

Actors

Origins

- ▶ Hewitt, early 1970s (1973 paper)
- ▶ Around the same time as Smalltalk
- ▶ Concurrency plus Scheme
- ▶ Agha, early 1980s (1986 book)
- ▶ Erlang (Ericsson), 1990s
- ▶ Akka, 2010s

Actors: Way of Thinking

- ▶ Key idea: support for autonomy
 - ▶ Designed for concurrency
 - ▶ Equally good for distribution
- ▶ Shared nothing
 - ▶ Style of thinking
 - ▶ Architecture: no longer a single locus of control and storage
 - ▶ Programming: reacting to events propagated via messages
 - ▶ Encapsulate: eschew states, visibility of internal information, synchronization primitives (locks)
 - ▶ Forget threads as a programming abstraction—threads may implicitly do the work but not a good programming construct
 - ▶ Messaging and no shared memory
- ▶ Resource management
 - ▶ Start additional actors as needed for work and resources available
 - ▶ Stop actors when not needed
 - ▶ Migrate actors when needed, taking advantage of a universal actor reference
 - ▶ Handle exceptions through monitoring and supervision

Actor Basics

- ▶ An actor
 - ▶ Encapsulates local state (memory)—hence, the state may not be directly accessed from outside an actor
 - ▶ Encapsulates a thread
 - ▶ Mailbox (incoming)
 - ▶ Processed in order of arrival (e.g., as enqueued), though could be reordered (e.g., PriorityMailbox)
 - ▶ ActorRef: globally unique ID (serializable)
- ▶ To create an actor class in Akka,
 - ▶ Extend appropriate abstract class (or, in Scala, trait)
 - ▶ Define a `receive` method
 - ▶ Define corresponding `Props` configuration class

HelloAkkaJava.java (Messages)

```
public class HelloAkkaJava {
    //MPS: The first message is Greet; it has no parameters; its
    // expected outcome is to for the recipient to send a greeting
    public static class Greet implements Serializable {}

    //MPS: The second message is WhoToGreet; it has one parameter,
    // the target of the greeting; its expected outcome is for
    // the recipient to change the target
    public static class WhoToGreet implements Serializable {
        public final String who;
        public WhoToGreet(String who) {
            this.who = who;
        }
    }

    //MPS: The third message is Greeting; it has one parameter, the
    // greeting message; it is a message to be sent
    public static class Greeting implements Serializable {
        public final String message;
        public Greeting(String message) {
            this.message = message;
        }
    }
}
```

HelloAkkaJava.java (Actor)

```
public static class Greeter extends AbstractActor {
    String greeting = ""; //internal state of the actor

    @Override
    //MPS: Mapping of messages to behaviors. This snippet
    // handles the two incoming messages; it doesn't mention
    // the outgoing message (Greeting)
    public Receive createReceive() {
        return receiveBuilder()
            .match(WhoToGreet.class, this::onWhoToGreet)
            .match(Greet.class, this::onGreet)
            .build();
    }

    // MPS: Update internal state on receiving a WhoToGreet message
    private void onWhoToGreet(WhoToGreet whoToGreet) {
        greeting = "hello , " + whoToGreet.who;
    }

    // MPS: Send greeting message on receiving a Greet message
    private void onGreet(Greet greet) {
        // Send the current greeting back to the sender
        getSender().tell(new Greeting(greeting), getSelf());
    }
}
```

HelloAkkaJava.java (Main 1)

```
public static void main(String [] args) {
    try {
        // Create the helloAkka actor system and the greeter actor
        final ActorSystem system = ActorSystem.create("helloAkka");
        final ActorRef greeter =
            system.actorOf(Props.create(Greeter.class), "greeter");

        // MPS: The inbox (apparent misnomer) functions as an actor to
        // communicate with actors; sort of a "main" for actors to use
        // as a place for send and receive
        final Inbox inbox = Inbox.create(system);

        // Tell the greeter to change its 'greeting' message
        greeter.tell(new WhoToGreet("akka"), ActorRef.noSender());

        // Ask for the current greeting; reply to go to inbox
        inbox.send(greeter, new Greet());

        // Wait 5 seconds for the reply with the 'greeting' message
        final Greeting greeting1 = (Greeting)
            inbox.receive(Duration.create(5, TimeUnit.SECONDS));
        System.out.println("Greeting one: " + greeting1.message);
    }
}
```

