Funkcionalno programiranje

Konkurentno programiranje i buduće vrednosti

Uvod

- Konkurentno programiranje u jeziku Scala zasniva se na konceptu aktera
  - koncept vodi poreklo iz jezika Erlang
  - “If Java is 'write once, run anywhere', then Erlang is 'write once, run forever’.” – Joe Armstrong
- Generalna ideja:
  - akteri su učesnici u komunikaciji
  - razmenjuju poruke
  - jedini način komunikacije: kroz razmenu poruka
- Akteri mogu da prime bilo kakvu poruku
  - posao aktera je da analizira i evaluira prispelu poruku
  - na osnovu toga odlučuje da li da je odbaci ili nešto uradi
  - šta će uraditi zavisi od poruke, ali i unutrašnjeg stanja aktera
- Akter = onaj ko reaguje na prispeće poruke

Uvod

- Akteri su realizovani u vidu "lakih" niti
  - JVM je ograničena na nekoliko hiljada niti
  - Aktera može biti za nekoliko redova veličine više
  - Veći potencijal za skalabilnost

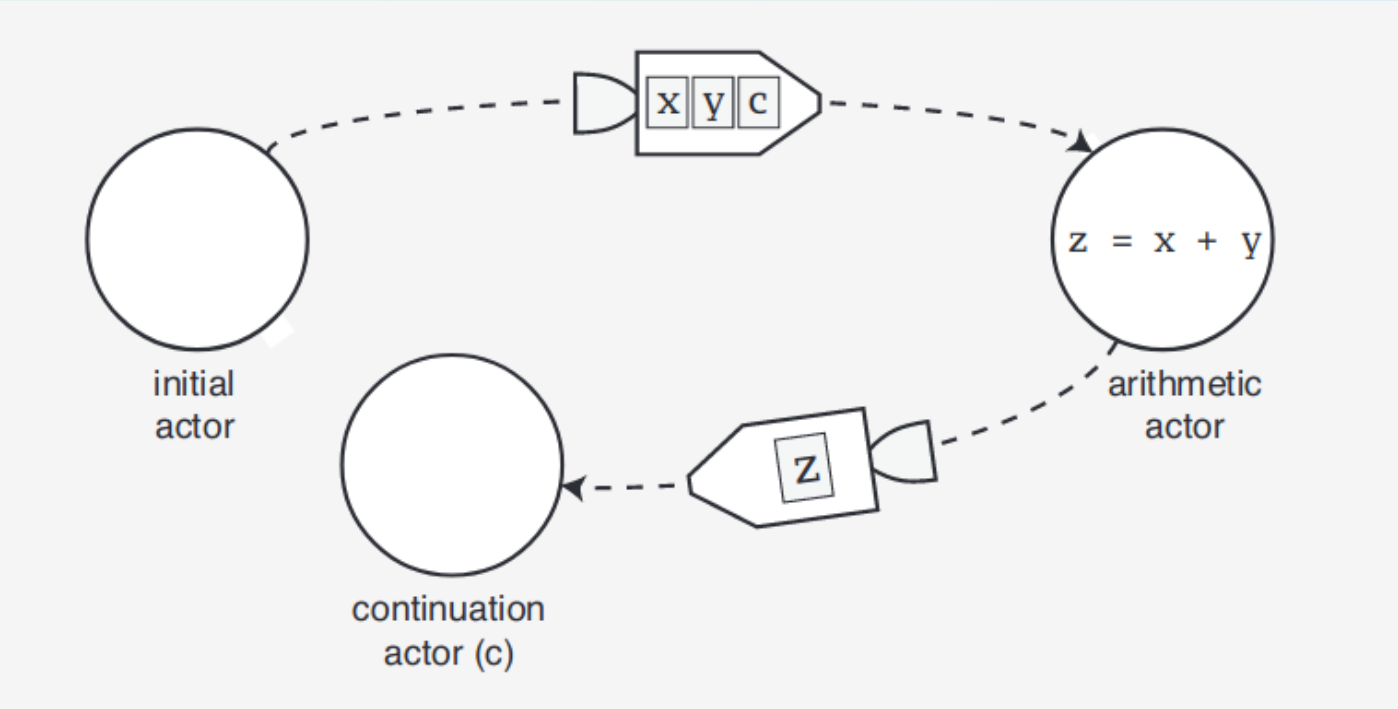
- Do verzije 2.10, jezik Scala je u svojoj biblioteci podržavao aktere
- Od verzije 2.11 koristi se biblioteka AKKA
  - palindrom od AK – action kernel
  - takođe – ime boginje iz švedske mitologije

