like Google, Microsoft, Amazon, or Typesafe, there are many goals that need to be achieved concurrently. Hundreds or thousands of employees work towards achieving these goals, and are usually organized in a hierarchical structure. Different employees work at different positions. A team leader makes important technical decisions for a specific project, a software engineer implements and maintains various parts of a software product, and a system administrator makes sure that the personal workstations, servers, and various equipments are functioning correctly. Many employees, such as the team leader, delegate their own tasks to other employees who are lower in the hierarchy than themselves. To be able to work and make decisions efficiently, employees use e-mails to communicate.

When an employee comes to work in the morning, he inspects his e-mail client and responds to the important messages. Sometimes, these messages contain work tasks that come from his boss or requests from other employees. When an e-mail is important, the employee must compose the answer right away. While the employee is busy answering one e-mail, additional e-mails can arrive, and these e-mails are enqueued in his e-mail client. Only once the employee is done with one e-mail is he able to proceed to the next one.

In the preceding scenario, the workflow of the company is divided into a number of functional components. It turns out that these components closely correspond to different parts of an actor framework. We will now identify these similarities by defining the parts of an actor system, and relating them to their analogs in the software company.

An **actor system** is a hierarchical group of actors that share common configuration options. An actor system is responsible for creating new actors, locating actors within the actor system, and logging important events. An actor system is an analog of the software company itself.

An **actor class** is a template that describes a state internal to the actor, and how the actor processes the messages. Multiple actors can be created from the same actor class. An actor class is an analog of a specific position within the company, such as a software engineer, a marketing manager, or a recruiter.

An **actor instance** is an entity that exists at runtime and is capable of receiving messages. An actor instance might contain mutable state, and can send messages to other actor instances. The difference between an actor class and an actor instance directly corresponds to the relationship between a class and an object instance of that class in object-oriented programming. In the context of the software company example, an actor instance is analogous to a specific employee.

A **message** is a unit of communication that actors use to communicate. In Akka, any object can be a message. Messages are analogous to e-mails sent within the company. When an actor sends a message, it does not wait until some other actor receives the message. Similarly, when an employee sends an e-mail, he does not wait until the e-mail is received or read by the other employees. Instead, he proceeds with his own work; an employee is too busy to wait. Multiple e-mails might be sent to the same person concurrently.

The **mailbox** is a part of memory that is used to buffer messages, specific to each actor instance. This buffer is necessary, as an actor instance can process only a single message at a time. The mailbox corresponds to an e-mail client used by an employee. At any point, there might be multiple unread e-mails buffered in the e-mail client, but the employee can only read and respond to them one at a time.

An **actor reference** is an object that allows you to send messages to a specific actor. This object hides information about the location of the actor from the programmer. Actor might run within separate processes or on different computers. The actor reference allows you to send a message to an actor irrespective of where the actor is running. From the software company perspective, an actor reference corresponds to the e-mail address of a specific employee. The e-mail address allows us to send an e-mail to an employee, without knowing anything about the physical location of the employee. The employee might be in his office, on a business trip, or on a vacation, but the e-mail will eventually reach him no matter where he goes.

A **dispatcher** is a component that decides when actors are allowed to process messages, and lends them computational resources to do so. In Akka, every dispatcher is at the same time an execution context. The dispatcher ensures that actors with non-empty mailboxes eventually get run by a specific thread, and that these messages are handled serially. A dispatcher is best compared to the e-mail answering policy in the software company. Some employees, such as the technical support specialists, are expected to answer e-mails as soon as they arrive. Software engineers sometimes have more liberty: they can choose to fix several bugs before inspecting their e-mails. The janitor spends his day working around the office building, and only takes a look at his e-mail client in the morning.

To make these concepts more concrete, we start by creating a simple actor application. This is the topic of the next section, in which we learn how to create actor systems and actor instances.

Creating actor systems and actors

When creating an object instance in an object-oriented language, we start by declaring a class, which can be reused by multiple object instances. We then specify arguments for the constructor of the object. Finally, we instantiate an object using the `new` keyword and obtain a reference to the object.

Creating an actor instance in Akka roughly follows the same steps as creating an object instance. First, we need to define an actor class, which defines the behavior of the actor. Then, we need to specify the configuration for a specific actor instance. Finally, we need to tell the actor system to instantiate the actor using the given configuration. The actor system then creates an actor instance and returns an actor reference to that instance. In this section, we will study these steps in more detail.

An actor class is used to specify the behavior of an actor: it describes how the actor responds to messages and communicates with other actors, encapsulates actor state, and defines the actor's startup and shutdown sequences. We declare a new actor class by extending the `Actor` trait from the `akka.actor` package. This trait comes with a single abstract method `receive`. The `receive` method returns a partial function object of type `PartialFunction[Any, Unit]`. This partial function is used when an actor receives a message of `Any` type. If the partial function is not defined for the message, the message is discarded.

In addition to defining how an actor receives messages, the actor class encapsulates references to objects used by the actor. These objects comprise the actor's state. Throughout this chapter, we use Akka's `Logging` object to print to the standard output. In the following code, we declare a `HelloActor` actor class, which reacts to a `hello` message specified with the `hello` constructor argument. The `HelloActor` class contains a `Logging` object `log` as part of its state. The `Logging` object is created using the `context.system` reference to the current actor system, and the `this` reference to the current actor. The `HelloActor` class defines a partial function in the `receive` method, which determines if the message is equal to the `hello` string argument, or to some other object called `msg`. When an actor defined by the `HelloActor` class receives a hello message, it prints the message using the `Logging` object `log`. Otherwise, it prints that it received an unexpected message, and stops by calling `context.stop` on the actor reference `self`, which represents the current actor. This is shown in the following code snippet:

```
import akka.actor._
import akka.event.Logging
class HelloActor(val hello: String) extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case `hello` =>
      log.info(s"Received a '$hello'... $hello!")
    case msg     =>
      log.info(s"Unexpected message '$msg'")
      context.stop(self)
  }
}
```

Declaring an actor class does not create a running actor instance. Instead, the actor class serves as a blueprint for creating actor instances. The same actor class can be shared by many actor instances. To create an actor instance in Akka, we need to pass information about the actor class to the actor system. However, an actor class such as `HelloActor` is not sufficient for creating an actor instance; we also need to specify the `hello` argument. To bundle the information required for creating an actor instance, Akka uses objects called **actor configurations**.

An actor configuration contains information about the actor class, its constructor arguments, mailbox, and dispatcher implementation. In Akka, an actor configuration is represented with the `Props` class. A `Props` object encapsulates all the information required to create an actor instance, and can be serialized or sent over the network.

To create `Props` objects, it is a recommended practice to declare factory methods in the companion object of the actor class. In the following companion object, we declare two factory methods called `props` and `propsAlt`, which return `Props` objects for the `HelloActor` class, given the `hello` argument:

```
object HelloActor {
  def props(hello: String) = Props(new HelloActor(hello))
  def propsAlt(hello: String) = Props(classOf[HelloActor], hello)
}
```

The `props` method uses an overload of the `Props.apply` factory method, which takes a block of code by creating the `HelloActor` class. This block of code is invoked every time an actor system needs to create an actor instance. The `propsAlt` method uses another `Props.apply` overload, which creates an actor instance from the `Class` object of the actor class, and a list of constructor arguments. The two declarations are semantically equivalent.

The first `Props.apply` overload takes a closure that calls the actor class constructor. If we are not careful, the closure can easily catch references to the enclosing scope. When this happens, these references become a part of the `Props` object. Consider the `defaultProps` method in the following utility class:

```
class HelloActorUtils {
  val defaultHi = "Aloha!"
  def defaultProps() = Props(new HelloActor(defaultHi))
}
```

Sending the `Props` object that is returned by the `defaultProps` method over the network requires sending the enclosing `HelloActorUtils` object captured by the closure, incurring additional network costs.

Furthermore, it is particularly dangerous to declare a `Props` object within an actor class, as it can catch a `this` reference to the enclosing actor instance. It is safer to create the `Props` objects exactly as they were shown in the `propsAlt` method.

> Avoid creating the `Props` objects within actor classes to prevent accidentally capturing the actor's `this` reference. Wherever possible, declare `Props` inside factory methods in top-level singleton objects.

The third overload of the `Props.apply` method is a convenience method that can be used with actor classes with zero-argument constructors. If `HelloActor` defines no constructor arguments, we can write `Props[HelloActor]` to create a `Props` object.

