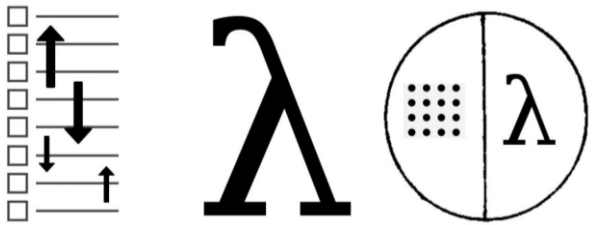
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”



Programming paradigms

L12: Scala Multithreading

by Michał Szczepanik

Scala and multithreading

Threads in Scala can be created by using two mechanisms:

- Extending the Thread class
- Extending the Runnable Interface

AnyRef class provides methods:

- final def synchronized[A] (e: => A): A
- final def wait(): Unit
- final def wait(milisec: Long): Unit
- final def wait(milisec: Long, nanosec: Int): Unit
- final def notify(): Unit
- final def notifyAll(): Unit

```
ob.synchronized {
        // this code can be accessed by only one task at a time
}
```

Thread creation by extending the Thread class

Create a class that extends the **Thread class**.

This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
class MyThread extends Thread  {
    override def run() {
        println("Thread " + Thread.currentThread().getName()
        +  " is running.")
    }
}

object GFG {
    def main(args: Array[String]) {
        for (x <- 1 to 5)   {
            var th = new MyThread()
            th.setName(x.toString())
            th.start()
        }
    }
}
```

Thread creation by Extending Runnable Interface

Create a new class which extends Runnable interface and override run() method. Then we instantiate a Thread object passing the created class to the constructor. We then call start() method on this object.

```
class MyThread extends Runnable  {
   override def run() {
      println("Thread " + Thread.currentThread().getName() +
                                  " is running.")
   }
}

object MainObject {
   def main(args: Array[String]) {
      for (x <- 1 to 5)  {
         var th = new Thread(new MyThread())
         th.setName(x.toString())
         th.start()
      }
   }
}
```

```
class Buffer {
  private var msg: String = _            // Message sent from producer to consumer.
  private var empty = true   // true if consumer should wait for producer to send message,
                                         // false if producer should wait for consumer to retrieve message.
  def put(msg: String) = this.synchronized {
    while (!empty) wait()      // Wait until message has been retrieved.
                                         //Never invoke the wait method outside of a loop!
    println(s"${Thread.currentThread.getName} puts $msg")
    this.msg = msg // Store message.
    empty = false //Toggle status
    notifyAll //Notify consumer that status has changed
  }
  def take: String = this.synchronized {
    while (empty) wait()       // Wait until message is available.
                                         // Never invoke the wait method outside of a loop!
    empty = true. // Toggle status
      notifyAll // Notify producer that status has changed
    println(s"${Thread.currentThread.getName} takes $msg")
    msg // return message
  }
}
```

```
class Producer(name: String, buf: Buffer) extends Thread(name) {
  override def run: Unit = {
    for (i <- 1 to 10) buf.put(s"m $i")
    buf.put("Done")
  }
}
class Consumer(name: String, buf: Buffer) extends Thread(name) {
  override def run: Unit = {
    var msg = ""
    do {
      msg = buf.take
    } while (msg != "Done")
  }
}
object prodCons {
  def main(args: Array[String]): Unit = {
    val buf = new Buffer
    new Producer("Producer", buf).start
    new Consumer("Consumer", buf).start
  }
}
```

Futures

Futures provide a way to reason about performing many operations in parallel – in an efficient and non-blocking way. A Future is a placeholder object for a value that may not yet exist.

Futures

```
val future = Future {
  val result = someLongRunningThing()
  result
}
```

Future values

A Future is an object holding a value which may become available at some point. This value is usually the result of some other computation:

- If the computation has not yet completed, we say that the Future is **not completed.**
- If the computation has completed with a value or with an exception, we say that the Future is **completed**.

Completion can take one of two forms:

- When a Future is completed with a value, we say that the future was **successfully completed** with that value.
- When a Future is completed with an exception thrown by the computation, we say that the Future was **failed** with that exception.

A Future has an important property that it may only be assigned once. Once a Future object is given a value or an exception, it becomes in effect immutable– it can never be overwritten.

Future completion state

Scala provides some callbacks that you can use at the end of a future (or chain of futures). You can perform different functions after both failed and successful executions. Again these operations also happen in a background thread.