- "Actors in Scala"
  - Philipp Haller, Frank Sommers, Aritma 2012

Primer upotrebe aktera

- Primer:
  - aritmetički akter prima dva broja
  - njegov posao je da ta dva broja sabere
- Takav akter nema mnogo smisla: gde se koristi proizvod njegovog rada?
- Trebalo bi da se rezultat prosledi drugom akteru (ili onom koji je poslao zahtev za sabiranje)

Primer upotrebe aktera



- Akter c se još naziva *nastavljajući* akter – nastavlja lanac
- Programski stil kod kojeg se u poruci dostavlja i nastavljajući akter se zove *continuation-passing* (CPS)

Objedinjuje tok kontrole i tok podataka

- Prenosom podataka i nastavljajućeg aktera u jednoj poruci, model aktera objedinjuje
  - tok kontrole
  - tok podataka
- To omogućava jednostavniji dizajn programa zasnovanih na akterima
- Neke od prednosti:
  - modularni dizajn
  - sastavljanje lanca aktera u vreme izvršenja programa
  - inkrementano dodavanje elemenata
- Akter koji primi poruku može stvoriti druge aktere koji bi obavili posao (fork / join)
- Veoma liči na način komunikacije objekata u OOP
  - ali uzima u obzir asinhronost u komunikaciji

Doprinos aktera

- U konkurentnom programiranju, tok podataka i tok kontrole programa postaju teški za razumevanje sa porastom programa
  - jedno rešenje: uvođenje serijalizacije u tok podataka
  - čak i tada je podložno greškama
- Mana rešenja: smanjuje se konkurentnost izvršavanja
  - deo programa se efektivno pretvara u sekvencijalni
  - samo jedna nit može da pristupa deljenom (ili globalnom) stanju
- Doprinos:
  - definiše kontrolne strukture tako da minimizuje zavisnost od globalnog stanja programa
  - odluke o toku su kapsulirane u objektima, (uglavnom) lokalno stanje se koristi za odlučivanje
  - → *princip lokalnosti*

Model aktera

- Smatra se da je slanje **asinhrona** operacija
- Može proteći proizvoljno vremena od kada A1 uputi poruku ka A2, do trenutka kada A2 primi poruku
  - kašnjenje u mreži
  - zato što je A1 poslao poruku do A3, a A2 je nastavljajući akter
- A1 odmah nastavlja sa radom, nezavisno od dalje sudbine poruke
  - eventualni odgovor takođe stiže asinhrono
  - pretpostavka je da se poruke ne gube
  - ali vreme stizanja odgovora nije ograničeno
- Postoji i podrška za sinhronu komunikaciju

Nedefinisan redosled prijema poruka

- Model aktera ne definiše mehanizam pouzdane isporuke poruka (ne bavi se tim problemom)
- Model podrazumeva da mnogo poruka može biti poslato u kratkom vremenskom intervalu
  - poruke se ne smeju izgubiti
- Zbog toga model zahteva da postoji specijalan objekat koji prima i čuva poruke sve dok ih akter ne obradi
  - poštansko sanduče (mailbox)
- Poruke se ne moraju čitati prema redosledu prispeća
  - po uzoru na "pravo" poštansko sanduče
  - redosled isporučivanja je suštinski nedefinisan
- Zbog toga ispravnost programa **ne sme da zavisi** od redosleda prispeća poruka

