



# Message Processing Semantics

Principles of Functional Programming

Roland Kuhn

## Actor Encapsulation

No direct access is possible to the actor behavior.

Only messages can be sent to known addresses (`ActorRef`):

- ▶ every actor knows its own address (`self`)
- ▶ creating an actor returns its address
- ▶ addresses can be sent within messages (e.g. `sender`)

## Actor Encapsulation

No direct access is possible to the actor behavior.

Only messages can be sent to known addresses (`ActorRef`):

- ▶ every actor knows its own address (`self`)
- ▶ creating an actor returns its address
- ▶ addresses can be sent within messages (e.g. `sender`)

Actors are completely independent agents of computation:

- ▶ local execution, no notion of global synchronization
- ▶ all actors run fully concurrently
- ▶ message-passing primitive is one-way communication

## Actor-Internal Evaluation Order

An actor is effectively single-threaded:

- ▶ messages are received sequentially
- ▶ behavior change is effective before processing the next message
- ▶ processing one message is the atomic unit of execution

This has the benefits of synchronized methods, but blocking is replaced by enqueueing a message.

## The Bank Account (revisited)

It is good practice to define an Actor's messages in its companion object.

```
object BankAccount {  
  case class Deposit(amount: BigInt) {  
    require(amount > 0)  
  }  
  case class Withdraw(amount: BigInt) {  
    require(amount > 0)  
  }  
  case object Done  
  case object Failed  
}
```



## Actor Collaboration

- ▶ picture actors as persons
- ▶ model activities as actors

## Transferring Money (0)

```
object WireTransfer {  
  case class Transfer(from: ActorRef, to: ActorRef, amount: BigInt)  
  case object Done  
  case object Failed  
}
```

## Transferring Money (1)

```
class WireTransfer extends Actor {  
  import WireTransfer._  
  
  def receive = {  
    case Transfer(from, to, amount) =>  
      from ! BankAccount.Withdraw(amount)  
      context.become(awaitWithdraw(to, amount, sender()))  
  }  
  
  def awaitWithdraw ...  
}
```

## Transferring Money (2)

```
class WireTransfer extends Actor {  
  ...  
  
  def awaitWithdraw(to: ActorRef, amount: BigInt, client: ActorRef): Receive = {  
    case BankAccount.Done =>  
      to ! BankAccount.Deposit(amount)  
      context.become(awaitDeposit(client))  
    case BankAccount.Failed =>  
      client ! Failed  
      context.stop(self)  
  }  
  
  def awaitDeposit ...  
}
```

## Transferring Money (3)

```
class WireTransfer extends Actor {  
  ...  
  
  def awaitDeposit(client: ActorRef): Receive = {  
    case BankAccount.Done =>  
      client ! Done  
      context.stop(self)  
  }  
}
```

## Message Delivery Guarantees

- ▶ all communication is inherently unreliable

## Message Delivery Guarantees

- ▶ all communication is inherently unreliable
- ▶ delivery of a message requires eventual availability of channel & recipient

**at-most-once:** sending once delivers  $[0, 1]$  times

**at-least-once:** resending until acknowledged delivers  $[1, \infty)$  times

**exactly-once:** processing only first reception delivers 1 time

# Reliable Messaging

Messages support reliability:

- ▶ all messages can be persisted
- ▶ can include unique correlation IDs
- ▶ delivery can be retried until successful

Reliability can only be ensured by business-level acknowledgement.

## Making the Transfer Reliable

- ▶ log activities of `WireTransfer` to persistent storage
- ▶ each transfer has a unique ID
- ▶ add ID to `Withdraw` and `Deposit`
- ▶ store IDs of completed actions within `BankAccount`

## Message Ordering

If an actor sends multiple messages to the same destination, they will not arrive out of order (this is Akka-specific).

## Summary

Actors are fully encapsulated, independent agents of computation.

Messages are the only way to interact with actors.

Explicit messaging allows explicit treatment of reliability.

The order in which messages are processed is mostly undefined.