To instantiate an actor, we pass an actor configuration to the `actorOf` method of the actor system. Throughout this chapter, we will use our custom actor system instance called `ourSystem`. We define `ourSystem` using the `ActorSystem.apply` factory method:

```
lazy val ourSystem = ActorSystem("OurExampleSystem")
```

We can now create and run `HelloActor` by calling the `actorOf` method on the actor system. When creating a new actor, we can specify a unique name for the actor instance with the argument called `name`. Without explicitly specifying the `name` argument, the actor system automatically assigns a unique name to the new actor instance. The `actorOf` method does not return an instance of the `HelloActor` class. Instead, it returns an actor reference object of the type `ActorRef`.

After creating a `HelloActor` instance `hiActor`, which recognizes the `hi` messages, we send it a message `hi`. To send a message to an Akka actor, we use the `!` operator (pronounced as *tell* or *bang*). For clarity, we then pause the execution for one second by calling `sleep`, and give the actor some time to process the message. We then send another message `hola`, and wait one more second. Finally, we terminate the actor system by calling its `shutdown` method. This is shown in the following program:

```
object ActorsCreate extends App {
  val hiActor: ActorRef =
    ourSystem.actorOf(HelloActor.props("hi"), name = "greeter")
  hiActor ! "hi"
  Thread.sleep(1000)
  hiActor ! "hola"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Upon running this program, the `hiActor` instance first prints that it received a `hi` message. After one second, it prints that it received a `hola`, an unexpected message, and terminates.

Managing unhandled messages

The `receive` method in the `HelloActor` example was able to handle any kind of messages. When the message was different from the pre-specified `hello` argument, such as `hi` used previously, the `HelloActor` actor reported this in the default case. Alternatively, we could have left the default case unhandled. When an actor receives a message that is not handled by its `receive` method, the message is wrapped into an `UnhandledMessage` object and forwarded to the actor system's event stream. Usually, the actor system's event stream is used for logging purposes.

We can override this default behavior by overriding the `unhandled` method in the actor class. By default, this method publishes the unhandled messages on the actor system's event stream. In the following code, we declare a `DeafActor` actor class, whose `receive` method returns an empty partial function. An empty partial function is not defined for any type of message, so all the messages sent to this actor get passed to the `unhandled` method. We override it to output the `String` messages to the standard output. We pass all other types of message to the actor system's event stream by calling `super.unhandled`. The following code snippet shows the `DeafActor` implementation:

```
class DeafActor extends Actor {
  val log = Logging(context.system, this)
  def receive = PartialFunction.empty
  override def unhandled(msg: Any) = msg match {
    case msg: String => log.info(s"I do not hear '$msg'")
    case msg         => super.unhandled(msg)
  }
}
```

Let's test a `DeafActor` class in an example. The following program creates a `DeafActor` instance named `deafy`, and assigns its actor reference to the value `deafActor`. It then sends the two messages `deafy` and `1234` to `deafActor`, and shuts down the actor system:

```
object ActorsUnhandled extends App {
  val deafActor: ActorRef =
    ourSystem.actorOf(Props[DeafActor], name = "deafy")
  deafActor ! "hi"
  Thread.sleep(1000)
  deafActor ! 1234
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Running this program shows that the first message, `deafy`, is caught and printed by the `unhandled` method. The `1234` message is forwarded to the actor system's event stream, and is never shown on the standard output.

An attentive reader might have noticed that we could have avoided the `unhandled` call by moving the case into the `receive` method, as shown in the following `receive` implementation:

```
def receive = {
  case msg: String => log.info(s"I do not hear '$msg'")
}
```

This definition of `receive` is more concise, but is inadequate for more complex actors. In the preceding example, we have fused the treatment of unhandled messages together with how the actor handles regular messages. Stateful actors often change the way they handle regular messages, and it is essential to separate the treatment of unhandled messages from the normal behavior of the actor. We will study how to change the actor behavior in the next section.

Actor behavior and state

When an actor changes its state, it is often necessary to change the way it handles incoming messages. The way that the actor handles regular messages is called the **behavior** of the actor. In this section, we will study how to manipulate actor behavior.

We have previously learned that we define the initial behavior of the actor by implementing the `receive` method. Note that the `receive` method must always return the same partial function. It is not correct to return different partial functions from `receive` depending on the current state of the actor. Let's assume we want to define a `CountdownActor` actor class, which decreases its `n` integer field every time it receives a `count` message, until it reaches zero. After the `CountdownActor` class reaches zero, it should ignore all subsequent messages. The following definition of the `receive` method is not allowed in Akka:

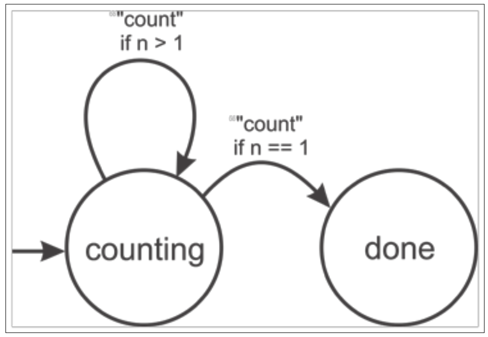
```
class CountdownActor extends Actor {
  var n = 10
  def receive = if (n > 0) { // never do this
    case "count" =>
      log(s"n = $n")
      n -= 1
  } else PartialFunction.empty
}
```

To correctly change the behavior of the `CountdownActor` class after it reaches zero, we use the `become` method on the actor's `context` object. In the correct definition of the `CountdownActor` class, we define two methods, `counting` and `done`, which return two different behaviors. The `counting` behavior reacts to the `count` messages and calls `become` to change to the `done` behavior once the `n` field is zero. The `done` behavior is just an empty partial function, which ignores all the messages. This is shown in the following implementation of the `CountdownActor` class:

```
class CountdownActor extends Actor {
  val log = Logging(context.system, this)
  var n = 10
  def counting: Actor.Receive = {
    case "count" =>
      n -= 1
      log.info(s"n = $n")
      if (n == 0) context.become(done)
  }
  def done = PartialFunction.empty
  def receive = counting
}
```

The `receive` method defines the initial behavior of the actor, which must be the `counting` behavior. Note that we are using the type alias `Receive` from the `Actor` companion object, which is just a shorthand for the `PartialFunction[Any, Unit]` type.

When modeling complex actors, it is helpful to think of them as **state machines**. A state machine is a mathematical model that represents a system with some number of states and transitions between these states. In an actor, each behavior corresponds to a state in the state machine. A transition exists between two states if the actor potentially calls the `become` method, when receiving a certain message. In the following figure, we illustrate the state machine corresponding to the `CountdownActor` class. The two circles represent the states corresponding to the behaviors `counting` and `done`. The initial behavior is `counting`, so we draw an arrow pointing to the corresponding state. We represent the transitions between the states with arrows starting and ending at a state. When the actor receives the `count` message and the `n` field is larger than `1`, the behavior does not change. However, when the actor receives the `count` message and the `n` field is decreased to `0`, the actor changes its behavior to `done`.



The following short program tests the correctness of our actor. We use the actor system to create a new `countdown` actor, and send it 20 `count` messages. The actor only reacts to the first 10 messages, before switching to the `done` behavior:

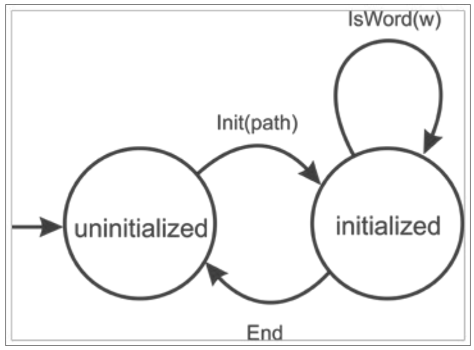
```
object ActorsCountdown extends App {
  val countdown = ourSystem.actorOf(Props[CountdownActor])
  for (i <- 0 until 20) countdown ! "count"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Whenever an actor responds to the incoming messages differently depending on its current state, you should decompose different states into partial functions and use the `become` method to switch between states. This is particularly important when actors get more complex, and ensures that the actor logic is easier to understand and maintain.

> When a stateful actor needs to change its behavior, declare a separate partial function for each of its behaviors. Implement the `receive` method to return the method corresponding to the initial behavior.

We now consider a more refined example, in which we define an actor that checks if a given word exists in a dictionary and prints it to the standard output. We want to be able to change the dictionary that the actor is using during runtime. To set the dictionary, we send the actor an `Init` message with the path to the dictionary. After that, we can check if a word is in the dictionary by sending the actor the `IsWord` message. Once we're done using the dictionary, we can ask the actor to unload the dictionary by sending it the `End` message. After that, we can initialize the actor with some other dictionary.

The following state machine models this logic with two behaviors called `uninitialized` and `initialized`:



It is a recommended practice to define the datatypes for the different messages in the companion object of the actor class. In this case, we add the case classes `Init`, `IsWord`, and `End` to the companion object of the `DictionaryActor` class:

