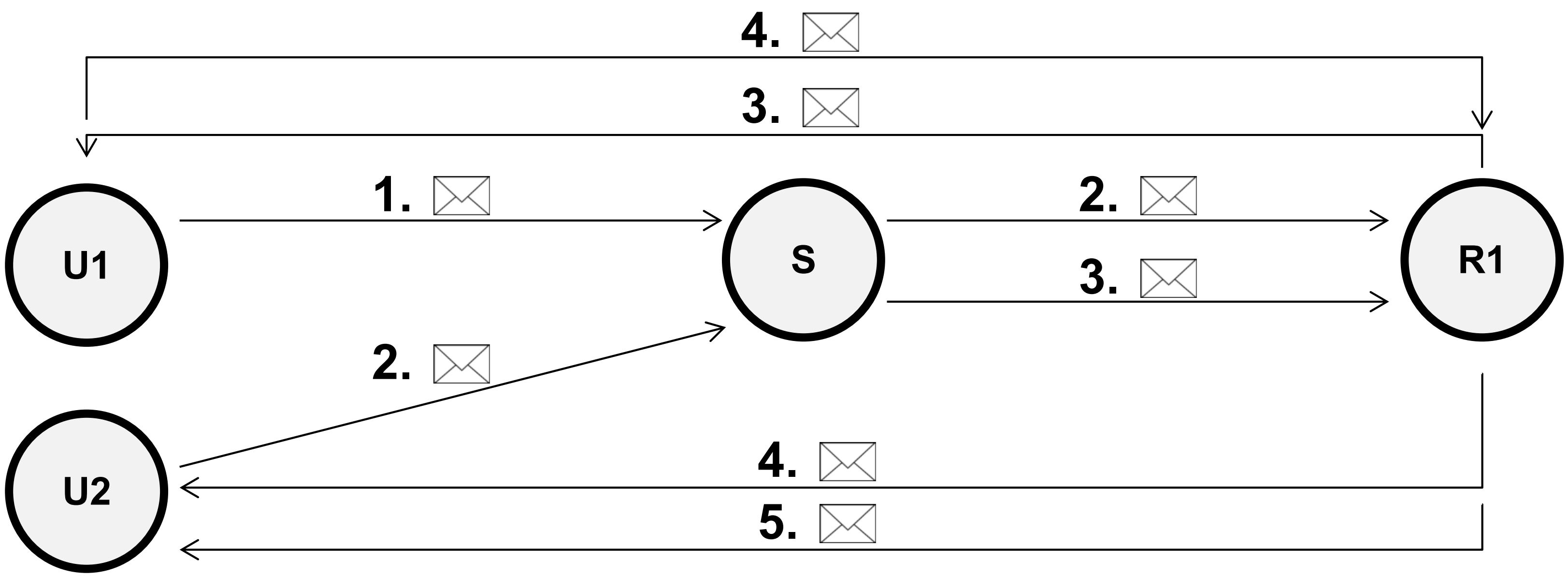
Fachhochschule Nordwestschweiz

Actors – Message Passing

- **Actor Model**
  - Basics
- **Actors in Scala**
  - Basics
  - Do it yourself "Lenzo Palace"
  - Advanced Messaging
  - Finite State Machines

Objectives:
- **Understand the actor model and its pros / cons**
- **Write small actor based programs in Scala**

Communication in Actor Systems



- **Actors communicate only by message passing**
  - Messages are sent asynchronously
  - No shared mutable state between actors! (Share nothing concurrency)
  - Messages must have "send by value" semantics (immutable)

