Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Aktoren

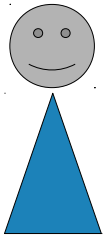
- aktiv – passiv / kontrollorientiert – reaktiv
- Aktoren: Leichtgewichtige Quasi-Threads als reaktive Komponenten
- (Akka-) Aktoren in Scala

SW-Komponenten: Passiv vs. Aktiv

In der klassischen Monitorprogrammierung bildet man zwei primäre Arten von Software-Abstraktionen:

Passive Komponenten

haben keine eigenen Handlungsfaden, ihr Code wird von Threads / Prozessen ausgeführt, die anderen Komponenten zugeordnet sind.



Passiv

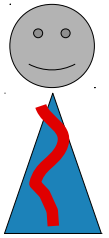
Aktive Komponenten

haben einen eigenen Handlungsfaden, ihr Code wird von einem dezidierten Thread / Prozess ausgeführt.

Sie können blockierende Operationen enthalten in denen sie eventuell blockieren.

Blockade: Die Ausführung wird abgebrochen, der aktuelle Ausführungszustand (Stack!) wird gespeichert.

Der aktuelle Ausführungszustand ist der Kern eines Threads und macht ihn „schwer-gewichtig“.



Aktiv

Die Unterscheidung in aktive und passive SW-Komponenten bildet die Grundausstattung von Monitorprogrammen: der klassischen Nebenläufigkeit.

Passiv / Aktiv : Möglichkeiten und Grenzen

Die Strukturierung einer nebenläufigen Anwendung in aktive und passive Komponenten

hat sich als extrem erfolgreich heraus gestellt

- **Seit mehr 40 Jahren im Einsatz in ungezählten Anwendungen**
- **Ist das Mittel der Wahl für System-nahe Anwendungen, Betriebssysteme, etc.**
- **Nebenläufigkeit war 40 Jahre im Wesentlichen auf das Anwendungsgebiet der systemnahen Programmierung beschränkt**

Passiv / Aktiv : Möglichkeiten und Grenzen

Die Strukturierung einer nebenläufigen Anwendung in aktive und passive Komponenten

Hat sich als problematisch heraus gestellt

- **Technische Problematik:**

  Threads sind oft eine zu aufwändige Technik, um eine SW-Abstraktion auf sie aufzubauen.

  100-te Threads mögen noch OK sein, 100'000-te sind es, auch auf moderner HW, vielleicht nicht mehr

- **SW-Technische Problematik:**

  Monitor-Programme (Threads, Locks, Bedingungsvariablen) sind schwierig zu handhaben

  Für spezialisierte hochqualifizierte Entwickler von Systemprogrammen OK, für bescheidene Anwendungsprogrammierer problematisch

  Höhere (einfacher zu bedienende) Software-Abstraktionen sind notwendig

  (*Concurrency* für die Massen, beginnend mit JUC in Java eingeführt)

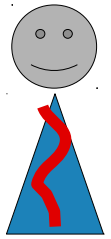
Prozesse: Kontrollorientiert vs. Reaktiv definiert

Prozesse können auf zwei Arten spezifiziert werden:

Als Kontrollorientierte Komponenten

Interaktion von Komponenten mit einem (langem) Handlungsfaden, der eine sequenzielle Folge von Handlungen spezifiziert

Unterbrechungen (Kontextwechsel) an Synchronisationspunkten sind transparent als Blockaden realisiert.

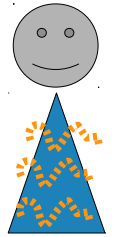


kontrollorientiert

Als Reaktive Komponenten

Interaktion von Komponenten mit mehreren (kurzen) Handlungsfäden, die jeweils eine Reaktion auf ein(-en) Ereignis (-Typ) spezifizieren

Unterbrechungen (Kontextwechsel) während der Verarbeitung eines Ereignisses sind nicht vorgesehen.



reaktiv

Reaktiv und Stromorientiert

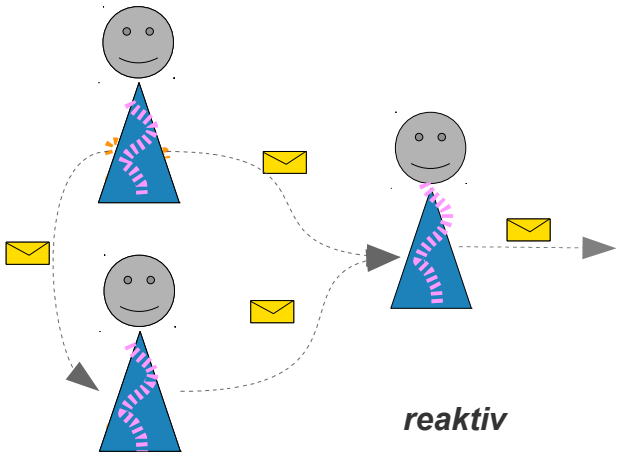
Reaktive Komponenten

Interaktion von Komponenten mit mehreren (kurzen) Handlungsfäden, die jeweils eine Reaktion auf ein(-en) Ereignis (-Typ) spezifizieren

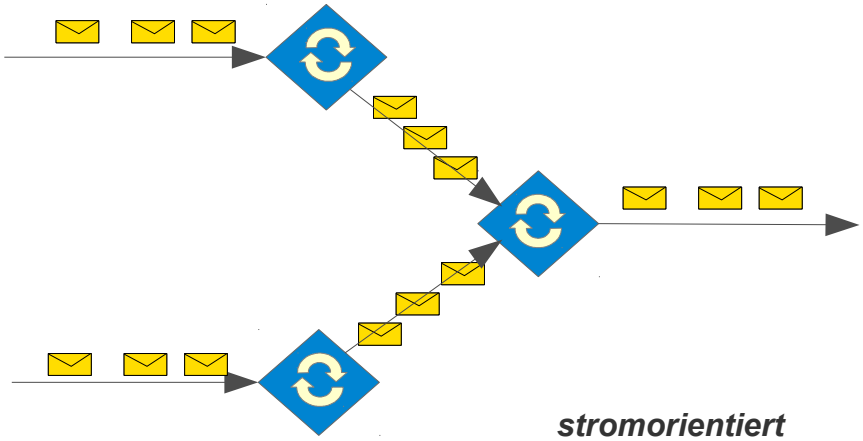
Stromorientiert

Stromorientierte Systeme erlauben es reaktive Komponenten auf einfache Art zu komplexeren Verarbeitungsprozessen zusammen zu schalten.