```
object DictionaryActor {
  case class Init(path: String)
  case class IsWord(w: String)
  case object End
}
```

We next define the `DictionaryActor` actor class. This class defines a private `Logging` object `log`, and a `dictionary` mutable set, which is initially empty and can be used to store words. The `receive` method returns the `uninitialized` behavior, which only accepts the `Init` message type. When an `Init` message arrives, the actor uses its `path` field to fetch the dictionary from a file, load the words, and call `become` to switch to the `initialized` behavior. When an `IsWord` message arrives, the actor checks if the word exists and prints it to the standard output. If an `End` message arrives, the actor clears the dictionary and switches back to the `uninitialized` behavior. This is shown in the following code snippet:

```
class DictionaryActor extends Actor {
  private val log = Logging(context.system, this)
  private val dictionary = mutable.Set[String]()
  def receive = uninitialized
  def uninitialized: PartialFunction[Any, Unit] = {
    case DictionaryActor.Init(path) =>
      val stream = getClass.getResourceAsStream(path)
      val words = Source.fromInputStream(stream)
      for (w <- words.getLines) dictionary += w
      context.become(initialized)
```

```
  }
  def initialized: PartialFunction[Any, Unit] = {
    case DictionaryActor.IsWord(w) =>
      log.info(s"word '$w' exists: ${dictionary(w)}")
    case DictionaryActor.End =>
      dictionary.clear()
      context.become(uninitialized)
  }
  override def unhandled(msg: Any) = {
    log.info(s"cannot handle message $msg in this state.")
  }
}
```

Note that we have overridden the `unhandled` method in the `DictionaryActor` class. In this case, using the `unhandled` method reduces code duplication, and makes the `DictionaryActor` class easier to maintain, as there is no need to list the default case twice in both the `initialized` and `uninitialized` behaviors.

If you are using a Unix system, you can load the list of words, separated by a newline character, from the file in the location `/usr/share/dict/words`. Alternatively, download the source code for this book and find the `words.txt` file, or create a dummy file with several words, and save it to the `src/main/resources/org/learningconcurrency/` directory. You can then test the correctness of the `DictionaryActor` class, using the following program:

```
val dict = ourSystem.actorOf(Props[DictionaryActor], "dictionary")
dict ! DictionaryActor.IsWord("program")
Thread.sleep(1000)
dict ! DictionaryActor.Init("/org/learningconcurrency/words.txt")
Thread.sleep(1000)
```

The first message sent to the actor results in an error message. We cannot send an `IsWord` message before initializing the actor. After sending the `Init` message, we can check if words are present in the dictionary. Finally, we send an `End` message and shut down the actor system, as shown in the following code snippet:

```
dict ! DictionaryActor.IsWord("program")
Thread.sleep(1000)
dict ! DictionaryActor.IsWord("balaban")
Thread.sleep(1000)
dict ! DictionaryActor.End
Thread.sleep(1000)
ourSystem.shutdown()
```

Having learned about actor behaviors, we will study how actors are organized into a hierarchy in the next section.

Akka actor hierarchy

In large organizations, people are assigned roles and responsibilities for different tasks in order to reach a specific goal. The CEO of the company chooses a specific goal, such as launching a software product. He then delegates parts of the work tasks to various teams within the company: the marketing team investigates who are the potential customers for the new product, the design team develops the user interface of the product, and the software engineering team implements the logic of the software product. Each of these teams can be further decomposed into subteams with different roles and responsibilities, depending on the size of the company. For example, the software engineering team can be composed into two developer subteams, responsible for implementing the backend of the software product, such as the server-side code, and the frontend, such as the website or a desktop UI.

Similarly, sets of actors can form hierarchies in which actors closer to the root work on more general tasks, and delegate work items to more specialized actors lower in the hierarchy. Organizing parts of the system in hierarchies is a natural and systematic way to decompose a complex program into its basic components. In the context of actors, a correctly chosen actor hierarchy can also guarantee better scalability of the application, depending on how the work is balanced between the actors. Importantly, a hierarchy between actors allows isolating and replacing parts of the system that fail more easily.

In Akka, actors implicitly form a hierarchy. Every actor can have some number of child actors, and it can create or stop child actors using the `context` object. To test this relationship, we will define two actor classes to represent the parent and child actors. We start by defining the `ChildActor` actor class, which reacts to the `sayhi` messages by printing the reference to its parent actor. The reference to the parent is obtained by calling the `parent` method on the `context` object. Additionally, we will override the `postStop` method of the `Actor` class, which is invoked after the actor stops. By doing this, we will be able to see precisely when a child actor is stopped. The `ChildActor` template is shown in the following code snippet:

```
class ChildActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "sayhi" =>
      val parent = context.parent
      log.info(s"my parent $parent made me say hi!")
  }
  override def postStop() {
    log.info("child stopped!")
  }
}
```

We now define an actor class called `ParentActor`, which can accept the messages `create`, `sayhi`, and `stop`. When `ParentActor` receives a `create` message, it creates a new child by calling `actorOf` on the `context` object. When the `ParentActor` class receives a `sayhi` message, it forwards the message to its children by traversing the `context.children` list, and resending the message to each child. Finally, when the `ParentActor` class receives a `stop` message, it stops itself:

```
class ParentActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "create" =>
      context.actorOf(Props[ChildActor])
      log.info(s"created a kid; children = ${context.children}")
    case "sayhi" =>
      log.info("Kids, say hi!")
      for (c <- context.children) c ! "sayhi"
    case "stop" =>
      log.info("parent stopping")
      context.stop(self)
  }
}
```

We test the actor classes `ParentActor` and `ChildActor` in the following program. We first create the `ParentActor` instance `parent`, and then send two `create` messages to `parent`. The `parent` actor prints that it had created a child actor twice. We then send a `sayhi` message to `parent`, and witness how the child actors output a message after the parent forwards `sayhi` to them. Finally, we send a `stop` message to stop the `parent` actor. This is shown in the following program:

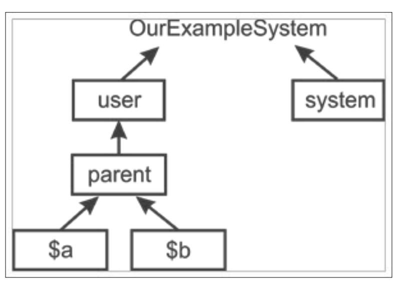
```
object ActorsHierarchy extends App {
  val parent = ourSystem.actorOf(Props[ParentActor], "parent")
  parent ! "create"
  parent ! "create"
  Thread.sleep(1000)
  parent ! "sayhi"
  Thread.sleep(1000)
  parent ! "stop"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

By studying the standard output, we find that each of the two child actors output a `sayhi` message immediately after the `parent` actor prints that it is about to stop. This is the normal behavior of Akka actors: a child actor cannot exist without its parent. As soon as the parent actor stops, its child actors are stopped by the actor system as well.

> When an actor is stopped, its child actors are also automatically stopped.

If you ran the preceding example program, you might have noticed that printing an actor reference reflects the actor's position in the actor hierarchy. For example, printing the child actor reference shows the `akka://OurExampleSystem/user/parent/$a` string. The first part of this string, `akka://`, denotes that this reference points to a local actor. The `OurExampleSystem` part is the name of the actor system that we are using in this example. The `parent/$a` part reflects the name of the parent actor and the automatically generated name `$a` of the child actor. Unexpectedly, the string representation of the actor reference also contains a reference to an intermediate actor called `user`.

In Akka, an actor that resides at the top of the actor hierarchy is called the **guardian actor**, which exists to perform various internal tasks, such as logging and restarting user actors. Every top-level actor created in the application is placed under the `user` predefined guardian actor. There are other guardian actors. For example, actors internally used by the actor system are placed under the `system` guardian actor. The actor hierarchy is graphically shown in the following figure, where the guardian actors `user` and `system` form two separate hierarchies in the actor system called `OurExampleSystem`:



In this section, we saw that Akka actors form a hierarchy, and learned about the relationships between actors in this hierarchy. Importantly, we learned how to refer to immediate neighbors of an actor using the `parent` and `children` methods of the `context` object. In the next section, we will see how to refer to an arbitrary actor within the same actor system.