```
val future = Future { getUser() }
```

```
Future
.onComplete {
  case Success(user) => println("yay!")
  case Failure(exception) => println("On no!")
}
```

Future transformations

Like many functional languages, Scala loves making things work like a list or collection of objects. You've probably seen this before with the Option class, which behaves mostly like a single element list.

Well a Future is no different. You can iterate over the result of a future and process it's results. This will also happen in a background thread.

```
val future = Future{
  val result = someLongRunningThing()
  result
}

future.map{ result => Database.save(result) }
```

Like with a list we can map, filter, flatMap, and iterate over a Future to transform it’s result into another future. In this example a background thread will compute result, then when it completes another background thread will save it to the database.

```
val productsFuture = Future{ getUser() }
.map{ user => Database.save(user) }
.map { dbResponse => Products.get(dbResponse.user.id) }
```

Future and threads

Future and Promises revolve around ExecutionContexts, responsible for executing computations.

An ExecutionContext is similar to an Executor (Java):
it is free to execute computations in a new thread, in a pooled thread or in the current thread

```
val inverseFuture: Future[Matrix] = Future {
  fatMatrix.inverse() // non-blocking long lasting computation
}(executionContext)

implicit val ec: ExecutionContext = ...
val inverseFuture : Future[Matrix] = Future {
  fatMatrix.inverse()
} // ec is implicitly passed
```

More:

https://www.youtube.com/watch?v=K9lt6yjuzbM

Producer/Consumer in Scala

```
import java.util.concurrent.ArrayBlockingQueue
import scala.concurrent.{ExecutionContext, Future}

trait ProducerConsumer[R, S] {
  def produce(state: S): Option[(R, S)]
  def consume(a: R): Unit
}
```

```
object ProducerConsumer {

  def await(fs: Future[Unit]*)(implicit ec: ExecutionContext): Future[Unit] =
Future.reduce(fs){case _ =>}

  class Runner[A, S](task: ProducerConsumer[A, S], maxQueueLength: Int) {
    private[this] val queue = new ArrayBlockingQueue[Option[(A, S)]](maxQueueLength)

    def run(start: S)(implicit ec: ExecutionContext): Future[Unit] = {
      val f1 = Future {
        Iterator.iterate(task.produce(start))(_.flatMap({case (_, s) => task.produce(s)}))
          .takeWhile(_.isDefined)
          .foreach(queue.put)
      }

      val f2 = Future {
        Iterator.continually(queue.take())
          .takeWhile(_.isDefined)
          .flatten
          .foreach({case (res, _) => task.consume(res)})
      }
      await(f1, f2)
}}}
```

Partial functions in Scala

Problem

You want to define a Scala function that will only work for a subset of possible input values, or you want to define a series of functions that only work for a subset of input values, and combine those functions to completely solve a problem.

Solutions

A *partial function* is a function that does not provide an answer for every possible input value it can be given. It provides an answer only for a subset of possible data, and defines the data it can handle. In Scala, a partial function can also be queried to determine if it can handle a particular value.

Sample of the problem:

```
val divide = (x: Int) => 42 / x

scala> divide(0)
java.lang.ArithmeticException: / by zero
```

Solution

```
val divide = new PartialFunction[Int, Int] {
        def apply(x: Int) = 42 / x
        def isDefinedAt(x: Int) = x != 0
}

val divide2: PartialFunction[Int, Int] = {
        case d: Int if d != 0 => 42 / d
}
```

Scala doc:

"A partial function of type PartialFunction[A, B] is a unary function where the domain does not necessarily include all values of type A. The function isDefinedAt allows [you] to test dynamically if a value is in the domain of the function."

The signature of the PartialFunction trait looks like this:

trait PartialFunction[-A, +B] extends (A) => B

As discussed in other recipes, the => symbol can be thought of as a *transformer*, and in this case, the (A) => B can be interpreted as a function that transforms a type A into a resulting type B.

Sample