Anatomy of an Actor

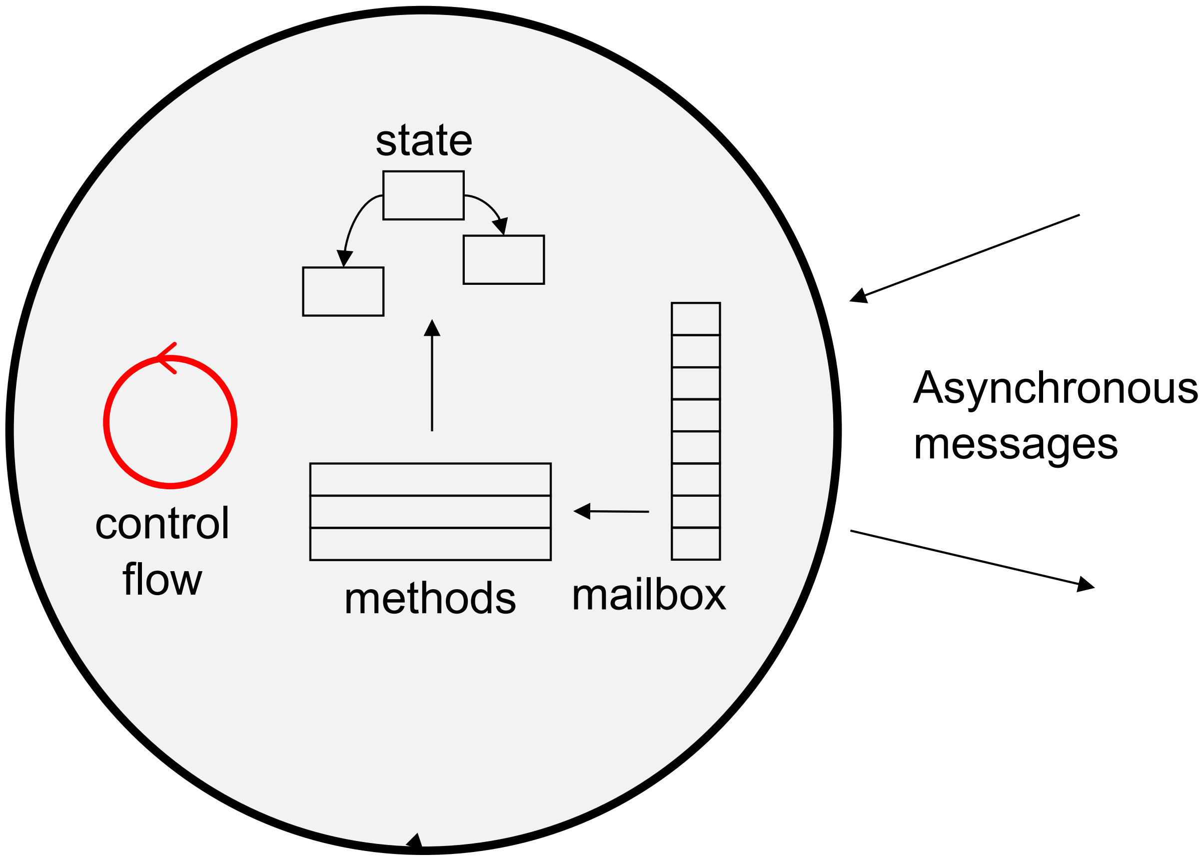
- **Actor =**

  Independent control

  + Encapsulated state

  + Behavior

  + Mailbox



Actor Model

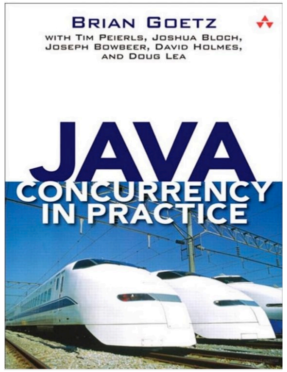
- **Actors = autonomous concurrent objects**
  - Actors are executing asynchronously:
    Independent control flow, active objects
  - Actors have a private state:
    No shared mutable state between actors!
    - No race conditions
    - No lost update problem
  - Actors communicate via asynchronous message passing:
    Messages are buffered in the actors mailbox and processed sequentially
  - Actors have a behavior:
    Upon receipt of a message, the actor can
    - Send a number of messages to other actors
    - Create a number of new actors
    - Change its state

Fachhochschule
Nordwestschweiz

The Actor Approach to Concurrency

- **Remember:**

  If multiple threads modify shared mutable state without coordination, your program is broken!