Stromorientiert = Reaktiv + strukturierte Kombination der Komponenten



reaktiv



stromorientiert

Reaktive Komponenten – Aktoren

Aktoren als Programmkonstrukt

Abgeschlossene reaktive Objekte: Zustand + Verhalten

Aktoren haben

- **einen jeweils eigenen Zustand**
- **es gibt keinen gemeinsamen Zustand**

Sie interagieren

- **ausschließlich über asynchrone Nachrichten**

Jeder Aktor hat eine Mailbox

- **eine Queue für alle eingehenden Nachrichten**

Nachrichtenverarbeitung

- **Eintreffende Nachrichten aktivieren lokale Handler**
- **Handler werden aktiv wenn die Mailbox eine passende Nachricht enthält**
- **Blockaden während der Nachrichtenverarbeitung sind nicht erlaubt (möglich)**

Aktor-Modell

Variante der Modelle verteilte Systeme mit asynchroner Kommunikation

mit sehr hoher Ausdruckskraft (dynamische Topologie; alles kann gesendet werden)

Reaktive Komponenten – Aktoren

Reaktive Komponenten: Aktoren

- haben nur Operationen ohne blockierende Aktionen

  Die Nutzung von synchronisierten passiven Komponenten ist verboten

  Reaktive Komponenten dürfen sich darum nicht in einer passiven Komponente begegnen (Begegnungen aktiver Komponenten erfordert Synchronisation, Synchronisation bedeutet eventuelle Blockaden)

- benötigen wegen der Blockadefreiheit aller Aktionen keinen dedizierten Handlungsfaden (kein dezidierter Thread oder Prozess)

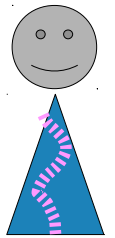
  Sind darum leichtgewichtig – kein Ausführungszustand / kein Stack in Phasen der Inaktivität notwendig!

- können bei Bedarf mit einem „geliehenem“ Leben agieren

  Ausführung der Operationen in einem Threadpool mit wechselnden Threads darum möglich

- benötigen immer direkten oder indirekten Anstoß durch aktive Komponenten

- Werden Aktor genannt



Reaktiv: ein Aktor

Reaktive Komponenten – Aktoren

Aktoren sind Ereignisorientiert

Die Ausführung von Operationen in Aktoren wird in natürlicher Art angesehen als Ereignisverarbeitung:

- **Ereignis ~ Operation mit ihren Argumenten**
- **Ereignisverarbeitung**
  nicht unterbrochene (atomare), nicht blockierende Operation
- **Nach der Ereignisverarbeitung ist die Komponente**
  - **in eine eventuell modifizierten Zustand und**
  - **bereit das nächste Ereignis zu verarbeiten**

Reaktive Komponenten – Aktoren

Aktoren haben zwei Aspekte:

Technischer Aspekt: Implementierung reaktive Komponenten

Java / JVM unterstützt reaktive Komponenten nicht direkt!

Das ist nicht verwunderlich denn

- **reaktive Komponenten benötigen einen angepassten Scheduler (der JVM-Scheduler hantiert mit Threads)**
- **dazu sind spezielle Sprach- / Bibliotheks-Mechanismen notwendig.**

SW-Technischer Aspekt: Reaktiver Programmierstil

Reaktiver Code sieht anders aus, als nicht-reaktiver (kontroll-orientierter) Code

Reaktiver Code ist entlang des Datenflusses organisiert:

- **er besteht aus reaktiven Komponenten**
- **diese reagieren auf Ereignisse = durchfließende Daten**

Dies unterscheidet sich deutlich von (gewohnten) Kontrollfluss-orientiertem Stil

Reaktive Komponenten – Aktoren

Reaktive Komponenten und Aktoren:

- Aktoren sind ein Konzept das ursprünglich aus der Theorie der verteilten Systeme kommt, dem Aktor-Modell
- Sie wurden als
  - konkretes Konzept reaktiver Komponenten
  - in das Arsenal der Programmier-Abstraktionen
  - auch für nebenläufige Systeme übernommen
- Aktoren verwischen die Grenzen zwischen Verteilt und Nebenläufig

  Nebenläufige Anwendungen werden mit Mitteln bearbeitet, die für verteilte Anwendungen konzipiert wurden

Aktoren in Scala: Akka-Aktoren

Scala verwendet die Aktor-Implementierung von Akka

Akka-Aktoren via sbt:

```
libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.5.2"
```

(Akka-) Aktoren in Scala

Beispiel, Aktor: Aktor-Klasse + Aktor-Instanz

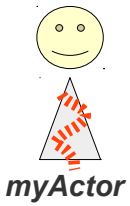
```
import akka.actor.{Actor, ActorSystem, Props}

class MyActor extends Actor {
  def receive = {
    case s: String => println("Actor received \"" + s + "\" from " + sender )
    case _ => println("Actor received unknown msg from " + sender )
  }
}


object Test_Actors {

  val system  = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name = "myActor")

  def main(args: Array[String]): Unit = {
    myActor ! "Hello actor"

    Thread.sleep(1000)

    system.terminate()
  }

}
```

Aktortyp MyActor als Ableitung von Actor

Die receive-Methode definiert das Verhalten als Reaktion auf empfangene Nachrichten.

Aktoren leben in einem ActorSystem

Aktor-Instanzen werden mit einer Fabrik-Methode actorOf erzeugt.