```
// converts 1 to "one", etc., up to 5
val convert1to5 = new PartialFunction[Int, String] {
  val nums = Array("one", "two", "three", "four", "five")
  def apply(i: Int) = nums(i-1)
  def isDefinedAt(i: Int) = i > 0 && i < 6
}

// converts 6 to "six", etc., up to 10
val convert6to10 = new PartialFunction[Int, String] {
  val nums = Array("six", "seven", "eight", "nine", "ten")
  def apply(i: Int) = nums(i-6)
  def isDefinedAt(i: Int) = i > 5 && i < 11
}

val handle1to10 = convert1to5 orElse convert6to10
```

Implicit Parameters in Scala

Scala “implicits” allow you to omit calling methods or referencing variables directly but instead rely on the compiler to make the connections for you. For example, you could write a function to convert from and Int to a String and rather than call that function *explicitly*, you can ask the compiler to do it for you, *implicitly*.

Implicit Parameters sample

def multiply(implicit by: Int) = value * by

You tell the compiler what it can pass in implicitly but annotating values with implicit

implicit val multiplier = 2

and call the function like this

multiply

The compiler knows to convert this into a call to multiply(multiplier). If you forget to define an implicit var, you'll get an error like the following.

```
  error: could not find implicit value for parameter by: Int
multiply
^
```

Actor Model

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

Defining an Actor class

Actors are implemented by extending the Actor base trait and implementing the receive method. The receive method should define a series of case statements (which has the type PartialFunction[Any, Unit]) that defines which messages your Actor can handle, using standard Scala pattern matching, along with the implementation of how the messages should be processed.

Sample

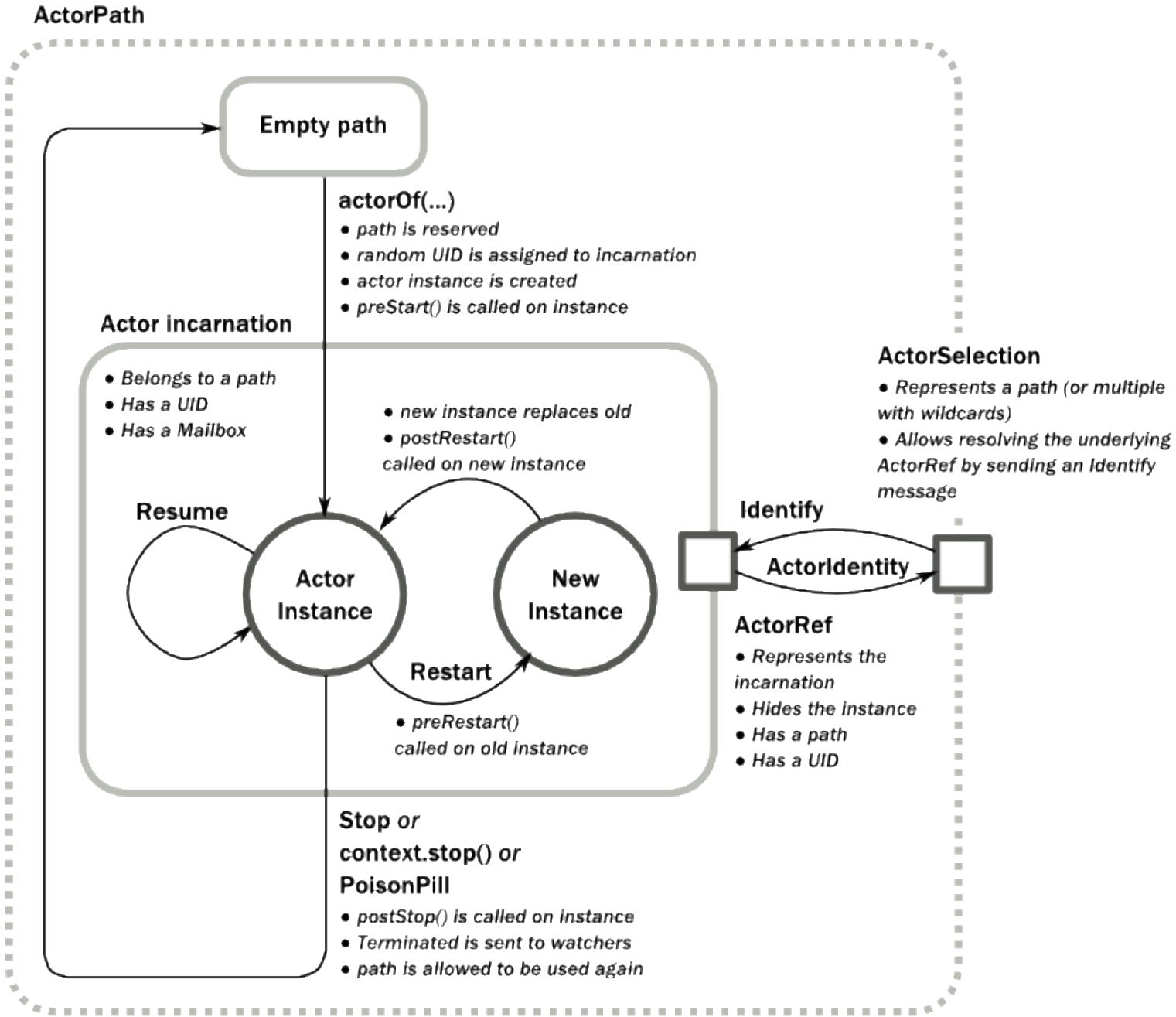
```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

```
import akka.actor.{Actor, ActorRef, ActorSystem, Props}
class Greeter extends Actor {
  def receive = {
    case "hello" => println("Hello World!")