- **The Actor Approach: No shared, mutable state**
  – All mutable state is private
  – All shared state is immutable
  – Communicate via immutable, asynchronous message-passing

Actors – Message Passing

- **Actor Model**
  - Basics
- **Actors in Scala**
  - Basics
  - Do it yourself "Lenzo Palace"
  - Advanced Messaging
  - Finite State Machines

Actors in Scala (akka.io)

```scala
import scala.language.postfixOps // required for `a ! msg` p
import akka.actor.{ActorSystem, Actor, ActorRef, Props}

val as = ActorSystem("as")           // Actor infrastructure

class PrintActor extends Actor {     // Actor definition
  var nthRequest = 0                 // Mutable state
  def receive = { case msg =>        // Behavior
    nthRequest += 1
    println(s"$nthRequest:$msg")
  }
}

val printActor: ActorRef = as.actorOf(Props[PrintActor]) // Creation
```

```
scala>printActor ! "Hello" // Sending asynchronous messages
scala>1:Hello
scala>printActor ! "Bye"
scala>2:Bye
```

Akka High Performance Systems

Build powerful reactive, concurrent, and distributed applications more easily

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

akka

Akka is *the* implementation of the Actor Model on the JVM.

Simpler Concurrent & Distributed Systems

Actors and Streams let you build systems that scale *up*, using the resources of a server more efficiently, and *out*, using multiple servers.

Resilient by Design

Building on the principles of The Reactive Manifesto Akka allows you to write systems that self-heal and stay responsive in the face of failures.

High Performance

Up to 50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.

Elastic & Decentralized

Distributed systems without single points of failure. Load balancing and adaptive routing across nodes. Event Sourcing and CQRS with Cluster Sharding. Distributed Data for eventual consistency using CRDTs.

Reactive Streaming Data

Asynchronous non-blocking stream processing with backpressure. Fully async and streaming HTTP server and client provides a great platform for building microservices. Streaming integrations with Alpakka.

Creating an Actor Instance

- **actorOf**
  - creates and starts the actor asynchronously
  - returns an immutable reference of type ActorRef to the created actor

```
import akka.actor._
val as = ActorSystem("as")

class PrintActor extends Actor {
  def receive = { case msg: Int => println(msg) }
}

val printActor: ActorRef = as.actorOf(Props[PrintActor])
```

- **Actors cannot be created using new**

```
scala> new PrintActor
akka.actor.ActorInitializationException:
You cannot create an instance of [PrintActor] explicitly using the constructor (new).
You have to use one of the factory methods to create a new actor.
...
```

Creating an Actor Instance

- **Default constructor**

```
class PrintActor extends Actor { ... }

val print: ActorRef = as.actorOf(Props[PrintActor])
```

- **Non default constructor**

```
class PrintActor(pre: String) extends Actor { ... }

val print: ActorRef = as.actorOf(Props(new PrintActor("Msg:")))
```

- **Anonymous Actor subclass**

```
val print: ActorRef = as.actorOf(Props(
  new Actor {
    def receive = { case msg => println(msg) }
  }
))
```

Creating Actor Hierarchies

```
import akka.actor._

class ChildActor() extends Actor {
  def receive = { case msg => println("I'm " + self + " : " + msg)}
}

class ParentActor extends Actor {
  def receive = {
    case name: String =>
      val child = context.actorOf(Props[ChildActor], "child")
      child ! "Greets from dad"
  }
}
```

```
val as = ActorSystem("as")
val p = as.actorOf(Props[ParentActor], "parent")
p ! "Hi Kid"
```

Sending Messages: Fire-Forget

- **The ! (tell) operator sends messages**
  - Asynchronous

```
a ! msg
```

  - Message is stored in the mailbox of the receiver
  - Messages can be anything (`Any`)
  - Result of a send expression is ()

- **Message Delivery Guarantees**
  - at-most-once delivery / no guaranteed delivery (*send-and-pray*)
    - Messages may be lost
  - message ordering per sender-receiver pair
    - *For a given pair of actors, messages sent from the first to the second will not be received out-of-order*

Receiving Messages

- **receive specifies the initial behavior of an actor**
  - Defines a series of case statements each consisting of
    - A pattern which defines what messages your Actor can handle
    - An implementation of how the matched message should be processed

```
def receive = {
   case pattern_1 => StatementSeq_1
   case pattern_2 => StatementSeq_2
   ...
   case pattern_N => StatementSeq_N
}
```

- **Every time a message is processed, it is matched against the current behavior of the actor**
  - Match: Execute the corresponding statements
  - No match: Message will be published to the ActorSystem's EventStream

Receiving Messages

- **receive defines a partial function from Any => Unit**

```
def receive: PartialFunction[Any,Unit]
```

  - Functions which are defined only for certain arguments

```
val pf: PartialFunction[Any,Unit] = {
  case i: Int if i > 42 => println("huge")
  case s: String        => println(s.reverse)
}
```