Identifying actors

In the previous section, we learned that actors are organized in a hierarchical tree, in which every actor has a parent and some number of children. Thus, every actor lies on a unique path from the root of this hierarchy, and can be assigned a unique sequence of actor names on this path. The `parent` actor was directly beneath the `user` guardian actor, so its unique sequence of actor names is `/user/parent`. Similarly, the unique sequence of actor names for the `parent` actor's child actor `$a` is `/user/parent/$a`. An **actor path** is a concatenation of the protocol, the actor system name, and the actor names on the path from the top guardian actor to a specific actor. The actor path of the `parent` actor from the previous example is `akka://OurExampleSystem/user/parent`.

Actor paths closely correspond to file paths in a filesystem. Every file path uniquely designates a file location, just as an actor path uniquely designates the location of the actor in the hierarchy. Just as a file path in a filesystem does not mean that a file exists, an actor path does not imply that there is an actor on that file path in the actor system. Instead, an actor path is an identifier used to obtain an actor reference if one exists. Also, parts of the names in the actor path can be replaced with wildcards and the `..` symbol, similar to how parts of filenames can be replaced in a shell. In this case, we obtain a **path selection**. For example, the path selection `..` references the parent of the current actor. The selection `../*` references the current actor and all its siblings.

Actor paths are different from actor references; we cannot send a message to an actor using its actor path. Instead, we must first use the actor path to identify an actor on that actor path. If we successfully find an actor reference behind an actor path, we can send messages to it.

To obtain an actor reference corresponding to an actor path, we call the `actorSelection` method on the context object of an actor. This method takes an actor path, or a path selection. Calling the `actorSelection` method might address zero actors if no actors correspond to the actor path. Similarly, it might address multiple actors if we use a path selection. Thus, instead of returning an `ActorRef` object, the `actorSelection` method returns an `ActorSelection` object, which might represent zero, one, or more actors. We can use the `ActorSelection` object to send messages to these actors.

> Use the `actorSelection` method on the `context` object to communicate with arbitrary actors in the actor system.

If we compare the `ActorRef` object to a specific e-mail address, an `ActorSelection` object can be compared to a mailing list address. Sending an e-mail to a valid e-mail address ensures that the e-mail reaches a specific person. On the other hand, when we send an e-mail to a mailing list, the e-mail might reach zero, one, or more people, depending on the number of mailing list subscribers.

An `ActorSelection` object does not tell us anything about the concrete paths of the actors, in a similar way to how a mailing list does not tell us anything about its subscribers. For this purpose, Akka defines a special type of message called `Identify`. When an Akka actor receives an `Identify` message, it will automatically reply by sending back an `ActorIdentity` message with its `ActorRef` object. If there are no actors in the actor selection, the `ActorIdentity` message is sent back to the sender of `Identify` without an `ActorRef` object.

> Send `Identify` messages to the `ActorSelection` objects to obtain actor references of arbitrary actors in the actor system.

In the following example, we define a `CheckActor` actor class, which describes actors that check and print actor references whenever they receive a message with an actor path. When the actor of type `CheckActor` receives a string with an actor path or a path selection, it obtains an `ActorSelection` object and sends it an `Identify` message. This message is forwarded to all actors in the selection, which then respond with an `ActorIdentity` message. The `Identify` message also takes a `messageId` argument. If an actor sends out multiple `Identify` messages, the `messageId` argument allows disambiguating between the different `ActorIdentity` responses. In our example, we use the `path` string as the `messageId` argument. When `CheckActor` receives an `ActorIdentity` message, it either prints the actor reference or reports that there is no actor on the specified path. The `CheckActor` class is shown in the following code snippet:

```
class CheckActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case path: String =>
      log.info(s"checking path $path")
      context.actorSelection(path) ! Identify(path)
    case ActorIdentity(path, Some(ref)) =>
      log.info(s"found actor $ref at $path")
    case ActorIdentity(path, None) =>
      log.info(s"could not find an actor at $path")
  }
}
```

Next, we instantiate a `checker` actor of the `CheckActor` class, and send it the path selection `../*`. This references all the child actors of the `checker` parent: the `checker` actor itself and its siblings:

```
val checker = ourSystem.actorOf(Props[CheckActor], "checker")
checker ! "../*"
```

We did not instantiate any top-level actors besides `checker`, so checker receives only a single `ActorIdentity` message and prints its own actor path. Next, we try to identify all the actors one level above `checker`. Recall the earlier figure. Since `checker` is a top-level actor, this should identify the guardian actors in the actor system:

```
checker ! "../../*"
```

As expected, `checker` prints the actor paths of the `user` and `system` guardian actors. We are curious to learn more about the system-internal actors from the `system` guardian actor. This time, we send an absolute path selection to `checker`:

```
checker ! "/system/*"
```

The `checker` actor prints the actor paths of the internal actors `log1-Logging` and `deadLetterListener`, which are used for logging and for processing unhandled messages, respectively. We next try identifying a non-existing actor:

```
checker ! "/user/checker2"
```

There are no actors named `checker2`, so `checker` receives an `ActorIdentity` message with the `ref` field set to `None` and prints that it cannot find an actor on that path.

Using the `actorSelection` method and the `Identify` message is the fundamental method for discovering unknown actors in the same actor system. Note that we will always obtain an actor reference, and never obtain a pointer to the actor object directly. To better understand the reasons for this, we will study the life cycle of actors in the next section.

The actor life cycle

Recall that the `ChildActor` class from a previous section overrode the `postStop` method to produce some logging output when the actor is stopped. In this section, we investigate when exactly `postStop` gets called, along with the other important events that comprise the life cycle of the actor.

To understand why the actor life cycle is important, we consider what happens if an actor throws an exception while processing an incoming message. In Akka, such an exception is considered abnormal behavior, so top-level user actors that throw an exception are by default restarted. Restarting creates a fresh actor object, and effectively means that the actor state is reinitialized. When an actor is restarted, its actor reference and actor path remain the same. Thus, the same `ActorRef` object might refer to many different physical actor objects during the logical existence of the same actor. This is one of the reasons why an actor must never allow its `this` reference to leak. Doing so allows other parts of the program to refer to an old actor object, consequently invalidating the transparency of the actor reference. Additionally, revealing the `this` reference of the actor can reveal the internals of the actor implementation, or even cause data corruption.

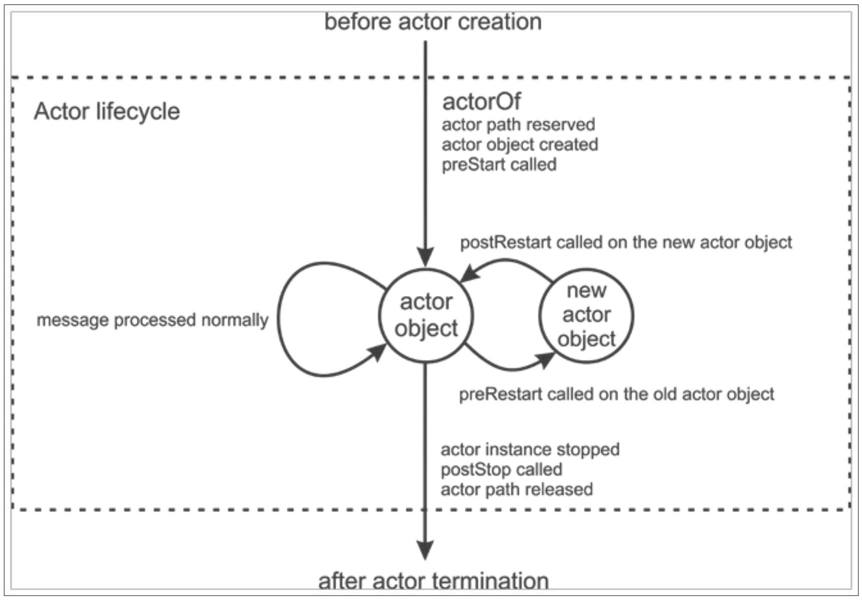
> Never pass an actor's `this` reference to other actors, as it breaks actor encapsulation.

Let's examine the complete actor life cycle. As we have learned, a logical actor instance is created when we call `actorOf`. The `Props` object is used to instantiate a physical actor object. This object is assigned a mailbox, and can start receiving input messages. The `actorOf` method returns an actor reference to the caller, and the actors can execute concurrently. Before the actor starts processing messages, its `preStart` method is called. The `preStart` method is used to initialize the logical actor instance.

After creation, the actor starts processing messages. At some point, an actor might need to be restarted due to an exception. When this happens, the `preRestart` method is first called. All the child actors are then stopped. Then, the `Props` object, previously used in order to create the actor with `actorOf`, is reused to create a new actor object. The `postRestart` method is called on the newly created actor object. After `postRestart` returns, the new actor object is assigned the same mailbox as the old actor object, and it continues to process messages that were in the mailbox before the restart.