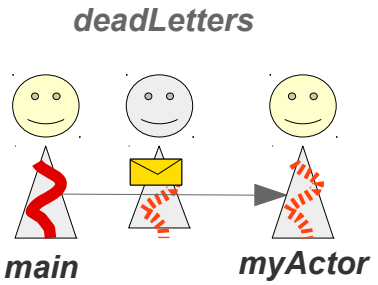
myActor

reagiert auf Nachrichten
- *vom Typ String mit der Ausgabe der Nachricht und ihres Senders*
- *auf andere Nachrichten mit der Angabe des Senders*

(Akka-) Aktoren in Scala

Beispiel, ein kommunizierende Aktoren:

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

class MyActor extends Actor {
  def receive = {
    case s: String => println("Actor received \"" + s + "\" from " + sender )
    case _ => println("Actor received unknown msg from " + sender )
  }
}


object Test_Actors {

  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name = "myActor")

  def main(args: Array[String]): Unit = {
    myActor ! "Hello actor"

    Thread.sleep(1000)

    system.terminate()
  }

}
```



Der Main-Thread sendet an den Aktor myActor.
Die Zustellung erfolgt indirekt via deadLetters (ein System-Aktor).

~> `Actor received "Hello actor" from Actor[akka://MySystem/deadLetters]`

Der* main-*Thread sendet, er ist kein* Actor, *darum wird als Absender* deadLetter *angegeben.

Basis-Komponenten

Actor

Instanz einer Aktor-Klasse

Reaktives Verhalten

Daten komplett gekapselt (kein Zugriff von außen)

Ausführung stets *single-theaded*

(In der Regel) kein dezidierter Thread zugeordnet

Ein gemeinsamer Zustand von Aktoren kann natürlich implementiert werden. Dies ist aber ein Missbrauch des Aktor-Konzepts.

ActorRef

Referenz auf eine Aktor-Instanz

Aktoren werden immer über eine Referenz angesprochen

ActorSystem

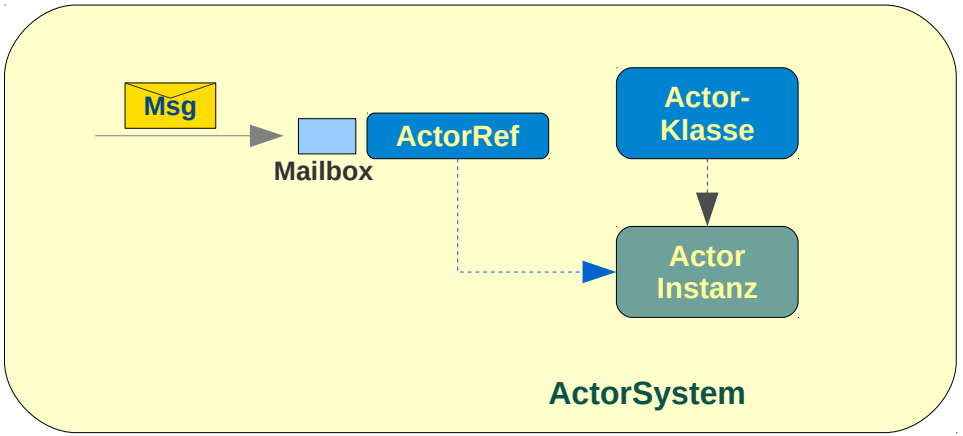
Mechanismus zum Ausführen der Aktor-Logik

vergleichbar mit Executor

enthält *Dispatcher*

Mailbox

Nachrichteneingang des Aktors



Ausführung

Mailbox

Nachrichten-Queue, üblicherweise pro Aktor-Instanz

Aktor Instanz

Aktor-Objekt in einem Zustand

Aktor-Klasse

Code der Nachrichtenverarbeitung

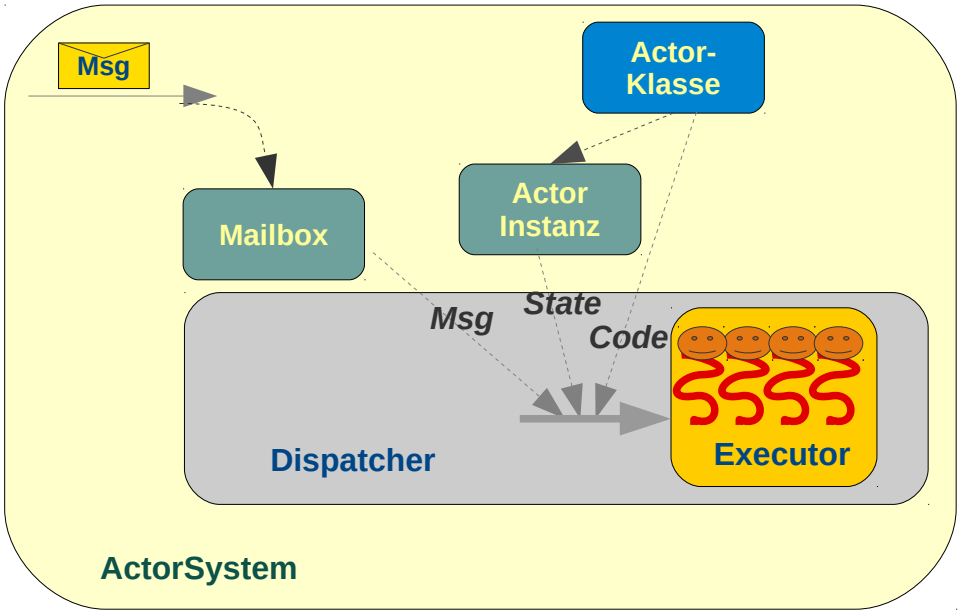
Dispatcher

Bringt Nachricht und Aktor zusammen:

Führt den der Nachrichtenverarbeitung aus

Nutzt dazu einen Executor

Dieser steht als ExecutionContext zur Verfügung – z.B. zur Ausführung von Futures

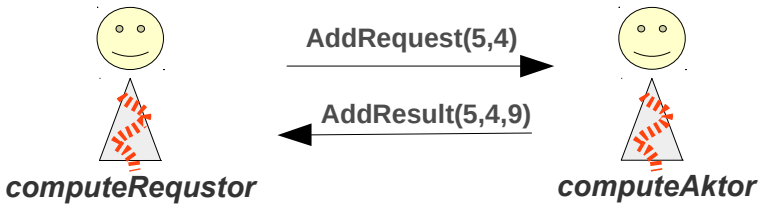


Aktoren: Senden und Empfangen

kommunizierende Aktoren

Beispiel:

- Ein Aktor, der addieren kann und
- ein Aktor der addieren lässt




