Java Actoren

Contents

Akka

Wichtige Momente

Aktorsystem estellen

```
ActorSystem actorSystem = ActorSystem.create("TeaseLisa");
```

Aktor Class erstllen

extends **AbstractActor**, implementiere methode **public Receive createReceive()**

```
public class PingPong extends AbstractActor {
    ActorRef partner;
    String name;

    public PingPong(ActorRef sendTo, String name) {
        this.partner = sendTo;
        this.name = name;
    }

@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(Integer.class, message -> beat(message))
        .build();
}
```

Aktor ohne Parameter erstellen

```
ActorRef lisa = actorSystem.actorOf(Props.create(Kid.class));
```

Aktor mit Parameter erstellen

```
ActorRef philosoher = actorSystem.actorOf(
    Props.create(Philosopher.class, table, name));
```

Struktur: Props.create(class, args);

Child Actor mit context() erstellen

Man verwendet getContext().getSystem(), um actorSystem zu finden

```
public class Kid extends AbstractActor {
  private int cursed = 0;
  private ActorRef mommy;

  public Kid()  {
    mommy = getContext().getSystem()
             .actorOf(Props.create(Mommy.class));
  }
```

Aktor createReceive() example

Falls message ein String ist und message==“printHello“ ist, wird „Hello World!“ gedruckt
Otherwise unhandled(message)

A HelloWorld Actor

- An actor that prints “Hello World” in response to a message with the value "printHello" can be implemented as follows:

```
public class HelloWorldActor extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(String.class,
                message -> message.equals("printHello"),
                message -> System.out.println("Hello World!"))
            .matchAny(message -> unhandled(message))
            .build();
    }
}
```

This generic match can also be omitted, as it is automatically executed

match()

Nur dann, wenn class passt und bedingung erfüllt (optional)
match(class, bedingung, aktion);
match(class, aktion);

matchAny()

Beliebige message.
mathcAny(aktion);

unhandled()

Mache nichts, default case

tell()

sender sends a message to actor:

```
actor.tell(message, sender);
```

```
ping.tell(0, ActorRef.noSender());
```

message ist 0, sender() in ping gibt Null-Sender zurück

```
ping.tell("abc", pong);
```

message ist „abc“, sender() in pong gibt ActorRef von pong zurück

```
ping.tell("abc", self());
```

getSender() gibt ActorRef von current object.

sender()

Mit sender kann man ActorRef auf Sender vom Message bekommen, falls es beim tell() gesetzt wurde, ActorRef.noSender, falls es nicht gesetzt wurde.

```
if (partner != null) {
    partner.tell(message, self());
} else {
    sender().tell(message, self());
}
```

self()

“”returns ActorRef of **current Actor**””
Kann man für tell verwenden

Alle Objekte sind Aktoren

Speischende Philosopher: Alle Philosopher sind Aktoren, und Tisch ist auch ein Aktor

Folien

Akka Actors

- Akka provides an implementation of the actor model for JVM-based languages [Akka]
  - Akka is implemented in Scala, which is why many required classes are defined in the Scala namespace
  - Akka can also be used in Java
- All classes on the following slides are defined in one of the following packages:
  - `akka.actor`
  - `akka.util`
  - `scala.concurrent`

Further Methods of an Actor

- Actors can override further methods, which are executed if the actor state changes, especially –
  - `preStart()`: Executed before an actor is started
  - `postStop()`: Executed after an actor is stopped
  - `preRestart()` / `postRestart()`: Executed before/after an actor is restarted
- Actors provide three important methods:
  - `getSelf()` delivers a reference (`ActorRef`) to the actor
  - `getContext()` delivers an `ActorContext`, which is especially an `ActorRefFactory` (for creating new actors)
  - `getSender()` delivers a reference to the actor sending the currently processed message