Crta Actor

```
trait Actor extends AnyRef {
  type Receive = PartialFunction[Any, Unit]
  protected abstract def receive : Receive

  // Stores the context for this actor, including self, and sender.
  implicit val context : ActorContext
  def postRestart (reason: Throwable): Unit
  def postStop (): Unit
  def preRestart (reason: Throwable, message: Option[Any]): Unit
  def preStart (): Unit
  implicit val self : ActorRef
  def sender : ActorRef
  def supervisorStrategy (): SupervisorStrategy
  def unhandled (message: Any): Unit

}
```

Detalji crte Actor

- Jedna apstraktna metoda: `receive`
  - ako akter ne ume da obradi prispelu poruku – izuzetak
- Tip `ActorRef`
  - nepromenljiv objekat
  - omotač oko pravog aktera
  - to je jedini način da se interaguje sa akterom
  - serijalizabilan i "svestan" mreže
    - može da se snimi (serijalizuje)
    - može da se pošalje putem mreže i iskoristi na udaljenom računaru, jer se i dalje "zna" na kom računaru se akter nalazi
- `self`: referenca ka ActorRef objektu aktera
- `sender`: referenca ka akteru koji je poslao poruku
- `supervisorStrategy`: definicija strategije za nadgledanje aktera-potomaka

Detalji crte Actor

- `context` akteru izlaže informaciju o kontekstu i trenutnoj poruci
  - ima proizvodni metod za stvaranje aktera-potomaka (`actorOf`)
  - sistem kojem akter pripada
  - referenca ka roditelju – supervizoru
  - nadgledani akteri – potomci
  - praćenje životnog ciklusa

Pravljenje novog aktera

- Akteri se implementiraju umetanjem (nasleđivanjem) crte `Actor` i implementiranjem metode `receive`
- Metoda `receive` treba da obradi niz `case` iskaza
  - koristi se standardno uparivanje obrazaca
  - uparen obrazac definiše očekivano ponašanje
  - niz vraća tip `PartialFunction[Any, Unit]`
- Treba da obradi sve moguće vrste poruka
  - koristiti džoker znak
  - ako to ne uradi, baca se izuzetak u slučaju prijema neočekivane poruke

```
akka.actor.UnhandledMessage(message, sender, recipient)
```

Pravljenje novog aktera - klasa Props -

- Konfiguraciona klasa
  - služi za zadavanje opcija prilikom stvaranja aktera
  - nepromenljiv objekat

- Može se stvoriti na nekoliko načina

```
import akka.actor.Props

val props1 = Props[MyActor]          // u redu
val props2 = Props( new MyActor ) // opasno u dosegu drugog aktora,
                                     // može dovesti do utrkivanja
```

Pravljenje novog aktera

- Akteri se stvaraju prosleđivanjem `Props` objekta u `actorOf` proizvodnu metodu koja je dostupna u objektima `ActorSystem` i `ActorContext`
- `ActorSystem` je "teški" objekat
  - hijerarhijska grupa aktera
  - upravlja izvršenjem i raspoređivanjem aktera
  - preporučuje se stvaranje samo jedne instance po aplikaciji

```
import akka.actor.ActorSystem
// stvaranje nezavisnog aktera ("top level")
val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor2")

// stvaranje aktera-potomka
class FirstActor extends Actor {
  val child = context.actorOf(Props[MyActor], name = "myChild")
  // ...
}
```

Primer

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
object Hello extends App {
  class HelloActor extends Actor {
    def receive = {
      case "hello" => println("hello back at you " + sender() )
      case _ => println("no comprendo...")
    }
  }
  val system = ActorSystem("HelloSystem") // pravi aktore
  val helloActor = system.actorOf(Props[HelloActor],
                                  name = "helloactor")
  helloActor ! "hello"
  helloActor ! "buenos dias"
  system.shutdown
}
```

```
hello back at you Actor[akka://HelloSystem/deadLetters]
no comprendo...
```

Primer

- Obratiti pažnju: slanje je asinhrono!