By default, the `postRestart` method calls `preStart`. In some cases, we want to override this behavior. For example, a database connection might need to be opened only once during `preStart`, and closed when the logical actor instance is terminated.

Once the logical actor instance needs to stop, the `postStop` method gets called. The actor path associated with the actor is released, and returned to the actor system. By default, the `preRestart` method calls `postStop`. The complete actor life cycle is illustrated in the following figure:



Note that, during the actor life cycle, the rest of the actor system observes the same actor reference, regardless of how many times the actor restarts. Actor failures and restarts occur transparently for the rest of the system.

To experiment with the life cycle of an actor, we declare two actor classes `StringPrinter` and `LifecycleActor`. The `StringPrinter` actor prints a logging statement for each message that it receives. We override its `preStart` and `postStop` methods to precisely track when the actor has started and stopped, as shown in the following snippet:

```
class StringPrinter extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case msg => log.info(s"printer got message '$msg'")
  }
  override def preStart(): Unit = log.info(s"printer preStart.")
  override def postStop(): Unit = log.info(s"printer postStop.")
}
```

The `LifecycleActor` class maintains a `child` actor reference to a `StringPrinter` actor. The `LifecycleActor` class reacts to the `Double` and `Int` messages by printing them, and to the `List` messages by printing the first element of the list. When it receives a `String` message, the `LifecycleActor` instance forwards it to the `child` actor:

```
class LifecycleActor extends Actor {
  val log = Logging(context.system, this)
  var child: ActorRef = _
  def receive = {
    case num: Double  => log.info(s"got a double - $num")
    case num: Int     => log.info(s"got an integer - $num")
    case lst: List[_] => log.info(s"list - ${lst.head}, ...")
    case txt: String  => child ! txt
  }
}
```

We now override different life cycle hooks. We start with the `preStart` method to output a logging statement and instantiate the `child` actor. This ensures that the `child` reference is initialized before the actor starts processing any messages:

```
override def preStart(): Unit = {
  log.info("about to start")
  child = context.actorOf(Props[StringPrinter], "kiddo")
}
```

Next, we override the `preRestart` and `postRestart` methods. In `preRestart` and `postRestart`, we log the exception that caused the failure. The `postRestart` method calls `preStart` by default, so the new actor object gets initialized with a new `child` actor after a restart:

```
override def preRestart(t: Throwable, msg: Option[Any]): Unit = {
  log.info(s"about to restart because of $t, during message $msg")
  super.preRestart(t, msg)
}
override def postRestart(t: Throwable): Unit = {
  log.info(s"just restarted due to $t")
  super.postRestart(t)
}
```

Finally, we override `postStop` to track when the actor is stopped:

```
override def postStop() = log.info("just stopped")
```

We now create an instance of the `LifecycleActor` class called `testy`, and send a `math.Pi` message to it. The actor prints that it is about to start in its `preStart` method, and creates a `child` new actor. It then prints that it received the value `math.Pi`. Importantly, the `child about to start` logging statement is printed after the `math.Pi` message is received. This shows that actor creation is an asynchronous operation: when we call `actorOf`, creating the actor is delegated to the actor system, and the program immediately proceeds.

```
val testy = ourSystem.actorOf(Props[LifecycleActor], "testy")
testy ! math.Pi
```

We then send a `String` message to `testy`. The message is forwarded to the `child` actor, which prints a logging statement, indicating that it received the message:

```
testy ! "hi there!"
```

Finally, we send a `Nil` message to `testy`. The `Nil` object represents an empty list, so `testy` throws an exception when attempting to fetch the `head` element. It reports that it needs to restart. After that, we witness that the `child` actor prints the message that it needs to stop; recall that the child actors are stopped when an actor is restarted. Finally, `testy` prints that it is about to restart, and the new `child` actor is instantiated. These events are caused by the following statement:

```
testy ! Nil
```

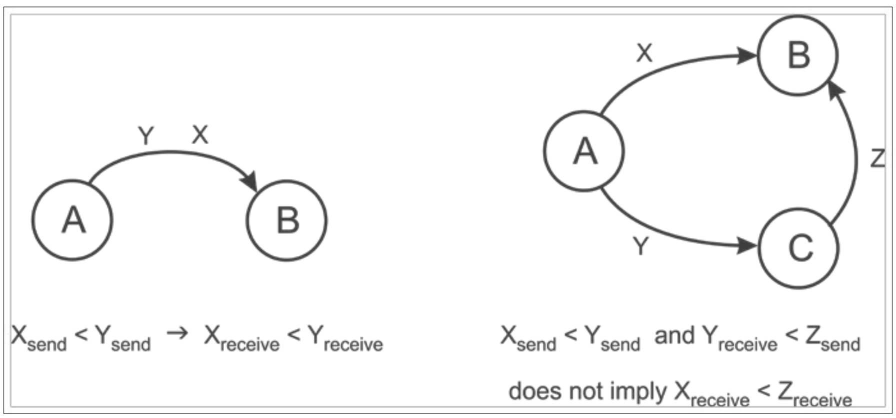
Testing the actor life cycle revealed an important property of the `actorOf` method. When we call `actorOf`, the execution proceeds without waiting for the actor to fully initialize itself. Similarly, sending a message does not block execution until the message is received or processed by another actor; we say that message sends are asynchronous. In the next section, we will examine various communication patterns that address this asynchronous behavior.

Communication between actors

We have learned that actors communicate by sending messages. While actors running on the same machine can access shared parts of memory in the presence of proper synchronization, sending messages allows isolating the actor from the rest of the system and ensures location transparency. The fundamental operation that allows you to send a message to an actor is the `!` operator. We have learned that the `!` operator is a non-blocking operation: sending a message does not block the execution of the sender until the message is delivered. This way of sending messages is sometimes called the **fire-and-forget** pattern, because it does not wait for a reply from the message receiver, nor does it ensure that the message is delivered.

Sending messages in this way improves the throughput of programs built using actors, but can be limiting in some situations. For example, we might want to send a message and wait for the response from the target. In this section, we learn about patterns used in actor communication that go beyond fire-and-forget.

While the fire-and-forget pattern does not guarantee that the message is delivered, it guarantees that the message is delivered **at most once**. The target actor never receives duplicate messages. Furthermore, the messages are guaranteed to be ordered for a given pair of sender and receiver actors. If an actor **A** sends messages **X** and **Y** in that order, the actor **B** will receive no messages, only the message **X**, only the message **Y**, or the message **X**, followed by the message **Y**. This is shown on the left in the following figure:



However, the delivery order is not ensured for a group of three or more actors. For example, as shown on the right in the preceding figure, actor **A** performs the following actions:

- Sends a message **X** to the actor **B**
- Sends a message **Y** to another actor **C**
- Actor **C** sends a message **Z** to the actor **B** after having received **Y**

In this situation, the delivery order between messages **X** and **Z** is not guaranteed. The actor **B** might receive the messages **X** and **Z** in any order. This property reflects the characteristics of most computer networks, and is adopted to allow actors to run transparently on network nodes that may be remote.

> The order in which an actor **B** receives messages from an actor **A** is the same as the order in which these messages are sent from the actor **A**.

Before we study various patterns of actor communication, we note that the `!` operator was not the only non-blocking operation. The methods `actorOf` and `actorSelection` are also non-blocking. These methods are often called while an actor is processing a message. Blocking the actor while the message is processed prevents the actor from processing subsequent messages in the mailbox and severely compromises the throughput of the system. For these reasons, most of the actor API is non-blocking. Additionally, we must never start blocking the operations from third-party libraries from within an actor.

> Messages must be handled without blocking indefinitely. Never start an infinite loop and avoid long-running computations in the `receive` block, the `unhandled` method, and within actor life cycle hooks.

The ask pattern

Not being able to block from within an actor prevents the request-respond communication pattern. In this pattern, an actor interested in certain information sends a request message to another actor. It then needs to wait for a response message from the other actor. In Akka, this communication pattern is also known as the **ask pattern**.

The `akka.pattern` package defines the use of convenience methods in actor communication. Importing its contents allows us to call the `?` operator (pronounced ask) on actor references. This operator sends a message to the target actor like the tell operator. Additionally, the ask operator returns a future object with the response from the target actor.

