Reactive Cheat Sheet

Help Center

This cheat sheet originated from the forums. There are certainly a lot of things that can be improved! If you would like to contribute, you have two options:

Click the “Edit” button on this file on GitHub: https://github.com/sjuvekar/reactive-programming-scala/blob/master/ReactiveCheatSheet.md You can submit a pull request directly from there without checking out the git repository to your local machine.

Fork the repository https://github.com/sjuvekar/reactive-programming-scala/ and check it out locally. To preview your changes, you need jekyll. Navigate to your checkout and invoke `jekyll serve --watch` (or `jekyll --auto --server` if you have an older jekyll version), then open the page `http://localhost:4000/ReactiveCheatSheet.html`.

Partial Functions

A subtype of trait `Function1` that is well defined on a subset of its domain.

```
trait PartialFunction[-A, +R] extends Function1[-A, +R] {
  def apply(x: A): R
  def isDefinedAt(x: A): Boolean
}
```

Every concrete implementation of PartialFunction has the usual `apply` method along with a boolean method `isDefinedAt`.

Important: An implementation of PartialFunction can return `true` for `isDefinedAt` but still end up throwing RuntimeException (like MatchError in pattern-matching implementation).

A concise way of constructing partial functions is shown in the following example:

```
trait Coin {}
case class Gold() extends Coin {}
case class Silver() extends Coin {}

val pf: PartialFunction[Coin, String] = {
  case Gold() => "a golden coin"
  // no case for Silver(), because we're only interested in Gold()
}

println(pf.isDefinedAt(Gold()))   // true
println(pf.isDefinedAt(Silver())) // false
println(pf(Gold()))               // a golden coin
println(pf(Silver()))             // throws a scala.MatchError
```

For-Comprehension and Pattern Matching

A general For-Comprehension is described in Scala Cheat Sheet here: https://github.com/lrytz/progfun-wiki/blob/gh-pages/CheatSheet.md. One can also use Patterns inside for-expression. The simplest form of for-expression pattern looks like

```
for { pat <- expr} yield e
```

where `pat` is a pattern containing a single variable `x`. We translate the `pat <- expr` part of the expression to

```
x <- expr withFilter {
  case pat => true
  case _ => false
} map {
  case pat => x
}
```

The remaining parts are translated to `map, flatMap, withFilter` according to standard for-comprehension rules.

Random Generators with For-Expressions

The `map` and `flatMap` methods can be overridden to make a for-expression versatile, for example to generate random elements from an arbitrary collection like lists, sets etc. Define the following trait `Generator` to do this.

```
trait Generator[+T] { self =>
  def generate: T
  def map[S](f: T => S) : Generator[S] = new Generator[S] {
    def generate = f(self.generate)
  }
  def flatMap[S](f: T => Generator[S]) : Generator[S] = new Generator[S] {
    def generate = f(self.generate).generate
  }
}
```

Let's define a basic integer random generator as

```
val integers = new Generator[Int] {
  val rand = new java.util.Random
  def generate = rand.nextInt()
}
```

With these definition, and a basic definition of `integer` generator, we can map it to other domains like `booleans, pairs, intervals` using for-expression magic

```
val booleans = for {x <- integers} yield x > 0
val pairs = for {x <- integers; y<- integers} yield (x, y)
def interval(lo: Int, hi: Int) : Generator[Int] = for { X <- integers } yield lo + x % (hi - lo)
```

Monads

A monad is a parametric type MT with two operations: `flatMap` and `unit`.

```
trait M[T] {
  def flatMap[U](f: T => M[U]) : M[U]
  def unit[T](x: T) : M[T]
}
```

These operations must satisfy three important properties:

1. **Associativity:** `(x flatMap f) flatMap g == x flatMap (y => f(y) flatMap g)`
2. **Left unit:** `unit(x) flatMap f == f(x)`
3. **Right unit:** `m flatMap unit == m`

Many standard Scala Objects like `List, Set, Option, Gen` are monads with identical implementation of `flatMap` and specialized implementation of `unit`. An example of non-monad is a special `Try` object that fails with a non-fatal exception because it fails to satisfy Left unit (See lectures).

Monads and For-Expression

Monads help simplify for-expressions.

Associativily helps us "inline" nested for-expressions and write something like

```
for { x <- e1; y <- e2(x) ... }
```

Right unit helps us eliminate for-expression using the identity

```
for{x <- m} yield x == m
```

Pure functional programming

In a pure functional state, programs are side-effect free, and the concept of time isn't important (i.e. redoing the same steps in the same order produces the same result).

When evaluating a pure functional expression using the substitution model, no matter the evaluation order of the various sub-expressions, the result will be the same (some ways may take longer than others). An exception may be in the case where a sub-expression is never evaluated (e.g. second argument) but whose evaluation would loop forever.