```
helloActor ! "hello"
helloActor ! "buenos dias"
println("test")
```

```
test
hello back at you Actor[akka://HelloSystem/deadLetters]
no comprendo...
```

Specifičnosti

- Poziv `actorOf` vraća instancu tipa `ActorRef`
- Ime aktera je opciono, ali se preporučuje imenovanje
  - ime ne sme biti prazno ili da počinje sa $
  - u slučaju dupliranog imena u okviru istog roditelja – baca izuzetak
- Akteri se ne zaustavljaju automatski (na primer, kada se više ne referenciraju) – moraju biti eksplicitno uništeni
- Zaustavljanje roditeljskog aktera automatski zaustavlja sve aktere potomke

become / unbecome

- ActorContext klasa ima metode `become` i `unbecome`
- Namena: promeniti ponašanje aktera
- Staro ponašanje se pamti na namenskom steku

```
abstract def become(behavior: Receive, discardOld: Boolean): Unit
```

  - `behavior` postaje novi `receive: PartialFunction[Any, Unit]`
  - `discardOld=true` – zameni element na vrhu steka (default)
  - `discardOld=false` – push new

```
abstract def unbecome(): Unit
```

  - skida sa steka tekuće ponašanje
  - novo ponašanje je ono na vrhu steka

Nadgledanje životnog ciklusa aktera

- Kada je potrebna notifikacija o permanentnom prestanku rada drugog aktera (za razliku od restartovanja)
  - zainteresovan akter se prijavi za prijem poruke `Terminated`
  - funkcionalnost obezbeđuje `DeathWatch` komponenta iz sistema aktera

```
import akka.actor.{ Actor, Props, Terminated }
class WatchActor extends Actor {
  val child = context.actorOf(Props.empty, "child")
  context.watch(child) // dovoljno za registraciju
  var lastSender = context.system.deadLetters