```
abstract sealed class Msg
case object Go extends Msg
case class AddRequest(a1: Int, a2: Int) extends Msg
case class AddResult(a1: Int, a2: Int, v: Int) extends Msg
```

Nachrichten sollten stets unveränderliche Objekte sein. Am besten als Case-Klassen definiert

kommunizierende Aktoren

Beispiel:

- **Ein Aktor der einen anderen Aktor addieren lässt**

computeRequstor

```
class ComputeRequester(computeActor : ActorRef) extends Actor {
  def receive = {
    case Go      => computeActor ! AddRequest(5, 6)
    case AddResult(x, y, sum) => println(s"ComputeActor computed: $x + $y = $sum")
    case _       => println("Actor received unexpected msg")
  }
}
```

Ein ComputeRequester muss wissen, wer für ihn rechnet:
Eine Referenz auf den ComputeActor wird dazu als Konstruktor-Parameter übergeben.

- *Go: Nachricht vom Main-Thread, tue etwas.*
- *AddResult: die Antwort auf den Rechen-Auftrag.*
- *_: Etwas anders sollte nicht eintreffen.*

Aktoren: Senden und Empfangen

kommunizierende Aktoren

Beispiel:

- **Ein Aktor, der addieren kann**

computeActor

```
class ComputeActor extends Actor {
  def receive = {
    case AddRequest(v1, v2)     => sender ! AddResult(v1, v2, v1+v2)
    case _      => println("Actor received unexpected msg")
  }
}
```

Ein ComputeActor erfüllt Addier-Aufträge:

- ***AddRequest: Nachricht mit der Aufforderung zu addieren. Die Addition wird ausgeführt und die Antwort an den Absender der Nachricht geschickt.***
- ***_: Etwas anders sollte nicht eintreffen.***

Aktor-Erzeugung

Aktoren können auf zwei Arten erzeugt werden

- **aus Aktor-Klassen via Default-Konstruktor:**

  system.actorOf(Props[*Aktor-Klasse*], ...)

- **aus Aktor-Klassen via Konstruktor mit Parameter:**

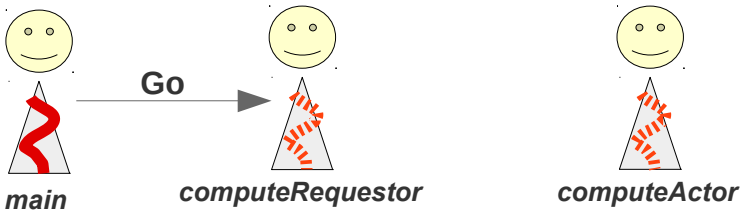
  system.actorOf(Props(classOf[*Aktor-Klasse*], argumente), ...)

classOf[A] ~ A.class in Java

Beispiel:

```
object ComputingActors {

  val system = ActorSystem("MySystem")

  val computeActor = system.actorOf(
    Props[ComputeActor], name = "computeActor"
  )

  val computeRequester = system.actorOf(
    Props(classOf[ComputeRequester], computeActor),
    name = "computeRequestor"
  )

  def main(args: Array[String]): Unit = {
    computeRequester ! Go

    Thread.sleep(1000)
    system.terminate()
  }

}
```

Konstruktor-Parameter




```
// Aktor mit default-Konstruktor
class ComputeActor extends Actor {
  ...
}

// Aktor mit parametrisiertem Konstruktor
class ComputeRequester(computeActor : ActorRef) extends Actor {
  ...
}
```

Aktor-Erzeugung

Props: Eine Klasse deren Objekte die Konfigurationsdaten für die Erzeugung von Aktoren enthalten

Die Erzeugung des Props-Objekts für eine Aktor-Klasse kann auch im Klassen-Objekt erfolgen

Beispiel:

```
class ComputeRequester(computeActor : ActorRef) extends Actor {
  def receive = {
    case Go     => computeActor ! AddRequest(5, 6)
    case AddResult(x, y, sum) => println(s"ComputeActor computed: $x + $y = $sum")
    case _      => println("Actor received unexpected msg")
  }
}

object ComputeRequester {
  def apply(computeActor : ActorRef): Props = Props(classOf[ComputeRequester], computeActor)
}
```

```
val computeRequester = system.actorOf(ComputeRequester(computeActor), "computeRequester")
```

Aktor-System

Aktoren gehören stets zu einem Aktor-System

Eine Anwendung hat typischerweise genau ein Aktor-System

Aktoren / Props in Aktoren

Aktor-Klassen dürfen nicht innerhalb von anderen Aktor-Klassen definiert werden

Props-Objekte dürfen nicht innerhalb von Aktor-Klassen erzeugt werden

Aktoren dürfen andere Aktoren erzeugen:

```
class Parent extends Actor {
  . . .

  val childActor =  context.actorOf(Props[ChildActor], name = "child")

  . . .
}
```

Aktor-Erzeugung durch einen Aktor: context *statt* system

Nachrichten Senden und Empfangen

Empfangen

- Aktoren können Nachrichten von jedem Typ empfangen
- Der Empfang erfolgt durch die receive-Methode,
  die üblicherweise mit Pattern-Matching über die Nachrichten definiert definiert ist
- Die Kommunikation ist asynchron

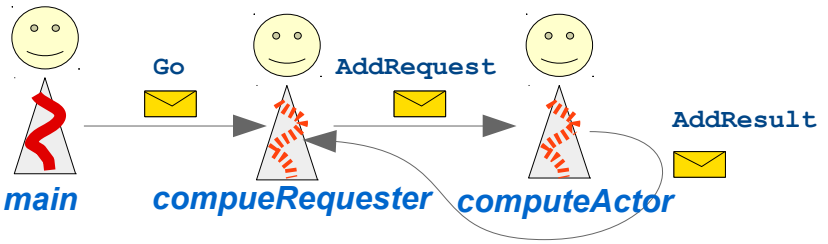
Senden

- Tell-Syntax: *<ActorRef-Empfänger>* ! *Msg*
  Die Nachricht wird in der Mailbox des Empfängers abgelegt
  Asynchrone Kommunikation
- Ask-Syntax: *<ActorRef-Empfänger>* ? *Msg*
  Die Nachricht wird in der Mailbox des Empfängers abgelegt und
  Der Sender erhält ein Future-Objekt mit der (zukünftigen) Antwort des Empfängers

Aktoren: Senden und Empfangen

Tell-Beispiel

Die Antwort auf den Auftrag wird nicht explizit abgewartet. Sie ist eine Nachricht wie jede andere.




