Funkcionalno programiranje

Vežbe

07 Akteri

Upotreba AKKA biblioteke

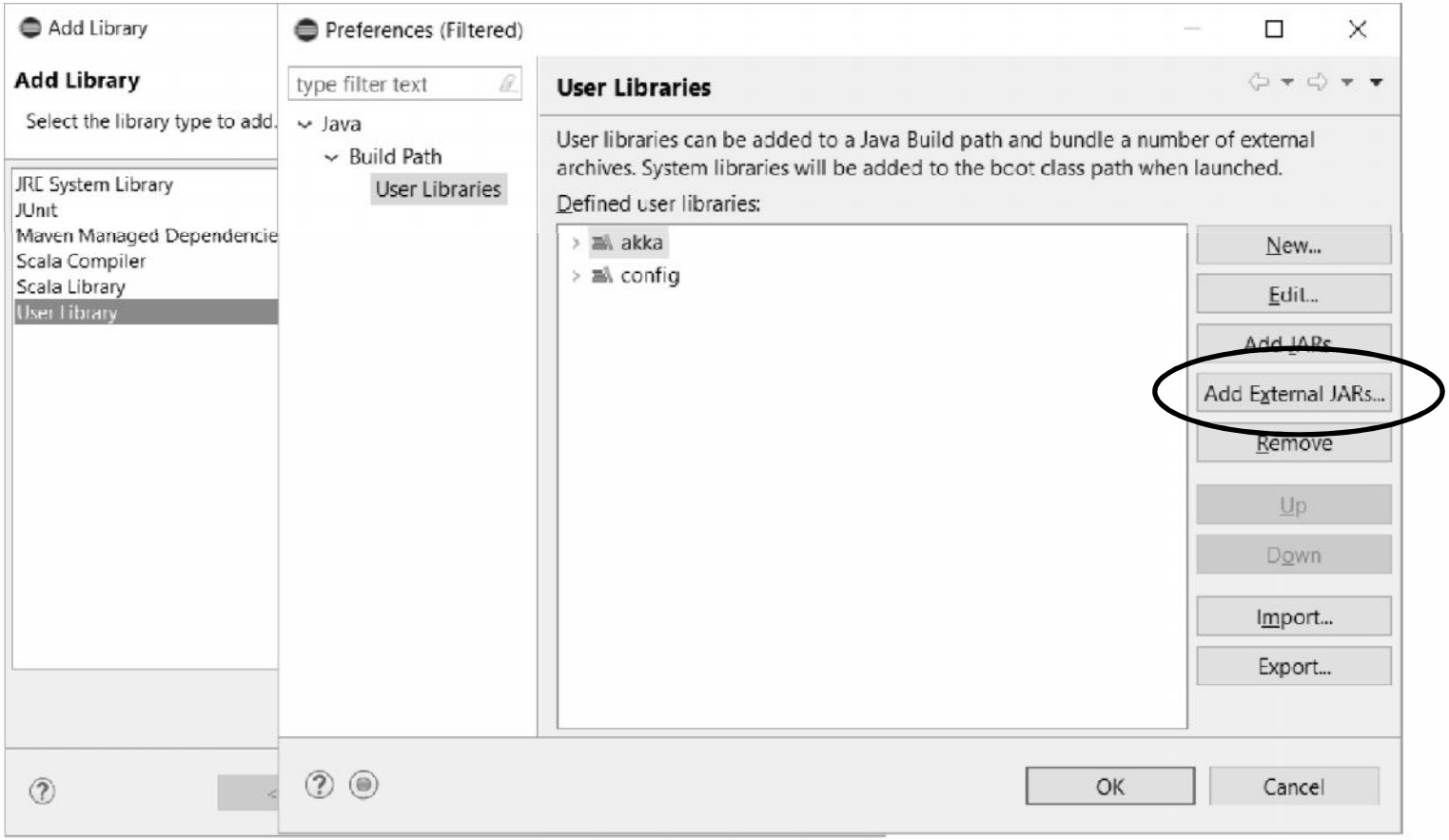
- http://akka.io
- http://akka.io/downloads/
- Aktuelna verzija 2.5.1

- Jedan od načina: upotreba SBT
  - Scala Build Tool
  - Alat za prevođenje Scala programa
  - Automatizacija prevođenja složenih projekata

- Alternativno: dostaviti .jar fajlove
  - preuzeti arhivu akka distribucije sa zvaničnog sajta
  - u konfiguraciji IDE postaviti odgovarajuće JAR fajlove kao biblioteke

Konfigurisanje

- Eclipse
  - Build path -> Add Library



Zadatak 1

- Napistati program poput višenitnog "ping-pong" programa u jeziku Java.

```
import akka.actor._

case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage
```