Mutable state

In a reactive system, some states eventually need to be changed in a mutable fashion. An object has a state if its behavior has a history. Every form of mutable state is constructed from variables:

```
var x: String = "abc"
x = "hi"
var nb = 42
```

The use of a stateful expression can complexify things. For a start, the evaluation order may matter. Also, the concept of identity and change gets more complex. When are two expressions considered the same? In the following (pure functional) example, x and y are always the same (concept of **referential transparency**):

```
val x = E; val y = E
val x = E; val y = x
```

But when a stateful variable is involved, the concept of equality is not as straightforward. “Being the same” is defined by the property of **operational equivalence**. x and y are operationally equivalent if no possible test can distinsuish between them.

Considering two variables x and y, if you can create a function f so that f(x, y) returns a different result than f(x, x) then x and y are different. If no such function exist x and y are the same.

As a consequence, the substitution model ceases to be valid when using assignments.

Loops

Variables and assignments are enough to model all programs with mutable states and loops in essence are not required. **Loops can be modeled using functions and lazy evaluation**. So, the expression

```
while (condition) { command }
```

can be modeled using function `WHILE` as

```
def WHILE(condition: => Boolean)(command: => Unit): Unit =
    if (condition) {
        command
        WHILE(condition)(command)
    }
    else ()
```

Note:

- Both **condition** and **command** are **passed by name**
- **WHILE** is **tail recursive**

For loop

The treatment of for loops is similar to the **For-Comprehensions** commonly used in functional programming. The general expression for `for loop` equivalent in Scala is

```
for(v1 <- e1; v2 <- e2; ...; v_n <- e_n) command
```

Note a few subtle differences from a For-expreesion. There is no `yield` expression, `command` can contain mutable states and `e1, e2, ..., e_n` are expressions over arbitrary Scala collections. This for loop is translated by Scala using a **foreach** combinator defined over any arbitrary collection. The signature for **foreach** over collection **T** looks like this

```
def foreach(f: T => Unit) : Unit
```

Using foreach, the general for loop is recursively translated as follows:

```
for(v1 <- e1; v2 <- e2; ...; v_n <- e_n) command =
    e1 foreach (v1 => for(v2 <- e2; ...; v_n <- e_n) command)
```

Monads and Effect

Monads and their operations like flatMap help us handle programs with side-effects (like exceptions) elegantly. This is best demonstrated by a Try-expression. **Note:** Try-expression is not strictly a Monad because it does not satisfy all three laws of Monad mentioned above. Although, it still helps handle expressions with exceptions.

Try

The parametric Try class as defined in Scala.util looks like this:

```
abstract class Try[T]
case class Success[T](elem: T) extends Try[T]
case class Failure(t: Throwable) extends Try[Nothing]
```

`Try[T]` can either be `Success[T]` or `Failure(t: Throwable)`

```
import Scala.util.{Try, Success, Failure}

def answerToLife(nb: Int) : Try[Int] = {
  if (nb == 42) Success(nb)
  else Failure(new Exception("WRONG"))
}

answerToLife(42) match {
  case Success(t)           => t        // returns 42
  case failure @ Failure(e) => failure  // returns Failure(java.Lang.Exception: WRONG)
}
```

Now consider a sequence of scala method calls:

```
val o1 = SomeTrait()
val o2 = o1.f1()
val o3 = o2.f2()
```

All of these method calls are synchronous, blocking and the sequence computes to completion as long as none of the intermediate methods throw an exception. But what if one of the methods, say `f2` does throw an exception? The `Try` class defined above helps handle these exceptions elegantly, provided we change return types of all methods `f1, f2, ...` to `Try[T]`. Because then, the sequence of method calls translates into an elegant for-comprehension:

```
val o1 = SomeTrait()
val ans = for {
    o2 <- o1.f1();
    o3 <- o2.f2()
} yield o3
```

This transformation is possible because `Try` satisfies 2 properties related to `flatMap` and `unit` of a **monad**. If any of the intermediate methods `f1, f2` throws and exception, value of `ans` becomes `Failure`. Otherwise, it becomes `Success[T]`.

Monads and Latency

The Try Class in previous section worked on synchronous computation. Synchronous programs with side effects block the subsequent instructions as long as the current computation runs. Blocking on expensive computation might render the entire program slow!. **Future** is a type of monad the helps handle exceptions and latency and turns the program in a non-blocking asynchronous program.