```
case class AddRequest(x: Int, y: Int)
case class AddResult(x: Int, y: Int, z: Int)

class ComputeActor extends Actor {
  def receive = {
    case AddRequest(x, y) =>
      sender ! AddResult(x, y, x+y)
    case _ => println("Actor received unexpected msg")
  }
}

case object Go

class ComputeRequester(computeActor : ActorRef) extends Actor {
  def receive = {
    case Go =>
           computeActor ! AddRequest(5, 6)
    case AddResult(x, y, sum) =>
           println(s"ComputeActor computed: $x + $y = $sum")
    case _ => println("Actor received unexpected msg")
  }
}
```

Aktor rechnet
Antwort wird an sender gesendet.
Sender ist eine Methode der Actor-Klasse

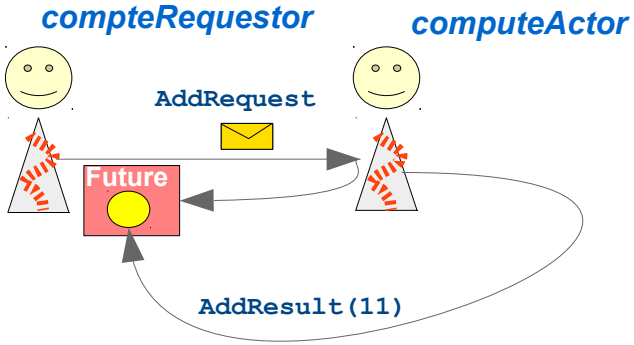
Aktor lässt rechnen

Die Antwort wird nicht abgewartet, sondern irgendwann empfangen

Ask-Beispiel: Nachricht erzeugt Antwort als Future-Objekt

```
import akka.actor.{Actor, ActorRef, ActorSystem, Props}
import akka.util.Timeout
import akka.pattern.ask
import scala.concurrent.duration._
import scala.util.{Failure, Success}

case class AddRequest(x: Int, y: Int)
case class AddResult(x: Int, y: Int, z: Int)
case object Go

class ComputeActor extends Actor {
  def receive = {
    case AddRequest(x, y) =>
      sender ! AddResult(x, y, x+y)
    case _ => println("Actor received unexpected msg")
  }
}

class ComputeRequester(computeActor : ActorRef) extends Actor {
  implicit val timeout = Timeout(1 seconds)
  implicit val execContext = context.dispatcher
  def receive = {
    case Go =>
      computeActor ! AddRequest(5, 6)
      val futureResult = computeActor ? AddRequest(40, 2)
      futureResult.onComplete{
        case Success(result) => println(s"Answer to ask is: $result")
        case Failure(failure) ⇒ println(s"Failed because of $failure")
      }
    case x => println(s"Actor received msg $x from $sender")
  }
}
```

```
Answer to ask is: AddResult(40,2,42)
Actor received msg AddResult(5,6,11) from Actor[akka://MySystem/user/computeActor#1430995469]
```



Aktor sendet die Antwort an* sender *mit tell-Syntax (!). Die Methode* sender *liefert den Sender der zuletzt empfangenen Nachricht.

Die Anfrage wird mit der Ask-Syntax (?) gestellt.
Dazu muss es ein Timeout definiert sein.

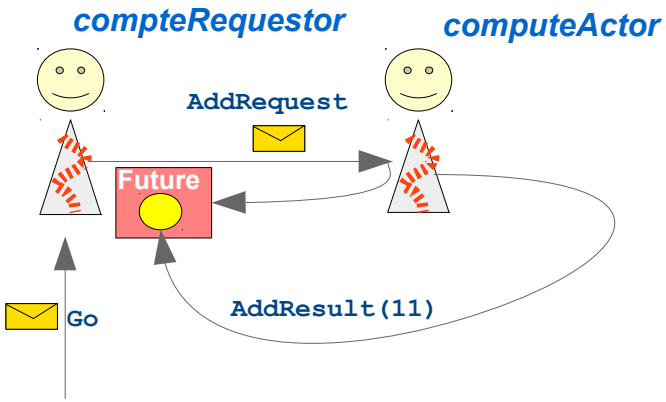
Da das Ergebnis ein Future-Objekt ist, kann es asynchron von einem Executor verarbeitet werden.

Jedes Aktor-System hat mit seinem Dispatcher einen passenden Executor.

Jeder Actor hat einen contex über den er auf den Dispatcher zugreifen kann.

Ask-Beispiel (fortgesetzt): Nachricht erzeugt Antwort als Future-Objekt

```
object Ask_Example {
  val system = ActorSystem("MySystem")

  val computeActor = system.actorOf(
    Props[ComputeActor], name = "computeActor"
  )

  val computeRequester = system.actorOf(
    Props(classOf[ComputeRequester], computeActor),
    name = "computeRequestor"
  )

  def main(args: Array[String]): Unit = {

    computeRequester ! Go

    Thread.sleep(1000)
    system.terminate()
  }

}
```




```
Answer to ask is: AddResult(40,2,42)
Actor received msg AddResult(5,6,11) from Actor[akka://MySystem/user/computeActor#1430995469]
```

Lang andauernde Berechnungen in Aktoren

sind prinzipiell möglich, sollten aber vermieden werden.

```
class ComputeActor extends Actor {
  def receive = {
    case Request(v) => sender ! ... result of some long running computation ...
    case _ => ...
  }
}
```

so nicht !

