Actors - Typed Overview

Typed Actors

- Why can’t an Actor be more like an Object?
  - Why do we have to send messages to Actors?
  - Why does the Actor have to be written as an event loop?
  - Why can’t we use call / return syntax?
- Well, with **Typed Actors** we can!
  - Typed actors are defined by a Java interface & implementation.
  - When created, work as a standard object in both client *and* provider.
    - Client gets a proxy (also an actor) for the actor of the interface type.
    - Proxy marshalls arguments and sends request to “service actor.”
    - Service actor responds to onReceive by unmarshalling arguments.
    - Service actor calls the specified method.
    - If non-void, marshalls results and responds to the proxy.
    - Proxy returns to the client.

Why Typed Actors?

- Actors are nice for bridging between actor systems (the “inside”) and non-actor code (the “outside”), because they allow you to write normal OO-looking code on the outside.
- `TypedActor` has officially been deprecated by Akka, however, it was replaced by an Actor based system that allows for types (`typed.Actor`).
  - Instead of everything being a generic `Object` that is passed around, as in `AbstractUntypedActor`, messages can be any class.
- The new Actor system eliminates a lot of the issues associated with the older `TypedActor`, making it a viable option for most systems that would be a good fit for actors.

Actor (Behavior)

- Actor based systems are predominantly composed of two major components.
  - Message, which is used to encapsulate system state
  - Actor, which is used to encapsulate the changes to the system state, AKA the systems behavior
- In the AKKA system, messages are a class, which are effectively immutable
- `Behavior`s describe an action that is performed on a message
  - Each Actor has only one behavior
  - The same behavior can be used by many actors

Messages

```java
public abstract class Message {
    private final String value;
    private final ActorRef<Message> replyTo ;

    public Message (String value,
                    ActorRef<Message> replyTo) {
        this.value = value;
        this.replyTo = replyTo;
    }

    public Message (String value) {
        this (value, null);
    }

    public String getValue() {
        return value;
    }

    public ActorRef<Message> getReplyTo() {
        return replyTo;
    }
}
```

```java
public class StartMessage extends Message {
  public StartMessage () {
    super ("Start");
  }
}

public class StatusMessage extends Message {
  public StatusMessage (String value) {
    super (value);
  }
}

public class StopMessage extends Message {
  public StopMessage (ActorRef<Message> replyTo){
      super ("Stop", replyTo);
  }

  public StopMessage () {
    super ("Stop", null);
  }
}
```

Defining a Behavior

```
class First extends AbstractBehavior<Message> {

    static Behavior<Message> create () {
      return Behaviors.setup (First::new);
    }

    private First (ActorContext<Message> context) {
      super (context);
    }
}
```

Factory Method

Constructor

Notice the constructor is private

What To Do When Receiving a Message

```
class First extends AbstractBehavior<Message> {
```

Create a receiver for all messages

```
    @Override
    public Receive<Message> createReceive() {
      return newReceiveBuilder().onMessage(StartMessage.class, this::start).build();
    }
```

Describe behavior for a specific message

```
    private Behavior<Message> start (Message m) {
      System.out.println ("First: " + m.getValue());
      return Behaviors.stopped();
    }
}
```

Behaviors are stream based, so to end the stream, return a stopped Behavior

Classname.class is a generic way to access a typed reference to a class instance

Creating an Actor

```
public class Main {
    public static void main (String[] args) {
```

Create an Actor and assign it a Behavior

```
      ActorRef<Message> firstRef = ActorSystem.create(First.create(), “First");
```

Send the actor a message, using our old friend `tell`

```
      firstRef.tell(new StartMessage());
    }
}
```

A Single Actor Isn’t Very Useful

```
class First extends AbstractBehavior<Message> {

    private Behavior<Message> start (Message m) {
```

Create a second actor upon receiving the start message

```
        ActorRef<Message> secondRef = getContext().spawn(Second.create(), “Second");
        System.out.println("First: " + m.getValue ());
```