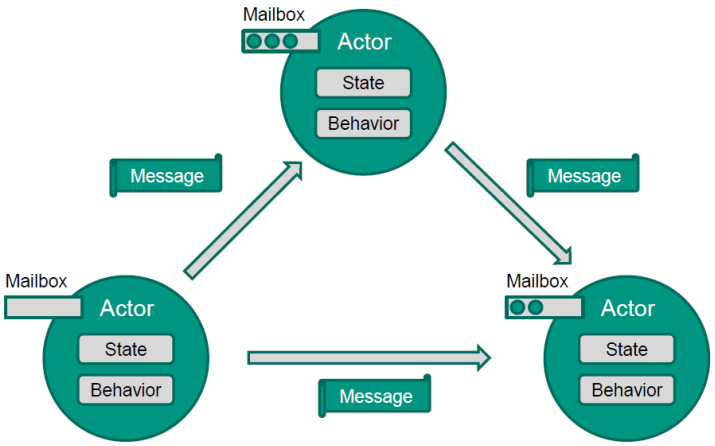
A HelloWorld Actor

- An actor that prints “Hello World” in response to a message with the value "`printHello`" can be implemented as follows:

```
public class HelloWorldActor extends AbstractActor {
   @Override
   public Receive createReceive() {
      return receiveBuilder()
         .match(String.class,
            message -> message.equals("printHello"),
            message -> System.out.println("Hello World!"))
         .matchAny(message -> unhandled(message))
         .build();
   }
}
```

This generic match can also be omitted, as it is automatically executed

Actors and Messages



Defining an Akka Actor

- An Akka actor has to extend the class `AbstractActor` and must at least implement the method:

```
Receive createReceive();
```

  - Is called if a message is sent to the actor
  - Has to define how a `Receive` object is created after receiving a message
- The method receiveBuilder() returns a pre-implemented builder for `Receive` objects
  - Provides several methods to match a received message
  - `match(Class<P> type, UnitApply<P> apply)` allows to handle messages of the specified `type` by the defined `apply` method (a functional interface with a method expecting an object of type `P`)
  - `match(Class<P> type, TypedPredicate<P> predicate, UnitApply<P> apply)` works like above but also checks the predicate
  - `matchAny(UnitApply<Object> apply)` handles messages of any type by the specified `apply` method

Messages

- A Message is delivered using an address, sometimes called “mailing address”
  - An actor can only communicate with actors whose addresses it has
  - An actor knows an address either through a received message or because it created the other actor itself
- Messages are sent asynchronously and stored in a mailbox of the receiving actor
- Messages are processed sequentially
  - An actor can only process one received message at a time
  - An actor processes messages in the order in which they arrived
  - Processing *n* message concurrently requires *n* actors

Actor Model

- Parallel computation with threads/shared memory –
  - requires locks to avoid race conditions
  - can easily lead to deadlocks when using locks wrong
  - uses blocking method calls

- The actor model is a conceptual, computational model for concurrent computation without locks and with asynchronous calls
  - Originally proposed by Hewitt, Bishop and Steiger in 1973 [Hewitt1973]
  - Refined by Gul Agha [Agha1986]
- The model is based on actors and messages
  - Actors are computation units
  - Actors communicate via messages → message passing

Actors

- Basic philosophy: *Everything is an actor*
- After receiving a message, an actor can –
  - send (a finite numbers of) messages to other actors
  - instantiate (a finite numbers of) new actors
  - designate the behavior when receiving the next message
    - which means that the actor can make computations and modify its local state, which influences what it does when receiving the next message
- Actors cannot access and modify the local state of other actors
- Actors keep the mutable state internal and communicate only via messages

Creating an Actor

- Actors are created using so called `Props`
  - Configuration class for specifying options for an actor creation
  - A `Props` for creating an actor of the given type can be instantiated with:

```
Props Props.create(Class<?> actorType, Object… parameters)
```

  - Props can also configure constructor parameters etc.
- An actor can be created using an `ActorRefFactory`'s method:

```
ActorRef actorOf(Props props)
```

  - Such a factory is implemented –
    - by the `ActorSystem`, which has to be instantiated once per application
    - by the actor context
- `Props`-based instead of ordinary constructor- or factory-based Actor instantiation is used due to different reasons:
  - Ensures that Actors only exists within an `ActorSystem`
  - Returns an `ActorRef` rather than an `Actor`, making it impossible to manually call Actor methods not using messages → less error-prone