Zadatak 1

```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint { count += 1; println("ping") }
  def receive = {
    case StartMessage =>
      incrementAndPrint
      pong ! PingMessage
    case PongMessage =>
      incrementAndPrint
      if (count > 99) {
        sender ! StopMessage
        println("ping zaustavljen")
        context.stop(self)
      } else {
        sender ! PingMessage
      }
    case _ => println("Primljena nepoznata poruka.")
  }
}
```

Zadatak 1

```
class Pong extends Actor {
  def receive = {
    case PingMessage =>
      println(" pong")
      sender ! PongMessage
    case StopMessage =>
      println("pong zaustavljen")
      context.stop(self)
    case _ => println("Primljena nepoznata poruka.")
  }
}
object PingPongTest extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")
  // start the action
  ping ! StartMessage
  //system.shutdown
}
```

Zadatak 2

- Napisati program na programskom jeziku Scala koji provodi aktera kroz njegov životni ciklus.

```
import akka.actor._

class Kenny extends Actor {
  println("entered the Kenny constructor")

  override def preStart { println("kenny: preStart") }

  override def postStop { println("kenny: postStop") }

  override def preRestart(reason: Throwable, message: Option[Any]) {
    println("kenny: preRestart")
    println(s" MESSAGE: ${message.getOrElse("")}")
    println(s" REASON: ${reason.getMessage}")
    super.preRestart(reason, message)
  }
```

Zadatak 2

```
  override def postRestart(reason: Throwable) {
    println("kenny: postRestart")
    println(s" REASON: ${reason.getMessage}")
    super.postRestart(reason)
  }

  def receive = {
    case ForceRestart => throw new Exception("Boom!")
    case _ => println("Kenny received a message")
  }
}
```