To illustrate the usage of the ask pattern, we will define two actors that play ping pong with each other. A `Pingy` actor will send a `ping` request message to another actor of type `Pongy`. When the `Pongy` actor receives the `ping` message, it sends a `pong` response message to the sender. We start by importing the `akka.pattern` package:

```
import akka.pattern._
```

We first define the `Pongy` actor class. To respond to the `ping` incoming message, `Pongy` needs an actor reference of the sender. While processing a message, every actor can call the `sender` method of the `Actor` class to obtain the actor reference of the sender of the current message. `Pongy` uses the `sender` method to send `ping` back to `Pingy`. The `Pongy` implementation is shown in the following code snippet:

```
class Pongy extends Actor {
  val log = Logging(context.system, this)
  def receive = {
```

```
      case "ping" =>
        log.info("Got a ping -- ponging back!")
        sender ! "pong"
        context.stop(self)
    }
    override def postStop() = log.info("pongy going down")
  }
```

Next, we define the `Pingy` actor class, which uses the ask operator to send a request to `Pongy`. When `Pingy` receives a `pongyRef` actor reference of `Pongy`, it creates an implicit `Timeout` object set to `2 seconds`. Using the ask operator requires an implicit `Timeout` object in scope; the future is failed with an `AskTimeoutException` if the response message does not arrive within the given timeframe. Once `Pingy` sends the `ping` message, it is left with an `f` future object. The `Pingy` actor uses the special `pipeTo` combinator that sends the value in the future to the sender of the `pongyRef` actor reference, as shown in the following code:

```
  import akka.util.Timeout
  import scala.concurrent.duration._
  class Pingy extends Actor {
    val log = Logging(context.system, this)
    def receive = {
      case pongyRef: ActorRef =>
        implicit val timeout = Timeout(2 seconds)
        val f = pongyRef ? "ping"
        f pipeTo sender
    }
  }
```

The message in the future object can be manipulated using the standard future combinators seen in *Chapter 4*, *Asynchronous Programming with Futures and Promises*. However, the following definition of `Pingy` would not be correct:

```
  class Pingy extends Actor {
    val log = Logging(context.system, this)
    def receive = {
      case pongyRef: ActorRef =>
        implicit val timeout = Timeout(2 seconds)
        val f = pongyRef ? "ping"
        f onComplete { case v => log.info(s"Response: $v") } // bad!
    }
  }
```

Although it is perfectly legal to call `onComplete` on the `f` future, the subsequent asynchronous computation should not access any mutable actor state. Recall that the actor state should be visible only to the actor, so concurrently accessing it opens the possibility of data races and race conditions. The `log` object should only be accessed by the actor that owns it. Similarly, we should not call the `sender` method from within the `onComplete` handler. By the time that the future is completed with the response message, the actor might be processing a different message with a different sender, so the `sender` method can return arbitrary values.

> When starting an asynchronous computation from within the `receive` block, the `unhandled` method, or a life cycle hook, never let the closure capture any mutable actor state.

To test `Pingy` and `Pongy` in action, we define the `Master` actor class that instantiates them. Upon receiving the `start` message, the `Master` actor passes the `pongy` reference to `pingy`. Once the `pingy` actor returns a `pong` message from `pongy`, the `Master` actor stops. This is shown in the following `Master` actor template:

```
class Master extends Actor {
  val pingy = ourSystem.actorOf(Props[Pingy], "pingy")
  val pongy = ourSystem.actorOf(Props[Pongy], "pongy")
  def receive = {
    case "start" =>
      pingy ! pongy
    case "pong" =>
      context.stop(self)
  }
  override def postStop() = log.info("master going down")
}
val masta = ourSystem.actorOf(Props[Master], "masta")
masta ! "start"
```

The ask pattern is useful because it allows you to send requests to multiple actors and obtain futures with their responses. Values from multiple futures can be combined within `for` comprehensions to compute a value from several responses. Using the fire-and-forget pattern when communicating with multiple actors requires changing the actor behavior, and is a lot more cumbersome than the ask pattern.

The forward pattern

Some actors exist solely to forward messages to other actors. For example, an actor might be responsible for load-balancing request messages between several worker actors, or it might forward the message to its mirror actor to ensure better availability. In such cases, it is useful to forward the message without changing the `sender` field of the message. The `forward` method on actor references serves this purpose.

In the following code, we use the `StringPrinter` actor from the previous section to define a `Router` actor class. A `Router` actor instantiates four child `StringPrinter` actors and maintains an `i` field with the index of the list child it forwarded the message to. Whenever it receives a message, it forwards the message to a different `StringPrinter` child before incrementing `i`:

```
class Router extends Actor {
  var i = 0
  val children = for (_ <- 0 until 4) yield
    context.actorOf(Props[StringPrinter])
  def receive = {
    case msg =>
      children(i) forward msg
      i = (i + 1) % 4
  }
}
```

In the following code, we create a `Router` actor and test it by sending it two messages. We can observe that the messages are printed to the standard output by two different `StringPrinter` actors, denoted with actors on the actor paths `/user/router/$b` and `/user/router/$a`:

```
val router = ourSystem.actorOf(Props[Router], "router")
router ! "Hola"
router ! "Hey!"
```

The forward pattern is typically used in router actors, which use specific knowledge to decide about the destination of the message; replicator actors, which send the message to multiple destinations; or load balancers, which ensure that the workload is spread evenly between a set of worker actors.

Stopping actors

So far, we have stopped different actors by making them call `context.stop`. Calling the `stop` method on the `context` object terminates the actor immediately after the current message is processed. In some cases, we want to have more control over how an actor gets terminated. For example, we might want to allow the actor to process its remaining messages or wait for the termination of some other actors. In Akka, there are several special message types that assist us in doing so, and we study them in this section.

In many cases, we do not want to terminate an actor instance, but simply restart it. We have previously learned that an actor is automatically restarted when it throws an exception. An actor is also restarted when it receives the `Kill` message: when we send a `Kill` message to an actor, the actor automatically throws an `ActorKilledException` and fails.

> Use the `Kill` message to restart the target actor without losing the messages in the mailbox.

Unlike the `stop` method, the `Kill` message does not terminate the actor, but only restarts it. In some cases, we want to terminate the actor instance, but allow it to process the messages from its mailbox. Sending a `PoisonPill` message to an actor has the same effect as calling `stop`, but allows the actor to process the messages that were in the mailbox before the `PoisonPill` message arrives.

> Use the `PoisonPill` message to stop the actor, but allow it to process the messages received before the `PoisonPill` message.

In some cases, allowing the actor to process its message using `PoisonPill` is not enough. An actor might have to wait for other actors to terminate before terminating itself. An orderly shutdown is important in some cases, as actors might be involved in sensitive operations, such as writing to a file on the disk. We do not want to forcefully stop them when we end the application. A facility that allows an actor to track the termination of other actors is called **DeathWatch** in Akka.

Recall the earlier example with the actors `Pingy` and `Pongy`. Let's say that we want to terminate `Pingy`, but only after `Pongy` has already been terminated. We define a new `GracefulPingy` actor class for this purpose. `GracefulPingy` calls the `watch` method on the `context` object when it gets created. This ensures that, after `Pongy` terminates and its `postStop` method completes, `GracefulPingy` receives a `Terminated` message with the actor reference to `Pongy`. Upon receiving the `Terminated` message, `GracefulPingy` stops itself, as shown in the following `GracefulPingy` implementation:

```
class GracefulPingy extends Actor {
  val pongy = context.actorOf(Props[Pongy], "pongy")
  context.watch(pongy)
  def receive = {
    case "Die, Pingy!" =>
      context.stop(pongy)
    case Terminated(`pongy`) =>
      context.stop(self)
  }
}
```

Whenever we want to track the termination of an actor from inside an actor, we use DeathWatch, as in the previous example. When we need to wait for the termination of an actor from outside an actor, we use the **graceful stop pattern**. The `gracefulStop` method from the `akka.pattern` package takes an actor reference, a timeout, and a shutdown message. It returns a future and asynchronously sends the shutdown message to the actor. If the actor terminates within the allotted timeout, the future is successfully completed. Otherwise, the future fails. In the following code, we create a `GracefulPingy` actor and call the `gracefulStop` method:

```
object CommunicatingGracefulStop extends App {
  val grace = ourSystem.actorOf(Props[GracefulPingy], "grace")
  val stopped =
    gracefulStop(grace, 3.seconds, "Die, Pingy!")
  stopped onComplete {
    case Success(x) =>
      log("graceful shutdown successful")
      ourSystem.shutdown()
    case Failure(t) =>
      log("grace not stopped!")
      ourSystem.shutdown()
  }
}
```

We typically use DeathWatch inside the actors, and the graceful stop pattern in the main application thread. The graceful stop pattern can be used within actors as well, as long as we are careful that the callbacks on the future returned by `gracefulStop` do not capture actor state. Together, DeathWatch and the graceful stop pattern allow safely shutting down actor-based programs.