Send some message to the new actor

```
        for (int i = 0; i < 10; i++) {
          secondRef.tell (new StatusMessage ("Message #" + i));
        }
```

Don’t stop this time, instead keep the stream open

```
        return this;
    }
}
```

Handling Multiple Messages

```
class First extends AbstractBehavior<Message> {
```

Any number of messages can be added to the receiver

```
    @Override
    public Receive<Message> createReceive () {
      return newReceiveBuilder ().onMessage (StartMessage.class, this::start)
                                 .onMessage (StopMessage.class, this::stop).build();
    }
```

For our second message, we really do want to stop

```
    private Behavior<Message> stop (Message m) {
      System.out.println ("First: Stopping");
      return Behaviors.stopped ();
    }
}
```

Every possible message must be added to the Receive

Responding to Sender

```
class Second extends AbstractBehavior<Message> {
  static Behavior<Message> create() {
    return Behaviors.setup (Second::new);
  }
  private Second (ActorContext<Message> context) {
    super (context);
  }
  @Override
  public Receive<Message> createReceive () {
    return newReceiveBuilder ().onMessage (StatusMessage.class, this::printIt)
                               .onMessage (StopMessage.class, this::stop).build ();
  }
```

A Message must keep track of the sender in order to reply

```
  private Behavior<Message> stop (Message m) {
    System.out.println ("Second: Stoppping");
    m.getReplyTo ().tell (new StopMessage());
    return Behaviors.stopped ();
  }
}
```

Entire Demo

The next three slides are the entire demo program, minus package imports and specific messages

```
public abstract class Message {
    private final String value;
    private final ActorRef<Message> replyTo ;

    public Message (String value, ActorRef<Message> replyTo) {
        this.value = value;
        this.replyTo = replyTo;
    }

    public Message (String value) {
        this (value, null);
    }

    public String getValue() {
        return value;
    }

    public ActorRef<Message> getReplyTo() {
        return replyTo;
    }
}

public class Main {
    public static void main (String[] args) {
      ActorRef<Message> firstRef = ActorSystem.create (First.create (), “first-actor");
      firstRef.tell (new StartMessage ());
    }
}
```

First Actor’s Behavior

```
class First extends AbstractBehavior<Message> {
    static Behavior<Message> create () {
      return Behaviors.setup (First::new);
    }

    private First (ActorContext<Message> context) {
      super (context);
    }

    @Override
    public Receive<Message> createReceive () {
      return newReceiveBuilder ().onMessage (StartMessage.class, this::start)
                                 .onMessage (StopMessage.class, this::stop).build();
    }

    private Behavior<Message> start (Message m) {
      ActorRef<Message> secondRef = getContext().spawn(Second.create(), “second-actor");

      System.out.println ("First: " + m.getValue ());
      for (int i = 0; i < 10; i++) {
        secondRef.tell (new StatusMessage ("Message #" + i));
      }
      secondRef.tell (new StopMessage (getContext().getSelf()));
      return this;
    }

    private Behavior<Message> stop (Message m) {
      System.out.println ("First: Stopping");
      return Behaviors.stopped ();
    }
}
```

Second Actor’s Behavior

```
class Second extends AbstractBehavior<Message> {

  static Behavior<Message> create() {
    return Behaviors.setup (Second::new);
  }

  private Second (ActorContext<Message> context) {
    super (context);
  }

  @Override
  public Receive<Message> createReceive () {
    return newReceiveBuilder ().onMessage (StatusMessage.class, this::printIt)
                               .onMessage (StopMessage.class, this::stop).build ();
  }

  private Behavior<Message> printIt (Message m) {
    System.out.println ("Second: " + m.getValue ());
    return this;
  }

  private Behavior<Message> stop (Message m) {
    System.out.println ("Second: Stoppping");
    m.getReplyTo ().tell (new StopMessage());
  }     return this;

}
```