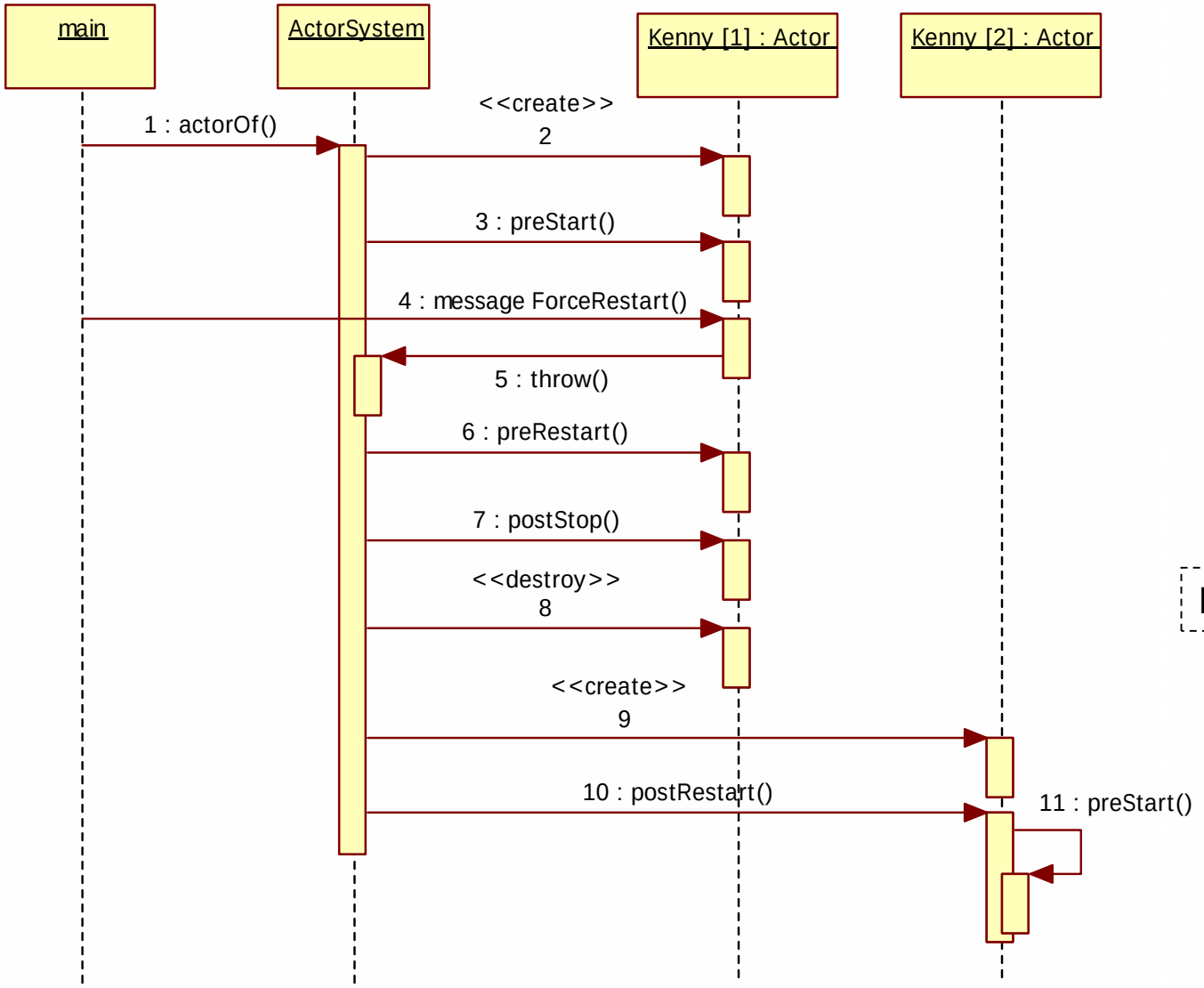
Zadatak 2

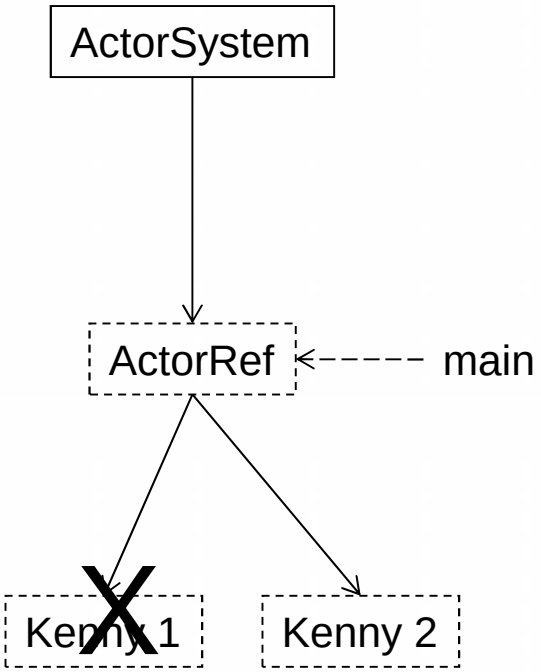
```
case object ForceRestart

object LifecycleDemo extends App {
  val system = ActorSystem("LifecycleDemo")
  val kenny = system.actorOf(Props[Kenny], name = "Kenny")

  println("===> sending kenny a simple String message")
  kenny ! "hello"
  Thread.sleep(1000)

  println("===> make kenny restart")
  kenny ! ForceRestart
  Thread.sleep(1000)

  println("===> killing kenny")
  system.stop(kenny)

  println("shutting down system")
  system.shutdown
}
```

Zadatak 2

```
entered the Kenny 1 constructor
Kenny 1: preStart
===> sending kenny a simple String message
Kenny 1 received a message
==> make kenny restart
Kenny 1 received ForceRestart message
Kenny 1: preRestart
 MESSAGE: ForceRestart
 REASON: Boom!
Kenny 1: postStop
entered the Kenny 2 constructor
Kenny 2: postRestart
 REASON: Boom!
Kenny 2: preStart
[ERROR] [05/17/2017 21:38:54.420] [LifecycleDemo-akka.actor.default-dispatcher-4]
    [akka://LifecycleDemo/user/Kenny] Boom!
java.lang.Exception: Boom!
    ...

===> killing kenny
===> shutting down system
Kenny 2: postStop
```

Zadatak 2


main
ActorSystem
Kenny [1] : Actor
Kenny [2] : Actor
1 : actorOf()
<<create>>
2
3 : preStart()
4 : message ForceRestart()
5 : throw()
6 : preRestart()
7 : postStop()
<<destroy>>
8
<<create>>
9
10 : postRestart()
11 : preStart()


ActorSystem
ActorRef
main
Kenny 1
Kenny 2

Zadatak 3

- Napisati Scala akter koji na poruku `CreateChild(ime)` napravi novi akter kao svog potomka. Akteri-potomci, prilikom zaustavljanja, ispišu svoje ime i putanju.

