Initsialiseerimine

Objektid luuakse (laisalt) enne esimest kasutamist.

```
object O1 { println("O1 initsialiseerimine") }
object O2 { println("O2 initsialiseerimine") }
object O3 { println("O3 initsialiseerimine") }

object Test {
  println("Test initsialiseerimine")
  def main(args: Array[String]): Unit = {
    val c = O2
  }
  val d = O1
}
```

App

```
object Test extends App {
  val c = new C
  val d = new D
  ...
}
```

- Mis juhtub, kui `C` loomisel lugeda `Test.d`? Miks?

DelayedInit

Klassid ja objektid, mis pärivad `DelayedInit`, muudetakse nii: `code` $\Longrightarrow$ `delayedInit(code)`. S.t

```
trait C1 extends DelayedInit {
  println("C1 initsialiseerimine")
  def delayedInit(body: => Unit): Unit = {
    println("enne C2 initsialiseerimist")
    body       // C2 initsialiseerimine
    println("peale C2 initsialiseerimist")
  }
}

class C2 extends C1 {
  println("C2 initsialiseerimine")
}

object Test {
  def main(args: Array[String]): Unit = {
    val c = new C2
  }
}
```

App

`DelayedInit` kasutatakse ka trait-i `App` poolt.

```
trait App extends DelayedInit { // mõned detailid eemaldatud
  private val initCode = new ListBuffer[() => Unit]

  override def delayedInit(body: => Unit) {
    initCode += (() => body)
  }

  def main(args: Array[String]) = {
    for (proc <- initCode) proc()
  }
}

object Test extends App {
  val c = new C
}
```

Mitmelõimelisus

- Igal klassil (monitoril) on järgnevad meetodid:

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

- Synchronized argument täidetakse teisi lõimi välistavalt: kaks lõime ei saa samaaegselt täita sama monitori *synchronized* koodi.

- Enamasti oodatakse mingi tingimuse C täitumist:

```
o.synchronized{
  ...
  while (!C) wait()
  ...
}
```

Näide: BoundedBuffer

```
class BoundedBuffer[A](N: Int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[A](N)

  def put(x: A) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
    if (n == 1) notifyAll()
  }

  def get: A = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll()
    x
  }
}
```

Näide: BoundedBuffer

- BoundedBuffer kasutamine

```
import scala.concurrent.ops._
...
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } } }
```

- Spawn definitsioon

```
def spawn(p: => Unit) {
  val t = new Thread() { override def run() = p } t.start()
}
```

Sünkroniseeritud muutujad: SyncVar

```
class SyncVar[A] {
  var isDefined: Boolean = false
  var value: A = _

  def get = synchronized {
    while (!isDefined) wait()
    value
  }

  def set(x: A) = synchronized {
    value = x; isDefined = true; notifyAll()
  }

  def isSet: Boolean = synchronized {
    isDefined
  }
  def unset = synchronized {
    isDefined = false
  }
}
```

Tulevikuväärtused (lihtsustatult): `future`

- Väärtus, mida arvutatakse teises lõimes.

```
import scala.concurrent.ops._
...
val x = future(pikk_arvutus)
teine_pikk_arvutus
val y = f(x()) + g(x())
```

- `future` on defineeritud nii:

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

Mitmelõimelisus

- Paketis `scala.concurrent` veel võimalusi: `Promise`, `Channel`, `Lock` jne.
- Keerukust aitab vähendada sobivama arvutusmudeli valimine: `Actor`id.
  - Sõnumite edastamisel põhinev mudel.
  - Aktorid reageerivad sissetulevale sõnumile, seejärel võivad teha kohalikke arvutusi ja omakorda saata sõnumeid.
  - Hea implementatsioon: Akka
  - Aktoreid saab viia ka teise arvutisse.

Akka

Aktoritel põhinev arvutusmudel:

- Programm on hulk aktoreid.
- Aktorile saab saata sõnumeid.
- Aktor töötleb sõnumeid järjest:
  - teeb kohalikke arvutusi ja
  - saadab sõnumeid teistele aktoritele.

Eelised teiste mudelite ees (reklaam):

- Kergekaaluline mudel.
- Koodi struktureerimisel üks kindel viis.
- Aktorid saab paigutada hajusalt.
- Aktorid saavad töötada paralleelselt.

Tegelikult:

- Sobib, kui vaja kiiresti reageerida reaalajas tekkivatele signaalidele.