  - Check if the PartialFunction will accept a given argument

```
pf.isDefinedAt(42) // false | pf.isDefinedAt(43) // true
```

  - PartialFunctions can be applied

```
pf(42) // throws MatchError | pf(43) // prints "huge"
```

Example: Print Actor

```
import akka.actor.{Actor, ActorSystem, Props}
val as = ActorSystem("as")

class PrintActor extends Actor {
  def receive = { case msg => println("received msg: " + msg) }
}

val printActor = as.actorOf(Props[PrintActor])
```

```
scala> printActor ! "hello"
scala> received msg: hello

scala> printActor ! 4711
scala> received msg: 4711
```

Example: Print Actor

```
import akka.actor.{Actor, ActorSystem, Props}
val as = ActorSystem("as")

val f: PartialFunction[Any,Unit] = {
  case msg => println("received msg: " + msg)
}

class PrintActor extends Actor {
  def receive = f
}

val printActor = as.actorOf(Props[PrintActor])
```

```
scala> printActor ! "hello"
scala> received msg: hello

scala> printActor ! 4711
scala> received msg: 4711
```

Receiving Messages: Case Classes

- **Typically case classes are used as messages**
  - Describe the vocabulary an actor understands (its API)
  - Convenient to be used in match expressions

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

case class PrintMsg(msg: String)
case class ShoutMsg(msg: String)

class PrintActor extends Actor {
  def receive = {
    case PrintMsg(m) => println("received: " + m)
    case ShoutMsg(m) => println("RECEIVED: " + m.toUpperCase)
  }
}
```

```
scala>ActorSystem("as").actorOf(Props[PrintActor])!ShoutMsg("Hello")
scala>RECEIVED: HELLO
```

Receiving Messages: Matching with Guards

- **Patterns can be refined with guards**

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

case class PrintMsg(msg: String)

class PrintActor extends Actor {
  def receive = {
    case PrintMsg(m) if m.contains("@") => println("mail: " + m)
    case PrintMsg(m)                    => println("text: " + m)
  }
}

val printActor = ActorSystem("as").actorOf(Props[PrintActor])
```

```
scala>printActor ! PrintMsg("me@you.com")
scala>mail: me@you.com
scala>printActor ! PrintMsg("Hello")
scala>text: Hello
```

Actors and the JMM

- **The actor send rule**
  - **Definition:** The send of a message happens-before the receive of that message
  - Consequence: Even messages which are not properly constructed are correctly visible

- **The actor subsequent processing rule**
  - **Definition:** Processing of one message happens before processing the next message by the same actor
  - Consequence: Changes to mutable state within an actor are visible when the next message is processed
    - Remark: Not every message is necessarily processed by the same thread

Actors and Threads

- **Message processing is scheduled on a thread pool**

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

def info(s: String): Unit =
  println(Thread.currentThread().getName() + ": " + s)

class PrintActor extends Actor {
  def receive = { case msg: String => info(msg) }
}

val as = ActorSystem("as")
val p1 = as.actorOf(Props[PrintActor], "p1")
val p2 = as.actorOf(Props[PrintActor], "p2")
info("Sending message")
p1 ! "P1: Hi"
p2 ! "P2: Hi"
p1 ! "P1: Bye"
p2 ! "P2: Bye"
```

```
main: Sending message
as-akka.actor.default-dispatcher-2: P1: Hi
as-akka.actor.default-dispatcher-3: P2: Hi
as-akka.actor.default-dispatcher-2: P1: Bye
as-akka.actor.default-dispatcher-3: P2: Bye
```

Actors – Message Passing

- **Actor Model**
  - Basics
- **Actors in Scala**
  - Basics
  - Do it yourself "Lenzo Palace"
  - Advanced Messaging
  - Finite State Machines

Actors – Message Passing

- **Actor Model**
  - Basics
- **Actors in Scala**
  - Basics
  - Do it yourself "Lenzo Palace"
  - Advanced Messaging
  - Finite State Machines

Reply to Messages

- **Use self (of type ActorRef) to refer to the current actor**
  - Can be safely passed around

```
import akka.actor._
val as = ActorSystem("as")

case class Msg(msg: String, sender: ActorRef)

class EchoActor extends Actor {
  def receive = { case Msg(msg,client) => client ! msg}}

val echoActor = as.actorOf(Props[EchoActor])

class Sender extends Actor {
  echoActor ! Msg("Hello", self)
  def receive = { case t => println(t) }
}
```