```
class ComputeActor extends Actor {
  implicit val execContext = context.dispatcher
  def receive = {
    case Request(v) =>
     ((replyTo: ActorRef) => Future {
                 replyTo ! ... result of some long running computation ...
           }) (sender)
    case _ => ...
  }
}
```

besser so.

execContext :
Der ThreadPool in dem das Future ausgeführt wird. Hier der Dispatcher des Aktor-Systems entnommen.

replyTo :
Der Sender wird vor der Ausführung der der asynchronen Aktion lokal gebunden damit er nicht von nachfolgenden Empfangs-Operationen überschrieben wird.

Ergebnisse langdauernder Berechnungen annehmen

Berechnungsergebnisse können in „normalen“ Empfangsoperationen im „Ruhezustand“ angenommen werden

```
class ComputeRequester(computeActor: ActorRef) extends Actor {
  def receive = {
    case Go => (1 to 10) foreach { i => computeActor ! Request(i) }
    case Response(value) => println(s"Result is $value")
    case ...
  }
}
```

Möglich, aber der Bezug zum Auftrag ist verloren

```
class ComputeRequester(computeActor : ActorRef) extends Actor {
  implicit val timeout = Timeout(1 seconds)
  implicit val execContext = context.dispatcher
  def receive = {
    case Go =>
      (1 to 10) foreach { i =>
        val futAnswer = computeActor ? AddRequest(i, 10*i)
        futAnswer.onComplete {
          case Success(v) => println(s"result for $i: $v")
          case Failure(t) => println(t)
        }
      }
    case x => println(s"Actor received msg $x from $sender")
  }
}
```

Besser beim Auftrag eine asynchrone Verarbeitung der Antwort in Auftrag geben.

Aktoren als leichtgewichtige Threads

Parallele Ausführung von Divide-and-Conquer-Algorithmen

Die parallele Ausführung von *Divide-and-Conquer*-Algorithmen (Fibonacci) in einem Thread-Pool ist problematisch:

- Viele Threads müssen erzeugt werden
- Die meiste warten auf Ergebnisse von „Sub-Threads“
- Der Pool ist schnell erschöpft

Lösung mit Aktoren

- Mit Aktoren kann das Problem behoben werden, da eine nahezu unbegrenzte Zahl wartender Aktoren möglich ist

Lösung mit ForkJoin / Recursive-Task

- Besser handhabbare / schnellere spezialisierte Lösungen

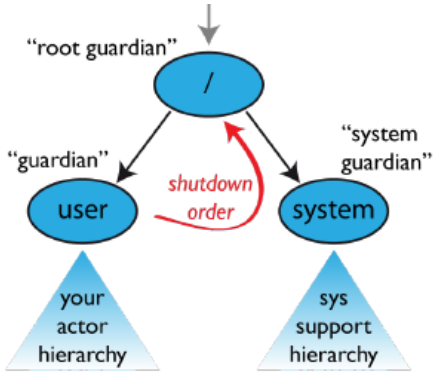
Aktor-System

Aktoren sind hierarchisch angeordnet

- Jeder (bis auf einen) Aktor hat einen Aktor der ihn erzeugt hat
- Jeder Aktor kann andere Aktoren erzeugen – seine Abkömmlinge / Kinder
- Aktoren überwachen und kontrollieren ihre Abkömmlinge
- Ein Aktor-System hat stets drei *Guardian*-Aktoren
  - **Root Guardian** Actor
    erzeugt und beaufsichtigt die beiden anderen
  - **(User) Guardian** Actor
    erzeugt und beaufsichtigt alle Aktoren des Benutzers
  - **System Guardian** Actor
    für System-Aufgaben

Ein Aktor-System

- muss einen eindeutigen Namen haben
- bietet Zugriff auf all seine Aktoren zum
  - Senden von Nachrichten
  - Stoppen von Aktoren
- kann konfiguriert werden
- kann gestoppt werden



aus der Akka-Dokumentation

Pfad

Identifikation eines Aktors durch eine „Aktor-URL“

Jeder (Benutzer-) Aktor kann durch einen String URL-Format:

`akka://actorSystem/user/aktorName-1/../aktorName-N`

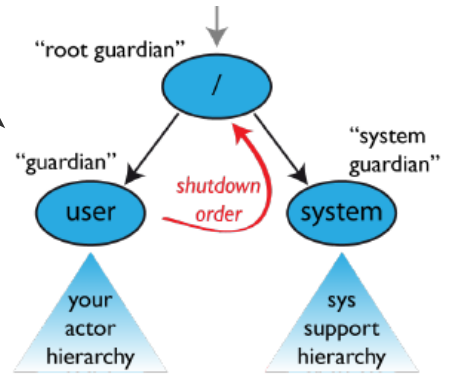
adressiert werden. Beispiel:

```
system.actorSelection("akka://MySystem/user/master/slave_1") ! "Hi"
```

Aktor mit dem Pfad `akka://MySystem/user/master/slave_1`

Dieser Aktor ist

- Ein **user**-Aktor (kein System-Aktor)
- im System mit dem Namen `MySystem`
- Er wird beaufsichtigt vom Aktor mit dem Pfad `akka://MySystem/user/master`
- und heißt `slave_1` .



Aktoren / Pfad

Pfad Beispiel:

```
object ActorPathEx_Main extends App {
  val system = ActorSystem("MySystem")
  val master = system.actorOf(Props[Master], name = "master")

  (0 until 3).foreach(master ! CreateSlave(_) )

  master ! "Hello Slave?" // to slaves via master

  (0 until 3).foreach( // to slaves directly
      i => system.actorSelection("akka://MySystem/user/master/slave_"+i) ! "Hi"
  )
}
```

Aktor-Erzeugung im system.

Aktor-Identifizierung via Aktor-Referenz

actorSelection: Aktor-Identifizierung via Pfad