Kasutusjuhud veebilehelt

- Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)
  - Scale up, scale out, fault-tolerance / HA
- Service backend (any industry, any app)
  - Service REST, SOAP, Cometd, WebSockets etc Act as message hub / integration layer Scale up, scale out, fault-tolerance / HA
- Concurrency/parallelism (any app)
  - Correct Simple to work with and understand Just add the jars to your existing JVM project (use Scala, Java, Groovy or JRuby)
- Simulation
  - Master/Worker, Compute Grid, MapReduce etc.

Kasutusjuhud veebilehelt (cont.)

- Batch processing (any industry)
  - Camel integration to hook up with batch data sources Actors divide and conquer the batch workloads
- Communications Hub (Telecom, Web media, Mobile media)
  - Scale up, scale out, fault-tolerance / HA
- Gaming and Betting (MOM, online gaming, betting)
  - Scale up, scale out, fault-tolerance / HA
- Business Intelligence/Data Mining/general purpose crunching
  - Scale up, scale out, fault-tolerance / HA
- Complex Event Stream Processing
  - Scale up, scale out, fault-tolerance / HA

Mis see praktiliselt tähendab?

```
class A {
  def m(...) {
    ???
  }
}
val a = new A
a.m(argument)
```

⇒

```
case class M(...)
class A extends Actor {
  def receive = {
    case M(...) => ???
  }
}
val a = ActorSystem("s").actorOf(Props[A], "a")
a ! M(argument)
```

või hajusa paigutuse puhul

```
case class M(...)
val a = ActorSystem("s")
        .actorSelection("akka://s@example.com:5678/user/A")
a ! M(argument)
```

- Sõnumi saatmine meetodiga '!'

```
def !(message: Any)(implicit sender: ActorRef =Actor.noSender): Unit
```

Väikseim töötav näide

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

case class M(s: String)

class A extends Actor {
  def receive: PartialFunction[Any, Unit] = {
    case M(s) => println("Hello!")
  }
}

object Test extends App {
  val a = ActorSystem("s").actorOf(Props[A], "a")
  a ! M("Hi!")
}
```

Omavaheline suhtlus

```
case object Pall

class M(s6num: String) extends Actor {
  var vastane: ActorRef = _
  def receive: PartialFunction[Any, Unit] = {
    case a:ActorRef => vastane = a
    case Pall => println(s6num); Thread.sleep(1000); vastane ! Pall
  }
}

object Test extends App {
  val a = ActorSystem("s").actorOf(Props(classOf[M],"Ping"))
  val b = ActorSystem("s").actorOf(Props(classOf[M],"Pong"))
  a ! b
  b ! a
  a ! Pall
}
```

- Peale loomist suhtlus ainult sõnumitega!

Vastamine ja sünkroonne suhtlus

```
import akka.pattern._
import scala.concurrent.ExecutionContext.Implicits.global

class S extends Actor {
  var d: Int = 0
  def receive: PartialFunction[Any, Unit] = {
    case a: Int =>     d += a
    case _: Unit =>    sender ! d
  }
}

object Test extends App {
  implicit val t: Timeout = Timeout(10,TimeUnit.SECONDS)
  val s = ActorSystem("s").actorOf(Props(classOf[S]))
  s ! 10
  s ! 5
  val q : Future[Int] = s ? ()
  q.map(println)
}
```

Kõrvalepõige: Future ja Promise

- Kuidas teha samaaegset arvutust?
- Kuidas teha asünkroonseid meetodikutseid?
- Future!

Tulevikuväärtused

```
object Main {
  def pikk_arvutus1() = 1+1
  def pikk_arvutus2() = 2+2
  def main(args: Array[String]): Unit = {
    val a = pikk_arvutus1()
    val b = pikk_arvutus2()
    println(a+b)
  }
}
```

- Tahame `a` ja `b` arvutada samaaegselt.

Tulevikuväärtused

```
import scala.concurrent.duration.Duration
import scala.concurrent.{Await, ExecutionContext, Future}

object Main {
  def pikk_arvutus1() = 1+1
  def pikk_arvutus2() = 2+2
  def main(args: Array[String]): Unit = {
    val a = Future{pikk_arvutus1()}(ExecutionContext.global)
    val b = pikk_arvutus2()
    println(Await.result(a,Duration.Inf)+b)
  }
}
```

- `Future[Int]`: lubadus tagastada `Int` tüüpi väärtus.

Tulevikuväärtused elegantsemalt