  def receive = {
    case "kill" =>
      context.stop(child); lastSender = sender()
    case Terminated(`child`) => lastSender ! "finished"
  }
}
```

Nadgledanje životnog ciklusa aktera

- Poruka `Terminated` se generiše nezavisno od redosleda u kojem se događaju registracija i terminacija
  - posmatrajući akter će dobiti poruku čak i kada je akter već završen u trenutku registracije
- Višestruka registracija ne implicira i višestruke Terminated poruke
  - ali ne postoji garancija da će samo jedna Terminated poruka biti primljena
- Deregistracija: `context.unwatch(actor)`
  - funkcioniše čak i kada je `Terminated` poruka prethodno stigla u poštansko sanduče
  - posle poziva, više neće biti obrađivane `Terminated` poruke za toga aktera

Nadgledanje životnog ciklusa aktera

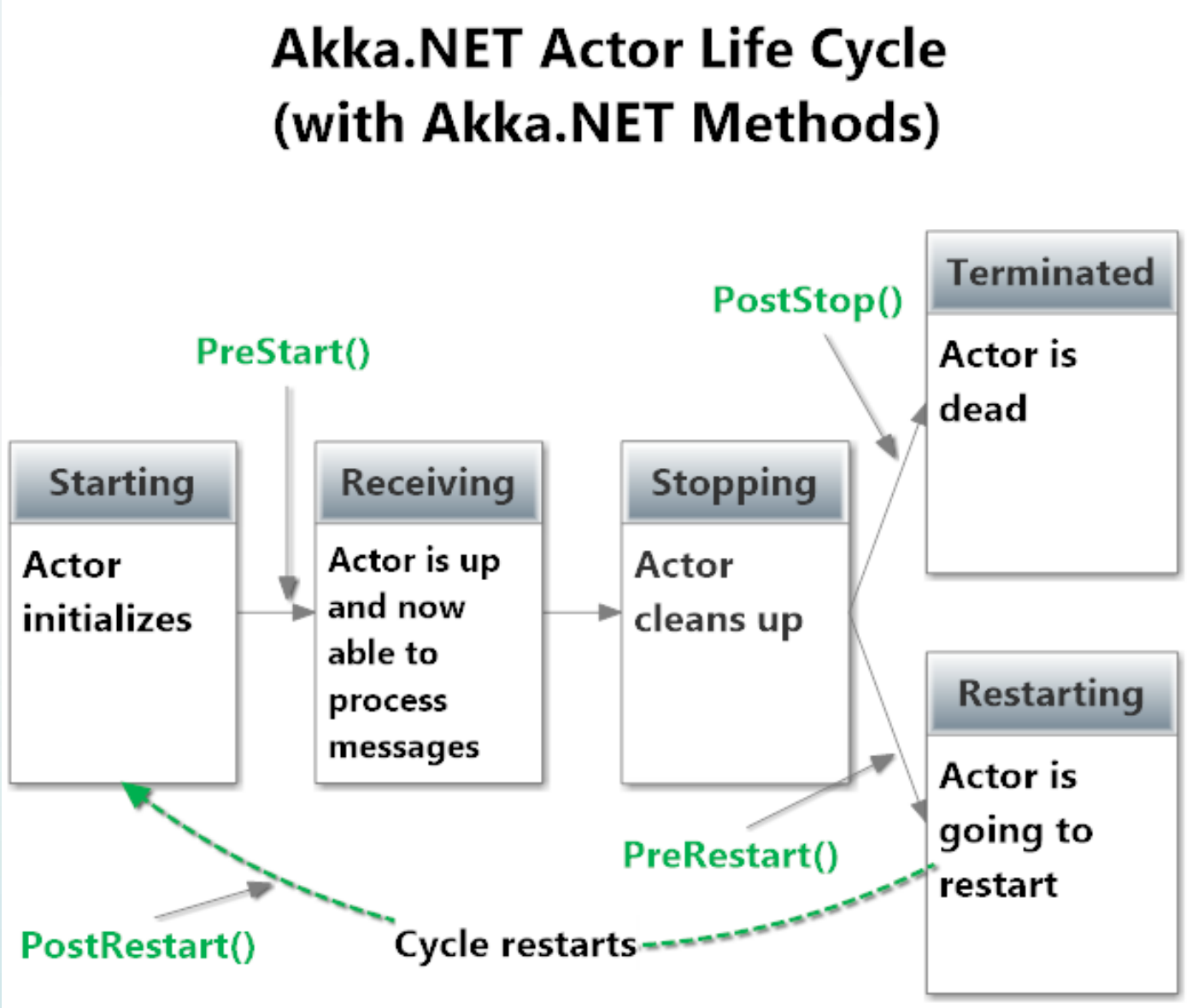
- Odmah po startovanju aktera, poziva se `preStart` metoda
  - poziva se kada se akter po prvi put stvara
  - poziva se iz podrazumevane implementacije `postRestart` metode
  - nadjačavanjem te metode može da se kontroliše da li se metoda `preStart` poziva samo jednom ili pri svakom restartu

- Akteri mogu biti restartovani ako se baci izuzetak tokom obrade poruka
  - staroj inkarnaciji aktera se poziva `preRestart` sa kontekstom koji je doveo do izuzetka
  - novoj inkarnaciji aktera se poziva `postRestart` sa izuzetkom koji je izazvao restart (default: poziva se `preStart`)
  - sadržaj poštanskog sandučeta se ne menja restartom, pa se obrada poruka nastavlja (bez poruke koja je izazvala restart)

Nadgledanje životnog ciklusa aktora

- Akteri se zaustavljaju metodom `stop`
  - iz klase `ActorContext`: zaustavljanje aktera ili njegove dece
  - iz klase `ActorSystem`: za aktere najvišeg nivoa
- Zaustavljanje je asinhrono
- Po zaustavljanju aktera, poziva se njegova `postStop` metoda
  - može da se koristi za odjavljivanje aktera od nekih servisa
- Poziv se garantovano obavlja nakon sprečavanja daljeg dodavanja poruka u sanduče
  - eventualne poruke prosleđuju se *deadLetters* akteru

Životni ciklus aktera



Nadgledanje životnog ciklusa aktora

- Zaustavljanje se obavlja u 2 koraka

1
- Suspenduje poštansko sanduče
- Šalje `stop` svoj deci

2
- Obrađuje notifikacije o zaustavljanju od dece
- Zaustavlja sebe
  - šalje postStop
  - uništava poštansko sanduče
  - objavljuje `Terminated` na `DeathWatch`, čime obaveštava supervizora

- Nakon toga, sistem poziva `postStop` metodu

Primer zaustavljanja

```
class MyActor extends Actor {

  val child: ActorRef = ...