Actor supervision

When studying the actor life cycle, we said that top-level user actors are by default restarted when an exception occurs. We now take a closer inspection at how this works. In Akka, every actor acts as a supervisor for its children. When a child fails, it suspends the processing messages, and sends a message to its parent to decide what to do about the failure. The policy that decides what happens to the parent and the child after the child fails is called the **supervision strategy**. The parent might decide to do the following:

- Restart the actor, indicated with the `Restart` message
- Resume the actor without a restart, indicated with the `Resume` message
- Permanently stop the actor, indicated with the `Stop` message
- Fail itself with the same exception, indicated with the `Escalate` message

By default, the `user` guardian actor comes with a supervision strategy that restarts the failed children access. User actors stop their children by default. Both supervision strategies can be overridden.

To override the default supervision strategy in user actors, we override the `supervisorStrategy` field of the `Actor` class. In the following code, we define a particularly troublesome actor class called `Naughty`. When the `Naughty` class receives a `String` message, it prints a logging statement. For all other message types, it throws a `RuntimeException`, as shown in the following implementation:

```
class Naughty extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case s: String => log.info(s)
    case msg => throw new RuntimeException
  }
  override def postRestart(t: Throwable) =
    log.info("naughty restarted")
}
```

Next, we declare a `Supervisor` actor class, which creates a child actor of the type `Naughty`. The `Supervisor` actor does not handle any messages, but overrides the default supervision strategy. If a `Supervisor` actor's child actor fails because of throwing an `ActorKilledException`, it is restarted. However, if its child actor fails with any other exception type, the exception is escalated to the `Supervisor` actor. We override the `supervisorStrategy` field with the value `OneForOneStrategy`, a supervision strategy that applies fault handling specifically to the actor that failed:

```
class Supervisor extends Actor {
  val child = context.actorOf(Props[StringPrinter], "naughty")
  def receive = PartialFunction.empty
  override val supervisorStrategy =
    OneForOneStrategy() {
      case ake: ActorKilledException => Restart
      case _ => Escalate
    }
}
```

We test the new supervisor strategy by creating an actor instance `super` of the `Supervisor` actor class. We then create an actor selection for all the children of `super`, and send them a `Kill` message. This fails the `Naughty` actor, but `super` restarts it due to its supervision strategy. We then apologize to the `Naughty` actor by sending it a `String` message. Finally, we convert a `String` message to a list of characters, and send it to the `Naughty` actor, which then throws a `RuntimeException`. This exception is escalated by `super`, and both actors are terminated, as shown in the following code snippet:

```
ourSystem.actorOf(Props[Supervisor], "super")
ourSystem.actorSelection("/user/super/*") ! Kill
ourSystem.actorSelection("/user/super/*") ! "sorry about that"
ourSystem.actorSelection("/user/super/*") ! "kaboom".toList
```

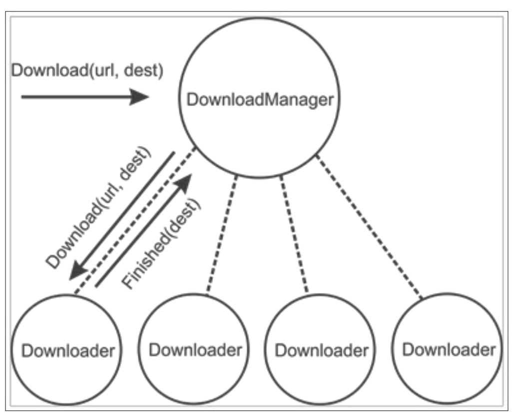
In this example, we saw how `OneForOneStrategy` works. When an actor fails, that specific actor is resumed, restarted, or stopped, depending on the exception that caused it to fail. The alternative `AllForOneStrategy` applies the fault-handling decision to all the children. When one of the child actors stops, all the other children are resumed, restarted, or stopped.

Recall our minimalistic web browser implementation from *Chapter 6, Concurrent Programming with Reactive Extensions*. A more advanced web browser requires a separate subsystem that handles concurrent file downloads. Usually, we refer to such a software component as a download manager. We now consider a larger example, in which we apply our knowledge of actors in order to implement the infrastructure for a simple download manager.

The download manager will be implemented as an actor, represented by the `DownloadManager` actor class. The two most important tasks of every download manager are to download the resources at the requested URL, and to track the downloads that are currently in progress. To be able to react to download requests and download completion events, we define the message types `Download` and `Finished` in the `DownloadManager` companion object. The `Download` message encapsulates the URL of the resource and the destination file for the resource, while the `Finished` message encodes the destination file where the resource is saved:

```
object DownloadManager {
  case class Download(url: String, dest: String)
  case class Finished(dest: String)
}
```

The `DownloadManager` actor will not execute the downloads itself. Doing so would prevent it from receiving any messages before the download completes. Furthermore, this will serialize different downloads and prevent them from executing concurrently. Thus, the `DownloadManager` actor must delegate the task of downloading the files to different actors. We represent these actors with the `Downloader` actor class. A `DownloadManager` actor maintains a set of `Downloader` children, and tracks which children are currently downloading a resource. When a `DownloadManager` actor receives a `Download` message, it picks one of the non-busy `Downloader` actors, and forwards the `Download` message to it. Once the download is complete, the `Downloader` actor sends a `Finished` message to its parent. This is illustrated in the following figure:



We first show the implementation of the `Downloader` actor class. When a `Downloader` actor receives a `Download` message, it downloads the contents of the specified URL, and writes them to a destination file. It then sends the `Finished` message back to the sender of the `Download` message, as shown in the following implementation:

```
class Downloader extends Actor {
  def receive = {
    case DownloadManager.Download(url, dest) =>
      val content = Source.fromURL(url)
      FileUtils.write(new java.io.File(dest), content.mkString)
      sender ! DownloadManager.Finished(dest)
  }
}
```

The `DownloadManager` actor class needs to maintain state to track which of its `Downloader` actors is currently downloading a resource. If there are more download requests than there are available `Downloader` instances, the `DownloadManager` actor needs to enqueue the download requests until some `Downloader` actor becomes available. The `DownloadManager` actor maintains a `downloaders` queue with actor references to non-busy `Downloader` actors. It maintains another queue `pendingWork` with `Download` requests that cannot be assigned to any `Downloader` instances. Finally, it maintains a map called `workItems` that associates actor references of the busy `Downloader` instances with their `Download` requests. This is shown in the following `DownloadManager` implementation:

```
class DownloadManager(val downloadSlots: Int) extends Actor {
  import DownloadManager._
  val log = Logging(context.system, this)
  val downloaders = mutable.Queue[ActorRef]()
  val pendingWork = mutable.Queue[Download]()
  val workItems = mutable.Map[ActorRef, Download]()
  private def checkDownloads(): Unit = {
    if (pendingWork.nonEmpty && downloaders.nonEmpty) {
      val dl = downloaders.dequeue()
      val item = pendingWork.dequeue()
      log.info(
        s"$item starts, ${downloaders.size} download slots left")
      dl ! item
      workItems(dl) = item
    }
  }
  def receive = {
    case msg @ DownloadManager.Download(url, dest) =>
      pendingWork.enqueue(msg)
      checkDownloads()
    case DownloadManager.Finished(dest) =>
```

```
        workItems.remove(sender)
        downloaders.enqueue(sender)
        log.info(
          s"'$dest' done, ${downloaders.size} download slots left")
        checkDownloads()
    }
  }
```

The `checkDownloads` private method maintains the `DownloadManager` actor's invariant: the `pendingWork` and the `downloaders` queue cannot be non-empty at the same time. As soon as both the queues become non-empty, a `Downloader` actor reference `dl` is dequeued from `downloaders` and a `Download` request item is dequeued from `pendingWork`. The `item` is then sent as a message to the `dl` actor, and the `workItems` map is updated.

Whenever the `DownloadManager` actor receives a `Download` message, it adds it to `pendingWork` and calls `checkDownloads`. Similarly, when a `Finished` message arrives, the `Downloader` actor is removed from `workItems` and enqueued on the `downloaders` list.

To ensure that the `DownloadManager` actor is created with the specified number of `Downloader` child actors, we override the `preStart` method to create the `Downloaders` and add their actor references to the `downloaders` queue:

```
  override def preStart(): Unit = {
    for (i <- 0 until downloadSlots) {
      val dl = context.actorOf(Props[Downloader], s"dl$i")
      downloaders.enqueue()
    }
  }
```

Finally, we must override the `supervisorStrategy` field of the `DownloadManager` actor. We use the `OneForOneStrategy` field again, but specify that the actor can be restarted or resumed only up to 20 times within a 2-second interval.