Aufgaben

SS17

```java
public class Worker extends UntypedActor {
    @Override
    public void onReceive(Object query) {
        // Handle query
    }
}

public class Balancer extends UntypedActor {
    private final int workerCount;
    private List<ActorRef> worker;

    private int nextWorker;

    public Balancer(int workerCount) {
        if (workerCount < 1) {
            throw new IllegalArgumentException();
        }
        this.workerCount = workerCount;
    }

    private ActorRef createWorker() {
        return getContext().actorOf(Props.create(Worker.class));
    }

    @Override
    public void preStart() throws Exception {
        super.preStart();
        this.worker = new ArrayList<>();

        this.nextWorker = 0;
        for (int i = 0; i < workerCount; i++) {
            this.worker.add(createWorker());
        }

    }

    @Override
    public void onReceive(Object query) {

        this.worker.get(nextWorker).tell(query, getSender());
        nextWorker = (nextWorker + 1) % workerCount;

    }
}
```

(a) Ergänzen Sie die Klasse `Balancer` durch Ausfüllen der Lücken im gegebenen Quellcode, sodass in `preStart` die `Worker`-Instanzen angelegt werden und in `onReceive` die Nachricht zu einer der `Worker`-Instanzen weitergeleitet wird. Die Anfragen sollen dabei auf alle `Worker` gleichverteilt werden (beispielsweise durch zyklische Zuweisung). [4,5 Punkte]
`Hinweis:` Sie dürfen auch außerhalb der Methoden Ergänzungen an der Klasse vornehmen.

Beispiellösung: Die Lösung befindet sich innerhalb der Implementierung. Alternativ kann die Nachricht auch via `forward` weitergeleitet werden.

(b) Gegeben seien folgende Voraussetzungen: [3,5 Punkte]

- Die Abarbeitung einer Anfrage durch einen `Worker` benötigt viermal soviel Zeit, wie das Verteilen einer Anfrage durch den `Balancer` auf einen `Worker`.
- Eine beliebig hohe Last (Anzahl von Anfragen pro Zeiteinheit) liegt an.

Betrachten Sie folgende zwei Szenarien:

1. Es gibt einen `Worker` (`workerCount == 1`) und nur einen (Ein-Kern-)Prozessor, auf dem sowohl der `Balancer` als auch der `Worker` ausgeführt werden.
2. Es gibt beliebig viele `Worker` (`workerCount` beliebig hoch) und beliebig viele Prozessoren, auf die die `Worker` und der `Balancer` verteilt werden.

Wie groß ist der Unterschied im Durchsatz der Anwendung (verarbeitete Anfragen pro Zeiteinheit) zwischen den beiden Szenarien? Erläutern Sie Ihre Antwort.

Beispiellösung: Jede Anfrage besteht aus einem durch den `Balancer` auszuführenden Teil und einem an einen `Worker` zugewiesen Teil im Verhältnis $\frac{1}{5}$ zu $\frac{4}{5}$. Im ersten Szenario wird das Programm vollständig sequentiell ausgeführt. Im zweiten Szenario, bei der Bearbeitung einer beliebig hohen Anzahl von Anfragen durch beliebig viele `Worker` sind die den Arbeitern zugewiesenen Teile parallel ausführbar, während nur die Verteilung sequentiell vom `Balancer` ausgeführt wird. Es lässt sich somit Amdahls Gesetz anwenden, aus welchem sich folgender maximaler Speedup ergibt: $S = \frac{T(1)}{T(n \to \text{inf})} = \frac{1}{1-\frac{4}{5}} = 5$
Der Durchsatz der Anwendung ist im zweiten Szenario dementsprechend fünfmal so hoch wie im ersten.

Zyklische Verteilung von Queries

1) `int counter Field in Balancer definieren`

2) `onReceive:`

```
ActorRef current = this.worker.get(counter % workerCount);
current.tell(query, self());
counter++;
```