  def receive = {
    case "interrupt-child" => context stop child

    case "done" => context stop self
  }

}
```

Primer kontrolisanog zaustavljanja

```
object Manager {
  case object Shutdown
}

class Manager extends Actor {
  import Manager._
  val worker = context.watch(context.actorOf(Props[Cruncher], "worker"))

  def receive = {
    case "job" => worker ! "crunch"
    case Shutdown =>
      worker ! PoisonPill
      context become shuttingDown
  }

  def shuttingDown: Receive = {
    case "job" => sender() ! "service unavailable, shutting down"
    case Terminated(worker) => context stop self
  }
}
```

Akteri i izuzeci

- Ako se izuzetak baci dok se poruka prosleđuje akteru
  - poruka se gubi (ne vraća se u sanduče)
  - programer mora da uhvati izuzetak i ponovi obradu, ako je poruka bitna
  - voditi računa – nekako ograničiti broj ponavljanja (inače – livelock)
  - sadržaj sandučeta ostaje nepromenjen, čak i kada se restartuje akter
- Ako izuzetak baci kod u samom akteru
  - akter se suspenduje i nadgledajući proces se pokreće
  - akter može biti nastavljen, restartovan ili okončan

Proširivanje aktera parcijalnim funkcijama

- Ponašanje aktera se može zadati kompozicijom više funkcija
- Ovo je moguće zato što `receive` metoda vraća `Actor.Receive`, što je alijas tipa za `PartialFunction[Any,Unit]`
- Parcijalna funkcija je funkcija koja je definisana samo za neke vrednosti ulaznih parametara
  - selekcija `case` je zapravo jedna parcijalna funkcija
- Primer:

```
val one: PartialFunction[Int, String] = { case 1 => "jedan" }
val two: PartialFunction[Int, String] = { case 2 => "dva" }
val wildcard: PartialFunction[Int, String] = { case _ => "drugo" }
val partial = one orElse two orElse wildcard
partial(5)     // drugo
partial(2)     // dva
```

Proširivanje aktera parcijalnim funkcijama

- Primer: akter koji je istovremeno i proizvođač i potrošač

```
trait ProducerBehavior {
  this: Actor =>

  val producerBehavior: Receive = {
    case GiveMeThings => sender() ! Give("thing")
  }
}

trait ConsumerBehavior {
  this: Actor with ActorLogging =>

  val consumerBehavior: Receive = {
    case ref: ActorRef => ref ! GiveMeThings

    case Give(thing) => println("Dobio sam: " + thing)
  }
}
```

Proširivanje aktora parcijalnim funkcijama

```
class Producer extends Actor with ProducerBehavior {
  def receive = producerBehavior
}

class Consumer extends Actor with ActorLogging with ConsumerBehavior {
  def receive = consumerBehavior
}

class ProducerConsumer extends Actor with ActorLogging
                                     with ProducerBehavior
                                     with ConsumerBehavior {
   def receive = producerBehavior.orElse[Any, Unit](consumerBehavior)
}

// protocol
case object GiveMeThings
final case class Give(thing: Any)
```

Vremenska kontrola izvršavanja

http://doc.akka.io/docs/akka/2.5.1/scala/howto.html#Scheduling_Periodic_Messages

- `ActorSystem` ima mogućnost raspoređivanja događaja (slanja poruka)
- Može da se kontroliše
  - slanje poruka datom aktoru
  - izvršavanje zadataka (funkcija ili `Runnable` objekata)
- Nije namenjen dugoročnom planiranju događaja (trenutna implementacija < 8 meseci)
- Nije namenjen preciznom okidanju događaja
  - pravi bakete poslova
  - baketi se prazne prema fiksnom rasporedu
  - na tikove pokreće sve čemu je isteklo predviđeno vreme

Primer