HelloAkkaJava.java (Main 2)

```
// Initially after 0 seconds, send a Greet message every second to
// the greeter; Spoof sender as GreetPrinter (new Actor below)
final ActorRef greetPrinter =
    system.actorOf(Props.create(GreetPrinter.class));
system.scheduler().schedule(Duration.Zero(), Duration.create(1,
    TimeUnit.SECONDS), greeter, new Greet(),
    system.dispatcher(), greetPrinter);
} catch (TimeoutException ex) {
    System.out.println("Got a timeout waiting for reply from an
        actor");
    ex.printStackTrace();
}
}

public static class GreetPrinter extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(Greeting.class, (greeting) ->
                System.out.println(greeting.message))
            .build();
    }
}
}
```

The Receive Method

- ▶ A reaction rule for each type of message to be handled
 - ▶ In Akka, the set of rules must be *exhaustive* in that all other messages will publish an `UnhandledMessage` to the ActorSystem's `EventStream`
 - ▶ Best practice is to include a default rule (using `matchAny` in Java) for unexpected messages
- ▶ Good practice to
 - ▶ Separately describe the allowed message types, e.g., as static classes in Java
 - ▶ Write each message's handler as a separate little method
- ▶ An actor's `receive` method
 - ▶ A (partial) function object stored within the actor
- ▶ Hot swapping the `receive` method: Avoid unless essential
 - ▶ Changed through `context.become` method
 - ▶ Alternative: push new behavior and use `unbecome` to post

Messages

- ▶ Immutable objects
 - ▶ Not enforced by Java, so beware
- ▶ No delivery guarantees, pairwise FIFO—even unrelated messages!
 - ▶ May be lost
 - ▶ May be duplicated
 - ▶ Option to ensure at least once delivery
- ▶ Pairwise FIFO
 - ▶ If
 - ▶ An actor A sends two messages to actor B *and*
 - ▶ Both messages arrive
 - ▶ Then
 - ▶ They arrive in order
- ▶ Messages to same recipient from distinct actors are unrelated
 - ▶ May be arbitrarily interleaved
- ▶ Redesign to avoid reliance on FIFO
- ▶ If your application requires reliability of delivery
 - ▶ Verify it yourself: use acknowledgments
 - ▶ Achieve it yourself: use retries

Messages: Programming

- ▶ 「tell」
 - ▶ Asynchronous
 - ▶ Send message and return immediately
 - ▶ Preferable to maximize decoupling
- ▶ 「ask」
 - ▶ Asynchronous
 - ▶ Send message and return a 「Future」 what will contain the reply
 - ▶ Greater overhead in maintaining the context than for 「tell」

```
Timeout t = new Timeout(Duration.create(5, TimeUnit.SECONDS));  
CompletableFuture<Object> future2 = ask(actorB, "another request",  
t).toCompletableFuture();
```

- ▶ The 「CompletableFuture」 class supports joining futures, piping, and so on

ActorSystem

/	root (and its guardian or supervisor)
/user	user space (and its guardian or supervisor), called Guardian
/system	system space (and its guardian or supervisor)

- ▶ Any actors we create are under `[/user]`, although we create actors through
 - ▶ `system.actorOf`: children (called “top-level” actors) of `/user`
 - ▶ `context.actorOf`: their descendants (all levels)
- ▶ Every actor has a *parent* or *supervisor* in whose scope it is created
- ▶ Stopping an actor: recursively: children first; then self
 - ▶ `getContext().stop(child)`
 - ▶ `getContext().stop(getSelf())`
 - ▶ `PoisonPill` message to stop an actor in its tracks after the previously arrived (enqueued) messages are processed

Exceptions

- ▶ Current message
 - ▶ Already removed from mailbox and potentially lost
 - ▶ Unless explicit action to save the message or process it again
- ▶ Mailbox
 - ▶ Preserved: what's remaining after the current message was removed
 - ▶ Available to the restarted actor, if any
- ▶ Supervision: An exception throwing actor is suspended and control passed to its supervisor
- ▶ The supervisor decides the fate of the actor by applying a strategy
 - ▶ Resume: back to where it was when the exception occurred
 - ▶ Restart: reset its internal state to initial
 - ▶ Stop: end it
- ▶ Akka provides a rich set of hooks (methods) through which to customize behavior
 - ▶ Pre and post of the major events
 - ▶ Start, Stop, Restart
 - ▶ For example, `preRestart()`