```
object Main {
  implicit val ec: ExecutionContext = ExecutionContext.global
  def pikk_arvutus1() = Future{1+1}
  def pikk_arvutus2() = Future{2+2}
  def main(args: Array[String]): Unit = {
    val a = pikk_arvutus1()
    val b = pikk_arvutus2()
    val c = for (x <- a; y <- b) yield
      println(x+y)
    Await.ready(c,Duration.Inf)
  }
}
```

Future praktiliselt

- `Future[Int]`-l on meetodid
  - `def isCompleted: Boolean`,
  - `def value: Option[Try[Int]]` ja
  - `def onComplete[U](f: Try[Int] =>U)`
    `(implicit executor: ExecutionContext): Unit`
- `Try[A]` on `A` tüüpi väärtus või erind.
- Aga kuidas ise sellist mugavalt implementeerida?

Promise

- `Promise` – ühekordselt kirjutatav muutuja
  - `def tryComplete(result: Try[T]): Boolean`
  - `def isCompleted: Boolean`
  - `def future: Future[T]`

```
object Main {
  implicit val ec: ExecutionContext = ExecutionContext.parasitic
  def main(args: Array[String]): Unit = {
    val parv = Promise[Int]
    val farv = parv.future
    for (x <- farv)
      println(x)
    parv.tryComplete(Try{1})
  }
}
```

Promise ja Future erinevus

- `Future`-d on rangelt kapseldatud
  - Ei pea muretsema ootamise järjekorra pärast.
- `Promise`-d on kapseldamata
  - Programmeerija peab ise hoolitsema, et ta ei hakkaks ootama iseenda järele.

Vastamine ja sünkroonne suhtlus

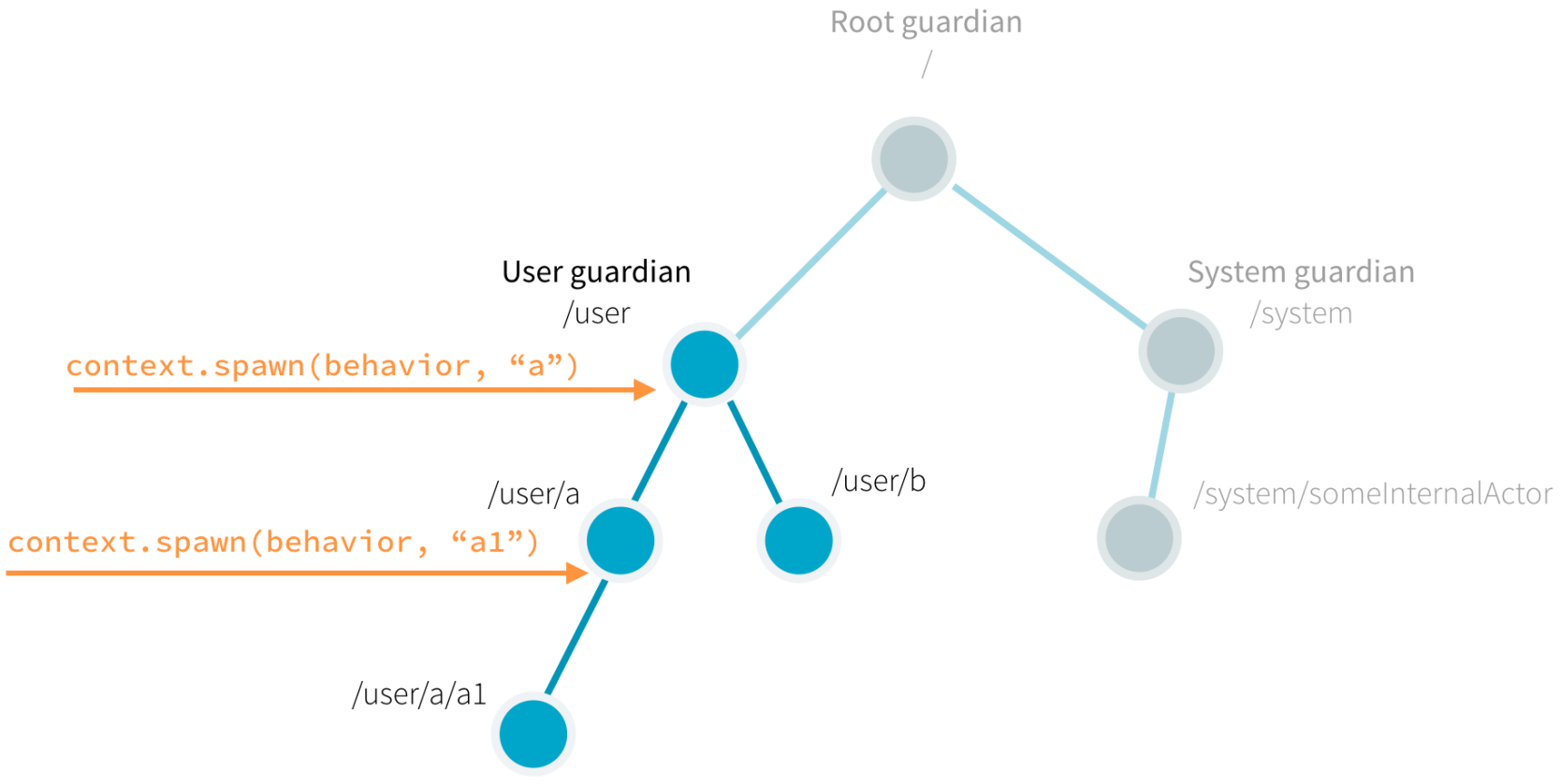
```
class S extends Actor {
  var d: Int = 0
  def receive: PartialFunction[Any, Unit] = {
    case a: Int =>     d += a
    case _: Unit =>    sender ! d
  }
}