```
case class CreateSlave(i: Int)

class Slave(i: Int) extends Actor {
  def receive = {
    case s:String => println(self + " received " + s)
    case _ => println(self + " received unknown msg")
  }
}
object Slave {
  def props(nr : Int): Props = Props(classOf[Slave], nr)
}

class Master extends Actor {
  var slave : Array[Option[ActorRef]] = Array(None, None, None)
  def receive = {
    case CreateSlave(i) => slave(i) = Some(context.actorOf(Slave.props(i), name = "slave_"+i))
    case msg: String => // forward msg to slaves
        slave.foreach(_.get ! msg)
  }
}
```

Aktor-Erzeugung im context.

Context

Jeder Aktor hat einen zugeordneten Context

Aktor-Erzeugung — In diesem *Context* können Aktoren erzeugt werden

Zugriff auf das Aktor-System — und auf das System zugegriffen werden

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

case object Msg

class AnActor extends Actor {
   def receive = {
    case Msg =>
      println ("I'm part of system: " + context.system + " my path is " + self.path)
      val a1 = context.system.actorOf(Props[AnActor], name= "AA1") // erzeugt akka://MySystem/user/AA1
      val a2 = context.actorOf(Props[AnActor], name= "AA2")        // erzeugt akka://MySystem/user/A/AA2
      a1 ! "Blub"     // -> akka://MySystem/user/AA1
      a2 ! "Blubber"  // -> akka://MySystem/user/A/AA2

    case m:String => println("Actor " + self.path + " received msg " + m + " from " + sender )
  }
}

object ActorContexEx_Main extends App {
  val system = ActorSystem("MySystem")
  val a = system.actorOf(Props[AnActor], name= "A")
  a ! Msg
  Thread.sleep(1000)
  system.shutdown
}
```

```
I'm part of system: akka://MySystem my path is akka://MySystem/user/A
Actor akka://MySystem/user/A/AA2 received msg Blubber from Actor[akka://MySystem/user/A#-
1418745665]
Actor akka://MySystem/user/AA1 received msg Blub from Actor[akka://MySystem/user/A#-1418745665]
```

Zugriff auf verwandte Aktoren

Im Context können Aktoren auf ihre Eltern und Kinder zugreifen

```
class AnotherActor extends Actor {

   def receive = {
    case Msg =>
      val a1 = context.actorOf(Props[AnActor], name= "A1")
      val a2 = context.actorOf(Props[AnActor], name= "A2")

      for (c <- context.children) {
        println("Child " + c.path)
      }

      println("Parent: " + context.parent.path)

    case m:String => println("Actor " + self.path + " received msg " + m + " from " + sender )
  }

}
```

Actor-DSL Aktoren können mit der Aktor-DSL erzeugt und gestartet werden

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

object ActorDslEx_Main extends App {

  val system = ActorSystem("MySystem")

  val my_actor = actor(system)(
    new Act {
      become {
        case s:String =>
           println("my_actor received "+s+" from "+sender)
      }
    })

  my_actor ! "Hello"
}
```

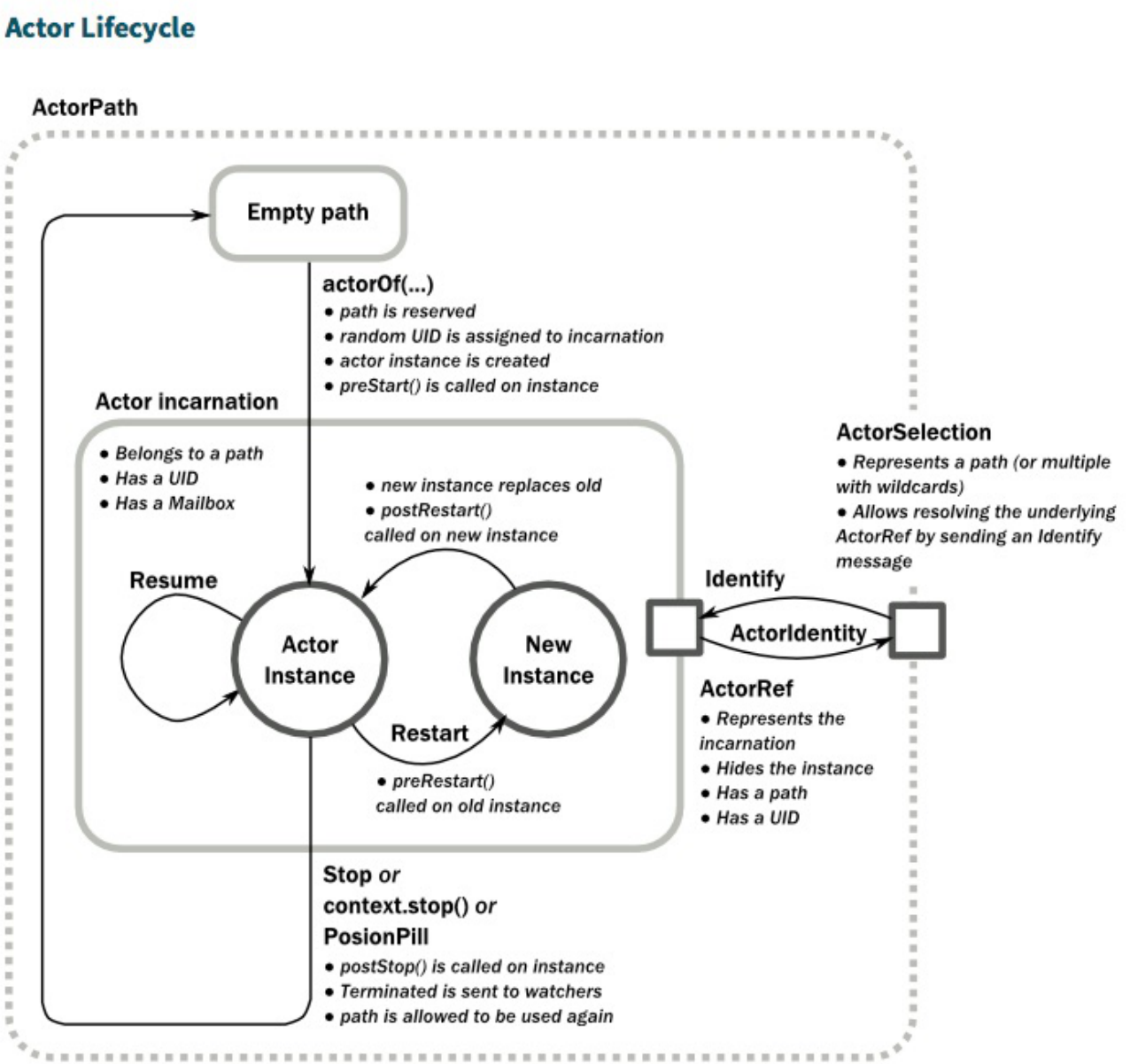
`become` : *das ursprünglich leere Verhalten des Basis-Aktors wird durch ein neues Verhalten ersetzt.*

Noch hübscher mit `system` *als implizitem Parameter*

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

object ActorDslEx_Main extends App {

  implicit val system = ActorSystem("MySystem")

  val my_actor = actor(
    new Act {
      become {
        case s:String =>
           println("my_actor received "+s+" from "+sender)
      }
    })

  my_actor ! "Hello"
}
```

Lebenszyklus

siehe Akka-Doku



Erzeugung ~> Start

Aktoren werden bei ihrer Erzeugung sofort gestartet. Eine besondere Start-Anweisung ist unnötig.