We expect that some URLs might be invalid; in which case, the actor fails with a `FileNotFoundException`. We need to remove such an actor from the `workItems` collection and add it back to the `downloaders` queue. It does not make sense to restart the `Downloader` actors, because they do not contain any state. Instead of restarting, we simply resume a `Downloader` actor that cannot resolve a URL. If the `Downloader` instances fail due to any other messages, we escalate the exception and fail the `DownloadManager` actor, as shown in the following `supervisorStrategy` implementation:

```
  override val supervisorStrategy =
    OneForOneStrategy(
      maxNrOfRetries = 20, withinTimeRange = 2 seconds
```

```
) {
  case fnf: java.io.FileNotFoundException =>
    log.info(s"Resource could not be found: $fnf")
    workItems.remove(sender)
    downloaders.enqueue(sender)
    Resume // ignores the exception and resumes the actor
  case _ =>
    Escalate
}
```

To test the download manager, we create a `DownloadManager` actor with four download slots, and send it several `Download` messages:

```
val downloadManager =
  ourSystem.actorOf(Props(classOf[DownloadManager], 4), "man")
downloadManager ! Download(
  "http://www.w3.org/Addressing/URL/url-spec.txt",
  "url-spec.txt")
```

An extra copy of the URL specification cannot hurt, so we download it to our computer. The download manager logs that there are only three download slots left. Once the download completes, the download manager logs that there are four remaining download slots again. We then decide that we would like to contribute to the Scala programming language, so we download the `README` file from the official Scala repository. Unfortunately, we enter an invalid URL, and observe a warning from the download manager, saying that the resource cannot be found:

```
downloadManager ! Download(
  "https://github.com/scala/scala/blob/master/README.md",
  "README.md")
```

The simple implementation of the basic actor-based download manager illustrates both how to achieve concurrency by delegating work to child actors, and how to treat failures in child actors. Delegating work is important both for decomposing the program into smaller, isolated components, and to achieve better throughput and scalability. Actor supervision is the fundamental mechanism for handling failures in isolated components that is implemented in separate actors.

Remote actors

So far in this book, we have mostly concentrated on writing programs on a single computer. Concurrent programs are executed within a single process on one computer, and they communicate using shared memory. Seemingly, actors described in this chapter communicate by passing messages. However, the message passing used throughout this chapter is implemented by reading and writing to shared memory under the hood.

In this section, we study how the actor model ensures location transparency by taking existing actors and deploying them in a distributed program. We take two existing actor implementations, namely, `Pingy` and `Pongy`, and deploy them inside different processes. We will then instruct `Pingy` to send a message to `Pongy`, as before, and wait until `Pingy` returns the `Pongy` actor's message. The message exchange will occur transparently, although `Pingy` and `Pongy` were previously implemented without knowing that they might exist inside separate processes, or even different computers.

The Akka actor framework is organized into several modules. To use the part of Akka that allows communicating with actors in remote actor systems, we need to add the following dependency to our build definition file:

```
libraryDependencies +=
  "com.typesafe.akka" %% "akka-remote" % "2.3.2"
```

Before creating our ping-pong actors inside two different processes, we need to create an actor system that is capable of communicating with remote actors. To do this, we create a custom actor system configuration string. The actor system configuration string can be used to configure a range of different actor system properties; we are interested in using a custom `ActorRef` factory object called `RemoteActorRefProvider`. This `ActorRef` factory object allows the actor system to create actor references that can be used to communicate over the network. Furthermore, we configure the actor system to use the Netty networking library with the TCP network layer and the desired TCP port number. We declare the `remotingConfig` method for this task:

```
import com.typesafe.config._
def remotingConfig(port: Int) = ConfigFactory.parseString(s"""
akka {
  actor.provider = "akka.remote.RemoteActorRefProvider"
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = $port
    }
  }
}
""")
```

We then define a `remotingSystem` factory method that creates an actor system object using the given name and port. We use the `remotingConfig` method, defined earlier, to produce the configuration object for the specified network port:

```
def remotingSystem(name: String, port: Int): ActorSystem =
  ActorSystem(name, remotingConfig(port))
```

Now, we are ready to create the `Pongy` actor system. We declare an application called `RemotingPongySystem`, which instantiates an actor system called `PongyDimension` using the network port `24321`. We arbitrarily picked a network port that was free on our machine. If the creation of the actor system fails because the port is not available, you can pick a different port in the range from `1024` to `65535`. Make sure that you don't have a firewall running, as it can block the network traffic for arbitrary applications.

The `RemotingPongySystem` application is shown in the following example:

```
object RemotingPongySystem extends App {
  val system = remotingSystem("PongyDimension", 24321)
  val pongy = system.actorOf(Props[Pongy], "pongy")
  Thread.sleep(15000)
  system.shutdown()
}
```

The `RemotingPongySystem` application creates a `Pongy` actor and shuts down after 15 seconds. After we start it, we will only have a short period of time to start another application running the `Pingy` actor. We will call this second application `RemotingPingySystem`. Before we implement it, we create another actor called `Runner`, which will instantiate `Pingy`, obtain the `Pongy` actor's reference, and give it to `Pingy`; recall that the Ping Pong game from the earlier section starts when `Pingy` obtains the `Pongy` actor's reference.

When the `Runner` actor receives a `start` message, it constructs the actor path for `Pongy`. We use the `akka.tcp` protocol and the name of the remote actor system, along with its IP address and port number. The `Runner` actor sends an `Identify` message to the actor selection in order to obtain the actor reference to the remote `Pongy` instance. The complete `Runner` implementation is shown in the following code snippet:

```
class Runner extends Actor {
  val log = Logging(context.system, this)
  val pingy = context.actorOf(Props[Pingy], "pingy")
  def receive = {
    case "start" =>
      val pongySys = "akka.tcp://PongyDimension@127.0.0.1:24321"
```

```
      val pongyPath = "/user/pongy"
      val url = pongySys + pongyPath
      val selection = context.actorSelection(url)
      selection ! Identify(0)
    case ActorIdentity(0, Some(ref)) =>
      pingy ! ref
    case ActorIdentity(0, None) =>
      log.info("Something's wrong - ain't no pongy anywhere!")
      context.stop(self)
    case "pong" =>
      log.info("got a pong from another dimension.")
      context.stop(self)
  }
}
```

Once the `Runner` actor sends the `Pongy` actor reference to `Pingy`, the game of remote ping pong can begin. To test it, we declare the `RemotingPingySystem` application, which starts the `Runner` actor and sends it a `start` message:

```
object RemotingPingySystem extends App {
  val system = remotingSystem("PingyDimension", 24567)
  val runner = system.actorOf(Props[Runner], "runner")
  runner ! "start"
  Thread.sleep(5000)
  system.shutdown()
}
```

We now need to start the `RemotingPongySystem` application, and the `RemotingPingySystem` application after that; we only have 15 seconds until the `RemotingPongySystem` application shuts itself down. The easiest way to do this is to start two SBT instances in your project folder, and run the two applications at the same time. After the `RemotingPingySystem` application starts, we soon observe a `pong` message from another dimension.

In the previous example, the actor system configuration and the `Runner` actor were responsible for setting up the network communication, and were not location-transparent. This is typically the case with distributed programs; a part of the program is responsible for initializing and discovering actors within remote actor systems, while the application-specific logic is confined within separate actors.

> Separate deployment logic from application logic in larger actor programs.

To summarize, remote actor communication requires the following steps:

- Declaring an actor system with an appropriate remoting configuration
- Starting two actor systems in separate processes or on separate machines
- Using actor path selection to obtain actor references
- Using actor references to transparently send messages

While the first three steps are not location-transparent, the application logic is usually confined within the fourth step, as we saw in this section. This is important, as it allows separating the deployment logic from the application semantics, and building distributed systems that can be deployed transparently to different network configurations.

Summary

In this chapter, we learned what actors are and how to use them to build concurrent programs. Using the Akka actor framework, we studied how to create actors, organize them into hierarchies, manage their life cycle, and recover them from errors. We examined important patterns in actor communication and learned how to model actor behavior. Finally, we saw how the actor model can ensure location transparency, and serve as a powerful tool to seamlessly build distributed systems.

Still, there are many Akka features that we omitted in this chapter. Akka comes with a detailed online documentation, which is one of the best sources of information on Akka. To obtain an in-depth understanding of distributed programming, we recommend the books *Distributed Algorithms*, *Nancy A. Lynch*, *Elsevier* and *Introduction to Reliable and Secure Distributed Programming*, *Christian Cachin, Rachid Guerraoui, Luis Rodrigues*, *Springer*.

In the next chapter, we will summarize the different concurrency libraries we learned about in this book, examine the typical use cases for each of them, and see how they work together in larger applications.