Zadatak 3

```
import akka.actor._
case class CreateChild (name: String)
case class Name (name: String)

class Child extends Actor {
  var name = "No name"
  override def postStop {
    println(s"Aktor ($name) zaustavljen: ${self.path}")
  }

  def receive = {
    case Name(name) => this.name = name
    case _ => println(s"Potomak $name je primio poruku")
  }
}
```

Zadatak 3

```
class Parent extends Actor {
  def receive = {
    case CreateChild(name) =>
      val child = context.actorOf(Props[Child], name = s"$name")
      child ! Name(name)
    case _ => println("Roditelj je primio poruku.")
  }
}
```

Zadatak 3

```
object ParentTest extends App {
  val actorSystem = ActorSystem("ParentTest")
  val parent = actorSystem.actorOf(Props[Parent], name = "Parent")

  parent ! CreateChild("T-800")
  parent ! CreateChild("T-1000")

  Thread.sleep(500)

  println("Saljem otrovnu pililu (PoisonPill) ka T-800...")
  val t800 = actorSystem.actorSelection("/user/Parent/T-800")

  t800 ! PoisonPill

  println("T-800 neutralizovan")
  Thread.sleep(1000)
  actorSystem.shutdown
}
```

Zadatak 4

- Napisati Scala akter kome se ponašanje prosleđuje kao poruka.

```
import akka.actor._

case class Become(r : Actor.Receive)
case class Message(s : String)

class M extends Actor {

  def receive = {
    case Become(x) => context.become(x, false)
    case Message(s) => println(s)
    case _ => println("nepoznato")
  }

}
```

Zadatak 4

```
object Mutator extends App {
  val system = ActorSystem("system")
  val actor = system.actorOf(Props[M], name = "M")

  actor ! Message("Poruka")

  def laz : Actor.Receive =
  {
    case Message(s) => println("Nisam dobio poruku: " + s)
    case _ => println("Nisam nista primio")
  }

  actor ! Become( laz )
  actor ! Message("Zdravo")

  Thread.sleep(1000)
  system.shutdown()
}
```

Zadatak 5

- Demonstrirati obradu podataka pomoću budućih vrednosti
  - kada se izvršenje blokira čekajući buduću vrednost
  - kada završetak izračunavanja buduće vrednosti izazove događaj

Zadatak 5

```
// sa blokiranjem
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object AwaitForFuture extends App {

  // Napravi buduću vrednost
  val f = Future {
    Thread.sleep(500)
    1 + 1
  }
  // Blokiraj izvršenje niti dok buduća vrednost ne završi
  // Baci izuzetak ako vreme istekne
  val result = Await.result(f, 1 second)

  println(result)
  Thread.sleep(1000)
}
```

Zadatak 5

```
// bez blokiranja
import scala.concurrent.{Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
import scala.util.Random
object FutureCallback extends App {
  println("počinjem ...")
  val f = Future { Thread.sleep(Random.nextInt(500)); 42 }
  println("Pre registrovanja povratnog poziva")
  f.onComplete {
    case Success(value) => println(s"Uspesno: $value")
    case Failure(e) => e.printStackTrace
  }
  println("A ..."); Thread.sleep(100)
  println("B ..."); Thread.sleep(100)
  println("C ..."); Thread.sleep(100)
  println("D ..."); Thread.sleep(100)
  println("E ..."); Thread.sleep(100)
  println("F ..."); Thread.sleep(100)
  Thread.sleep(2000)
}
```

```
počinjem  ...
Pre registrovanja povratnog poziva
A ...
B ...
C ...
Uspesno: 42
D ...
E ...
F ...
```

Zadatak 5

```
// bez blokiranja, resenje sa dva dogadjaja
import ...
object FutureTwoCallbacks extends App {
  val f = Future {
    Thread.sleep(Random.nextInt(500))
    if (Random.nextInt(500) > 250)
       throw new Exception("Izuzetak!") else 42
  }
  f onSuccess { case result => println(s"Uspesno: $result") }
  f onFailure { case t => println(s"Izuzetak: ${t.getMessage}") }
  // do the rest of your work
  println("A ..."); Thread.sleep(100)
  println("B ..."); Thread.sleep(100)
  println("C ..."); Thread.sleep(100)
  println("D ..."); Thread.sleep(100)
  println("E ..."); Thread.sleep(100)
  println("F ..."); Thread.sleep(100)
}
```

Zadatak 6

- Prikazati slučaj komunikacije dva aktera kod kojih inicijator poruke (pitanja) čeka na odgovor

```
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._

case object AskNameMessage

class TestActor extends Actor {
  def receive = {
    case AskNameMessage => sender ! "Kenny"
    case _ => println("??")
  }
}
```

Zadatak 6

```
object AskTest extends App {

  // Napravi aktora
  val system = ActorSystem("AskTestSystem")
  val myActor = system.actorOf(Props[TestActor], name = "myActor")

  // Pitaj i čekaj odgovor
  implicit val timeout = Timeout(5 seconds)
  val future = myActor ? AskNameMessage
  val result = Await.result(future, timeout.duration)
                    .asInstanceOf[String]
  println(result)

  // Pitaj i odmah konvertuj u string
  val future2: Future[String] =
                       ask(myActor, AskNameMessage).mapTo[String]
  val result2 = Await.result(future2, 1 second)
  println(result2)

  system.shutdown
}
```