Varianten der Erzeugung:

- Im System-Kontext:

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[MyActor], name = "myActor")
```

- Im System-Kontext mit Konstruktor-Parametern:

```
val system = ActorSystem("HelloSystem")
val myActor = system.actorOf(Props(classOf[MyActor],x), name = "myActor")
```

- Innerhalb eines Aktors:

```
context.actorOf(Props[MyActor], name = "myActor")  bzw.:
context.actorOf(Props(classOf[MyActor],x), name = "myActor")
```

Start- / Stop-Hooks

preStart / whenSarting

wird vor dem Start des Aktors aufgerufen

postStop / whenStopping

wird nach dem Stopp aktiviert

```
class HookedActor extends Actor {

  override def preStart {
    println("I'm going to be started")
  }
  override def postStop {
    println("I was stoped")
  }
  def receive = {
    case _ => println("I've got a msg")
  }
}


object HooksEx_Main extends App {
  implicit val system = ActorSystem("MySystem")

  val actor_1 = system.actorOf(Props[HookedActor], name = "actor_1")

  val actor_2 = actor(new Act {
      whenStarting {
            println("I'm going to be started - DSL version")
      }
      whenStopping {
            println("I was stopped - DSL version")
      }
      become {
            case _ => println("I've got a msg")
      }
  })

  system.shutdown
}
```

```
I'm going to be started
I'm going to be started - DSL version
I was stoped
I was stopped - DSL version
```

Akka-Aktoren / Stoppen, Beenden

Stoppen von Aktoren: Aktoren können mit Giftpillen und der Stop-Methode gestoppt werden

System beenden: Das gesamte Aktorsystem kann mit shutdown herunter gefahren werden

Actor 1 „tötet“ Aktor 2 mit einer Giftpille

Actor 1 wird vom main-Thread gestoppt

```
import akka.actor.ActorDSL._
import akka.actor.PoisonPill

object PoisonEx_Main extends App {
  implicit val system = ActorSystem("MySystem")

  val actor_1 = actor(system)(new Act {
     override def postStop() {
           println("actor 1 got stopped")
     }
     whenStarting {
           actor_2 ! "Hello"
           actor_2 ! PoisonPill
           actor_2 ! "Still alive ?"
     }
  })

  val actor_2 = actor(new Act {
  override def postStop() {
    println("actor 2 got stopped")
  }
  become {
    case s:String =>
     println("actor 2 received " + s + " from " + sender)
  }
  })

  Thread.sleep(1000)

  println("main stops actor 1")
  system.stop(actor_1)

  system.shutdown
}
```

Stoppen von Aktoren: (1)
Graceful Stop

```
import akka.actor.{Actor, ActorSystem, Props, ActorRef, ActorSelection, Terminated}
import akka.actor.ActorDSL._
import akka.pattern.gracefulStop
import scala.concurrent.{Await, Future}
import akka.util.Timeout
import scala.concurrent.duration._

class Helper(i: Int) extends Actor {
  def receive = {
    case s:String => println(self + " received " + s)
  }
}

// worker with 3 helpers
// helper 1 will be stopped by main
class Worker extends Actor {

  var helper : Array[Option[ActorRef]] = Array(None, None, None)

  override def preStart() {
      for ( i <- 0 until 3 ) {
        helper(i) = Some(context.actorOf(Props(classOf[Helper], i), name = "helper_"+i))
      }
  }

  def receive = {
    case msg: String =>
      for (h <- context.children) {
        h ! msg
      }
  }
}
```

Der Aktor Helper 1 wird „mit Anmut“ gestoppt werden.

Der Aktor Worker erzeugt 3 Helper.

Alle Nachrichten werden an die Helper weiter geleitet.

Stoppen von Aktoren: (2)
Graceful Stop

```
object GracefulStopEx_Main extends App {
  val system = ActorSystem("MySystem")
  val worker = system.actorOf(Props[Worker], name = "worker")

  worker ! "Hi"
  Thread.sleep(100)

  val actorSel : ActorSelection = system.actorSelection("akka://MySystem/user/worker/helper_1")

  implicit val timeout = Timeout(1 seconds)
  implicit val execContext = system.dispatcher

  // Get reference to helper 1 from selection:
  val actorRefFuture : Future[ActorRef] = actorSel.resolveOne()
  val actorRef : ActorRef = Await.result(actorRefFuture, 1 seconds)

  // stop helper 1 gracefully
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds)
  Await.result(stopped, 6 seconds)
  println(if (stopped.value.get.get) "Helper 1 was stopped" else "Helper 1 was not stoped")

  Thread.sleep(100)
  worker ! "How are you?"
  Thread.sleep(1000)
  system.shutdown

}
```

```
Actor[akka://MySystem/user/worker/helper_1#215447887] received Hi
Actor[akka://MySystem/user/worker/helper_2#2045892062] received Hi
Actor[akka://MySystem/user/worker/helper_0#439241858] received Hi
Helper 1 was stopped
Actor[akka://MySystem/user/worker/helper_0#439241858] received How are you?
Actor[akka://MySystem/user/worker/helper_2#2045892062] received How are you?
```

Dokumentation

Weitere Informationen

entnehme man der ausgezeichneten Akka-Dokumentation:

http://doc.akka.io/docs/akka/current/scala/actors.html