```
import akka.actor._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
object Clock extends App {
  case class Poruka()
  case class Gotovo()
  class Ispisivac extends Actor {
    def receive = {
      case Poruka() => println("protekla 1 sekunda")
      case Gotovo() => context stop self
    }
  }
  val system = ActorSystem("system")
  val actor = system.actorOf(Props( new Ispisivac ), name = "ispis")
  system.scheduler.schedule(0 seconds, 1 seconds, actor, Poruka())
  system.scheduler.scheduleOnce(5 seconds, actor, Gotovo())
}
```

Buduće vrednosti
Futures (poglavlje 32)

- Alternativni mehanizam za ostvarivanje konkurentnosti
- Predstavlja očekivani rezultat neke funkcije koja u trenutku stvaranja objekta još uvek nije završila (ili čak nije počela izvršavanje)
- U jeziku Scala je moguće primeniti transformacije nad budućim vrednostima koje još uvek nisu izračunate
  - nije neophodno blokiranje dok se čeka vrednost (za razliku od Jave)
  - povećava konkurentnost
- Rezultat transformacije buduće vrednosti je buduća vrednost
  - predstavlja asinhroni rezultat izvorne buduće vrednosti transformisane funkcijom

Buduće vrednosti
Futures

- Nit koja zapravo vrši računanje buduće vrednosti određuje se implicitno dostavljenim *izvršnim kontekstom* (execution context)
- Glavna prednost: programer opisuje asinhrona računanja kao niz transformacija nad nepromenljivim vrednostima
  - ne mora da se razmišlja o deljenoj memoriji
  - ne mora da se razmišlja o sinhronizaciji
- Zbog kompatibilnosti sa Javom, Scala podržava
  - metode modela *monitora*: `wait`, `notify` i `notifyAll`
  - predefinisanu metodu `synchronized` (ne postoji direktiva)

```
var brojac = 0
synchronized {
  // Samo jedna nit u datom trenutku
  brojac = brojac + 1
}
```

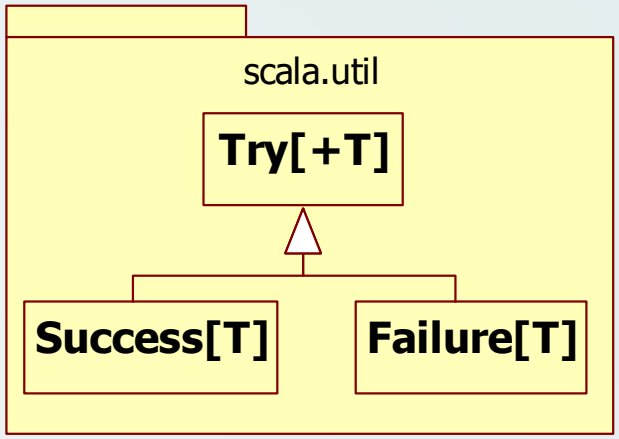
Kontekst izvršenja

- Buduća vrednost predstavlja obradu (računanje) koja će asinhrono biti izvršena (možda druga nit)
- Zateva poseban kontekst za svoje izvršenje
  - kontekst se implicitno usvaja
  - potrebno je da bude eksplicitno uvezen

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
object Futures extends App {

    val fut = Future { Thread.sleep(10000); 21 + 21 } // Future.apply
    // fut: scala.concurrent.Future[Int] = Future(<not completed>)
    println(fut.isCompleted) // false
    println(fut.value) // None
    // tip je: Option[scala.util.Try[Int]]
    ...
    println(fut.value) // Some(Success(42))
}
```

Try / Success / Failure



- Vrednost buduće vrednosti je:
  - uspeh (sadrži vrednost tipa T - rezultat)
  - neuspeh (sadrži izuzetak tipa java.lang.Throwable)
- Ovakav pristup omogućava da se obrađuju buduće vrednosti čije izračunavanje dovodi do bacanja izuzetka
- "Asinhroni `try-catch`"

Transformacije nad budućim vrednostima

- Načelno: umesto da se obrada blokira dok buduća vrednost ne bude spremna, pravi se nova buduća vrednost
- Primer:

```
scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...

scala> val result = fut.map(x => x + 1)
result: scala.concurrent.Future[Int] = ...

scala> result.value
res5: Option[scala.util.Try[Int]] = None

...

scala> result.value
res6: Option[scala.util.Try[Int]] = Some(Success(43))
```