Exceptions: Supervisor Strategy

- ▶ No call stack to pass an exception
 - ▶ Traditional idea doesn't work
 - ▶ Not clear which past or future caller should get the exception
- ▶ Therefore, pass to supervisor
 - ▶ Can apply its strategy
 - ▶ A couple of strategies are predefined
- ▶ One for One Strategy
 - ▶ If a child (supervisee) actor produces an exception, deal with that actor
- ▶ All for One Strategy
 - ▶ If the child actors are performing pieces of the same transaction and those not throwing an exception may be affected
- ▶ Predefined (fixed set of) directives on how to deal with a spoiled child
 - ▶ Resume
 - ▶ Stop
 - ▶ Restart
 - ▶ Escalate

Exception Handling: Throwing

```
class MadeUpException(msg: String) extends Exception(msg) {}
class JustBecauseException(msg: String) extends Exception(msg) {}

class SuperviseeActor(id: Int) extends Actor with ActorLogging {
  ...
  override def receive: Receive = {
    case SuperviseeActor.Fail =>
      log.info(s"$self fails now, identifier = $identifier;
        ActorRef = $this")
      throw new JustBecauseException(s"$self, identifier =
        $identifier, upon receiving a Fail message")
    case SuperviseeActor.Nudge =>
      import util.Random
      if (Random.nextBoolean())
        throw new MadeUpException(s"$self, identifier =
          $identifier, random effect on a Nudge message")
      log.info(s"$self receives nudge from $sender, identifier =
        $identifier; ActorRef = $this")
  }
}
```

Exception Handling: “Catching”

```
class SupervisorActor extends Actor with ActorLogging {
  ...
  val child = context.actorOf(...)
  override def receive: Receive = {
    case SupervisorActor.FailChild => child ! SuperviseeActor.Fail
    case SupervisorActor.NudgeChild => child ! SuperviseeActor.Nudge
  }

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 1, withinTimeRange = 5
      second) {
      case _: ArithmeticException => Resume
      case _: NullPointerException => Restart
      case _: IllegalArgumentException => Stop
      case _: IOException => Stop
      case x: JustBecauseException => {
        log.error(s" JustBecauseException occurred for the most
          outrageous reason;\n<<<$x>>>\n Restarting")
        Restart
      }
      case _: MadeUpException => {
        log.error(s" MadeUpException occurred;\n Resuming")
        Resume
      }
      case _: Exception => Escalate
    }
}
```

End-to-End Principle

Popularized by Jerome Saltzer, David Reed, and David Clark

- ▶ Originally formulated for computer network protocols
 - ▶ Usual examples pertain to error checking and performance
 - ▶ A similar case could be made for encryption
- ▶ Applies to (distributed) computing more generally
 - ▶ Any functionality that reflects application meaning must be verified at the end points
 - ▶ Such functionality
 - ▶ Does not need to be provided in the interior of the network, because it would need to be repeated at the end points
- ▶ Functionality that is not needed for a layer should not be provided in that layer because it is
 - ▶ Either superfluous—hence wastes resources
 - ▶ Or replicated (with a twist) in a higher layer—hence wastes resources
- ▶ In the case of actors
 - ▶ Ignoring message reliability and global ordering is wise
 - ▶ But why not also discard pairwise FIFO (not in actors but added by Akka)?

Actors and Protocols

Protocols to be introduced later

- ▶ Actors
 - ▶ Separate business logic from infrastructure
- ▶ Protocols
 - ▶ Separate reasoning from coordination
- ▶ Concordance: Protocols are geared for coordination of actor-like computational entities
 - ▶ Asynchronous (nonblocking) messaging
 - ▶ Shared nothing representation of local state
- ▶ Complementarity
 - ▶ Actors assume pairwise FIFO
 - ▶ Actors lack a model of multiparty interactions
 - ▶ Actors lack an explicit model of causality for messages
 - ▶ Actors don't provide an information model for messages
 - ▶ Protocols deal with interactions; actors deal with computations that can interact