```
scala>as.actorOf(Props[Sender]) // starts a sender actor
scala>Hello
```

Reply to Messages

- **Use sender (of type ActorRef) to refer to the actor which sent the message which is currently processed**

```
import akka.actor.{Actor,ActorSystem, Props}
val as = ActorSystem("as")


class EchoActor extends Actor {
  def receive = { case msg => sender ! msg }} // reply to sender

val echoActor = as.actorOf(Props[EchoActor])

class Sender extends Actor {
  echoActor ! "Hello"
  def receive = { case t => println(t) }
}
```

```
scala>as.actorOf(Props[Sender]) // starts a sender actor
scala>Hello
```

Receive Timeout

- **ActorContext#setReceiveTimeout defines the inactivity timeout**
  - In case of no activity a ReceiveTimeout message is triggered
  - Once set, the receive timeout stays in effect
  - Pass in Duration.Undefined to switch off this feature

```
import akka.actor._; import scala.concurrent.duration._
class TimeOutActor extends Actor {
  context.setReceiveTimeout(3.second)
  def receive = {
    case "Tick"           => println("Tick")
    case ReceiveTimeout => println("TIMEOUT")
  }
}
```

```
scala> val a = ActorSystem("as").actorOf(Props[TimeOutActor])
scala> TIMEOUT
scala> TIMEOUT
```

Fachhochschule Nordwestschweiz

Ask: Send-And-Receive-Future

- **Pattern for sending a message and receiving a Future containing an answer**

```
import akka.actor._
import akka.pattern.ask // brings '?' into scope
import akka.util.Timeout
import scala.concurrent.Future
import scala.concurrent.duration._

class EchoActor extends Actor {
  def receive = { case msg => sender ! msg }
}

val as = ActorSystem("as")
val echoActor = as.actorOf(Props[EchoActor])

implicit val timeout = Timeout(3.seconds) // consumed by '?'
val futResult: Future[Any] = (echoActor ? "Hello")
```

Ask: Send-And-Receive-Future

- '?' takes an implicit timeout (can be passed explicitly)

```
val timeout = Timeout(3 seconds)
val futResult: Future[Any] = echoActor ? ("Hello")(timeout)
```

  - Future is completed with AskTimeoutException in case of timeout

- Result type can be casted using mapTo[TargetType]

```
val futResult: Future[String] = (echoActor ? "Hello").mapTo[String]
```

- For further Future processing an ExecutionContext is required

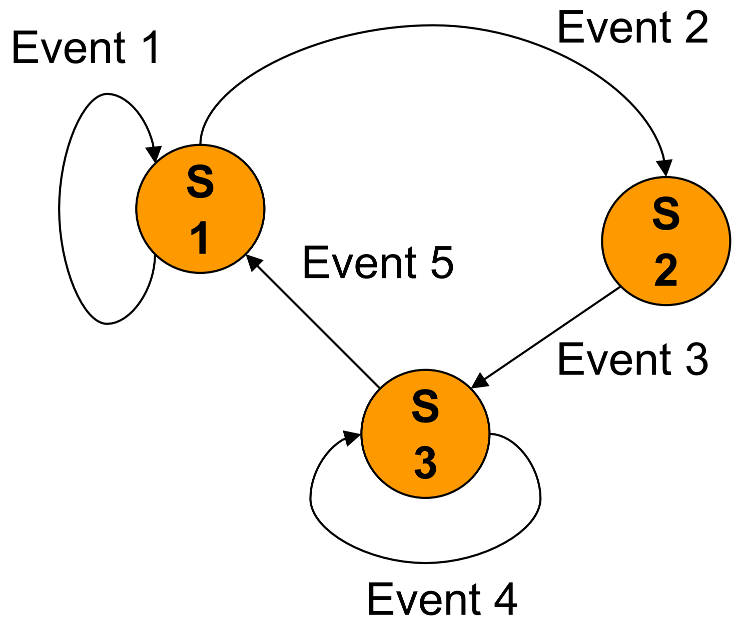
```
val as = ActorSystem("as")
import as.dispatcher // ExecutionContext required by Future#map
futureResult.map(s => s.toUpperCase)
```

Actors – Message Passing

- **Actor Model**
  - Basics
- **Actors in Scala**
  - Basics
  - Do it yourself "Lenzo Palace"
  - Advanced Messaging
  - Finite State Machines

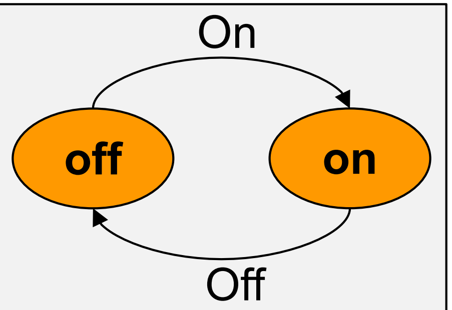
Finite State Machines

- **Finite State Machine (FSM)**
  - Consists of a number of states
  - State changes are triggered by events
- **Scala implementation**
  - FSM is represented as an actor
  - Events are represented as messages
  - States can be represented
    - in a variable (state directed)
    - by state specific behavior (receive hotswap)



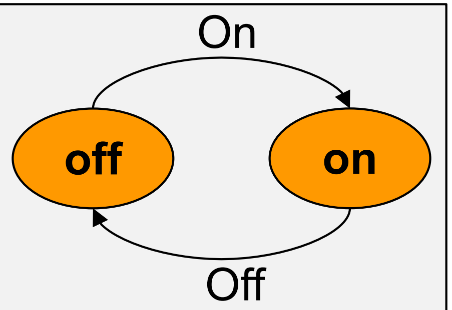
FSM Example: Switch (state directed)