object Test extends App {
  implicit val t: Timeout = Timeout(10,TimeUnit.SECONDS)
  val s = ActorSystem("s").actorOf(Props(classOf[S]))
  s ! 10
  s ! 5
  val q : Future[Int] = s ? ()
  q.map(println)
}
```

Edastamata sõnumid

- Sõnumid, mida ei saa edastada saadetakse aktorile `deadLetters`
- Sõnumid saab kätte importides `akka.actor.DeadLatter`.
- Võrguühenduse probleemide korral võivad sõnumid ka kaduma minna.

```
val as = ActorSystem("s")
as.eventStream.subscribe(minuDeadLetterAktor, classOf[DeadLetter])
```

Aktorite hierahia



- Valvurid (i.k. *Guardian*) vastutavad aktorite korrapärase sulgemise eest.

Perioodilised ja ajastatud sõnumid

```
import scala.concurrent.duration._
import system.dispatcher
...

// ajastatud sõnum
system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")

// korduv sõnum
val c = system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}

// tühistamine
c.cancle
```

Aktori peatamine

- stop

```
val actorSystem = ActorSystem("s")
val a = actorSystem.actorOf(Props[A]);
...
actorSystem.stop(a);
```

- PoisonPill

```
a ! PoisonPill
```

- gracefulStop

```
import akka.pattern.gracefulStop

val stopped: Future[Boolean] =
    gracefulStop(a, 2 seconds)(actorSystem)
```

Aktorite mudel

+ Mitmelõimelisusest tulenevad probleemid lahendatud.
+ Süsteemi hajusus kergelt muudetav.
+ Selge viis teatud ülesannete lahendamisel.

- Pole tüübitud
  - Akka Typed!

Akka Typed

```
object MinuAktor {
  def apply(): Behavior[String] =
    Behaviors.setup(context => new MinuAktor(context))
}

class MinuAktor(context: ActorContext[String])
  extends AbstractBehavior[String](context)
{
  override def onMessage(msg: String): Behavior[String] =
    msg match {
      case "start" =>
        println("bla")
        this
    }
}

object Main extends App {
  val testSystem = ActorSystem(MinuAktor(), "s")
  testSystem ! "start"
}
```

Akka Signaalid:

- ChildFailed,
  - erindi tõttu lapsaktoris
- PostStop, PreRestart, PreRestart, Terminated,
- DeleteEventsCompleted, DeleteEventsFailed,
- DeleteSnapshotsCompleted, DeleteSnapshotsFailed,
- RecoveryCompleted, RecoveryFailed,
- SnapshotCompleted, SnapshotFailed.

Aktor mis ei võta sõnumeid vastu

```
object Supervisor {
  def apply(): Behavior[Nothing] =
    Behaviors.setup[Nothing](context => new Supervisor(context))
}
class Supervisor(context: ActorContext[Nothing]) extends
   AbstractBehavior[Nothing](context)
{
  context.log.info("Application␣started")

  override def onMessage(msg: Nothing): Behavior[Nothing] = {
    // Pole vaja sõnumeid töödelda
    Behaviors.unhandled
  }

  override def onSignal: PartialFunction[Signal, Behavior[Nothing]] = {
    case PostStop =>
      context.log.info("Application␣stopped")
      this
  }
}
```