    case msg => println(s"Unexpected message '$msg' ")
  }
}


object Main extends App {
  val ourSystem = ActorSystem("MySystem")
  val greeter: ActorRef = ourSystem.actorOf(Props[Greeter])
  greeter ! "hello"
  greeter ! ”bye"
  greeter ! "hello"
  Thread.sleep(1000)
  ourSystem.terminate
}
```


ActorPath
Empty path
actorOf(...)
• path is reserved
• random UID is assigned to incarnation
• actor instance is created
• preStart() is called on instance
Actor incarnation
• Belongs to a path
• Has a UID
• Has a Mailbox
• new instance replaces old
• postRestart()
called on new instance
Resume
Actor Instance
New Instance
Restart
• preRestart()
called on old instance
Stop or
context.stop() or
PoisonPill
• postStop() is called on instance
• Terminated is sent to watchers
• path is allowed to be used again
ActorSelection
• Represents a path (or multiple with wildcards)
• Allows resolving the underlying ActorRef by sending an Identify message
Identify
ActorIdentity
ActorRef
• Represents the incarnation
• Hides the instance
• Has a path
• Has a UID

Akka typed

The new Actor APIs are known as Akka Typed because the actors must declare what messages types they can handle. Messages are sent via a location transparent handle to the actor, so called ActorRef. It's this ActorRef in Akka Typed that has a message type parameter, as well as the corresponding type in the behavior of the actor. Thereby you can only send messages that are part of the actor's defined message protocol, and the compiler will help with catching mistakes early.

Typed and untyped actors can coexist within the same application, within the same ActorSystem. Messages can be sent between both types of actors, they can watch each other and even spawn child actors in both ways. There is no plan of deprecating and removing untyped Actors. An existing application doesn't have to change, but if you like it's possible to introduce Akka Typed for new functionality or migrate at your own pace.

Thank you for your attention

michal.szczepanik@pwr.edu.pl