```
import akka.actor.{ Actor, ActorSystem, Props }
case object On
case object Off

class Switch extends Actor {
  var on = false
  def receive = {
    case On if !on => println("turned on");  on = true
    case Off if on => println("turned off"); on = false
    case _ => println("ignore")
  }
}

val as = ActorSystem("as")
val switch = as.actorOf(Props[Switch])
```




```
scala>switch ! On
scala>turned on
scala>switch ! On
scala>ignore
```

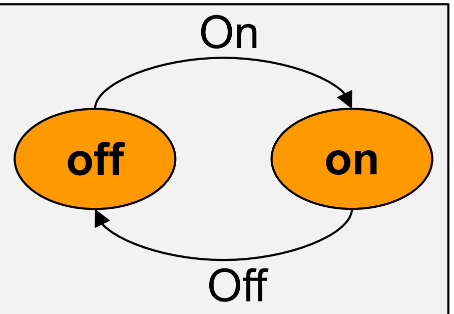
FSM Example: Switch (receive hotswap)




```
import akka.actor.{ Actor, ActorSystem, Props }
case object On
case object Off

class Switch extends Actor {
  val offBehavior: PartialFunction[Any,Unit] = {
    case On => println("turned on"); context.become(onBehavior)
    case _  => println("ignore")
  }
  val onBehavior: PartialFunction[Any,Unit] = {
    case Off => println("turned off"); context.become(offBehavior)
    case _   => println("ignore")
  }
  def receive = offBehavior
}

val as = ActorSystem("as")
val switch = as.actorOf(Props[Switch])
```

```
scala>switch ! On
scala>turned on
scala>switch ! On
scala>ignore
```

FSM Example: Switch (receive hotswap)




```
import akka.actor.{ Actor, ActorSystem, Props }
case object On
case object Off

class Switch extends Actor {
  val offBehavior: PartialFunction[Any,Unit] = {
    case On => println("turned on"); context.become(onBehavior, false)
    case _  => println("ignore")
  }
  val onBehavior: PartialFunction[Any,Unit] = {
    case Off => println("turned off"); context.unbecome()
    case _   => println("ignore")
  }
  def receive = offBehavior
}

val as = ActorSystem("as")
val switch = as.actorOf(Props[Switch])
```

do not discard the current behaviour but put the given one atop.

```
scala>switch ! On
scala>turned on
scala>switch ! On
scala>ignore
```

Pros / Cons

- **Pros**
  - Inside an actor one can use sequential programming
  - Communication is performed explicit
  - Works well in a distributed setting
  - Reasoning about concurrency and safety becomes easier

- **Cons**
  - Synchronous communication requires message roundtrip
  - Distributed mutable state (not easy to predict behavior)
  - Deadlocks still possible (when blocking on futures)
  - Transactions across several actors are difficult, i.e. each actor has to be moved into a special state