Future

Future trait is defined in scala.concurrent as:

```
trait Future[T] {
    def onComplete(callback: Try[T] => Unit)
    (implicit executor: ExecutionContext): Unit
}
```

The Future trait contains a method `onComplete` which itself takes a method, `callback` to be called as soon as the value of current Future is available. The insight into working of Future can be obtained by looking at its companion object:

```
object Future {
    def apply(body: => T)(implicit context: ExecutionContext): Future[T]
}
```

This object has an apply method that starts an asynchronous computation in current context, returns a `Future` object. We can then subsribe to this `Future` object to be notified when the computation finishes.

```
import scala.concurrent._
import ExecutionContext.Implicits.global

// The function to be run asyncronously
val answerToLife: Future[Int] = future {
  42
}

// These are various callback functions that can be defined
answerToLife onComplete {
  case Success(result) => result
  case Failure(t) => println("An error has occured: " + t.getMessage)
}

answerToLife onSuccess {
  case result => result
}

answerToLife onFailure {
  case t => println("An error has occured: " + t.getMessage)
}

answerToLife.now    // only works if the future is completed
```

Combinators on Future

A `Future` is a `Monad` and has `map`, `filter`, `flatMap` defined on it. In addition, Scala's Futures define two additional methods:

```
def recover(f: PartialFunction[Throwable, T]): Future[T]
def recoverWith(f: PartialFunction[Throwable, Future[T]]): Future[T]
```

These functions return robust features in case current features fail.

Finally, a `Future` extends from a trait called `Awaitable` that has two blocking methods, `ready` and `result` which take the value 'out of' the Future. The signatures of these methods are

```
trait Awaitable[T] extends AnyRef {
    abstract def ready(t: Duration): Unit
    abstract def result(t: Duration): T
}
```

Both these methods block the current execution for a duration of `t`. If the future completes its execution, they return: `result` returns the actual value of the computation, while `ready` returns a Unit. If the future fails to complete within time t, the methods throw a `TimeoutException`.

`Await` can be used to wait for a future with a specified timeout, e.g.

```
userInput: Future[String] = ...
Await.result(userInput, 10 seconds)   // waits for user input for 10 seconds, after which throws a TimeoutException
```

async and await

Async and await allow to run some part of the code aynchronously. The following code computes asynchronously any future inside the `await` block

```
import scala.async.Async._

def retry(noTimes: Int)(block: => Future[T]): Future[T] = async {
  var i = 0;
  var result: Try[T] = Failure(new Exception("Problem!"))
  while (i < noTimes && result.isFailure) {
    result = await { Try(block) }
    i += 1
  }
  result.get
}
```

Promises

A Promise is a nomad which can be used to successfully complete a future with a value (thus completing the promise) or failed with an exception

```
trait Promise[T]
  def future: Future[T]
  def complete(result: Try[T]): Unit  // to call when the promise is completed
  def tryComplete(result: Try[T]): Boolean
}
```

It is used as follows:

```
val p = Promise[T]  // defines a promise
p.future            // returns a future that will complete when p.complete() is called
p.complete(Try[T])  // completes the future
p success T         // successfully completes the promise
```

Observables

Observables are asynchronous streams of data. Contrary to Futures, they can return multiple values.

```
trait Observable[T] {
  def Subscribe(observer: Observer[T]): Subscription
}

trait Observer[T] {
  def onNext(value: T): Unit             // callback function when there's a next value
  def onError(error: Throwable): Unit  // callback function where there's an error
  def onCompleted(): Unit              // callback function when there is no more value
}

trait Subscription {
  def unsubscribe(): Unit    // to call when we're not interested in recieving any more values
}
```

Observables can be used as follows:

```
import rx.lang.scala._

val ticks: Observable[Long] = Observable.interval(1 seconds)
val evens: Observable[Long] = ticks.filter(s => s%2 == 0)

val bugs: Observable[Seq[Long]] = ticks.buffer(2, 1)
val s = bugs.subscribe(b=>println(b))

s.unsubscribe()
```

Some observable functions (see more at http://rxscala.github.io/scaladoc/index.html#rx.lang.scala.Observable):

- `Observable[T].flatMap(T => Observable[T]): Observable[T]` merges a list of observables into a single observable in a non-deterministic order
- `Observable[T].concat(T => Observable[T]): Observable[T]` merges a list of observables into a single observable, putting the results of the first observable first, etc.
- `groupBy[K](keySelector: T=>K): Observable[(K, Observable[T])]` returns an observable of observables, where the element are grouped by the key returned by `keySelector`

Subscriptions

Subscriptions are returned by Observables to allow to unsubscribe. With hot observables, all subscribers share the same source, which produces results independent of subscribers. With cold observables each subscriber has its own private source. If there is no subscriber no computation is performed.

Subscriptions have several subtypes: `BooleanSubscription` (was the subscription unsubscribed or not?), `CompositeSubscription` (collection of subscriptions that will be unsubscribed all at once), `MultipleAssignmentSubscription` (always has a single subscription at a time)

```
val subscription = Subscription { println("Bye") }
subscription.unsubscribe() // prints the message
subscription.unsubscribe() // doesn't print it again
```

Creating Rx Streams

Using the following constructor that takes an Observer and returns a Subscription

```
object Observable {
  def apply[T](subscribe: Observer[T] => Subscription): Observable[T]
}
```

It is possible to create several observables. The following functions suppose they are part of an Observable type (calls to `subscribe(...)` implicitely mean `this.subscribe(...)`):

```
// Observable never: never returns anything
def never(): Observable[Nothing] = Observable[Nothing](observer => { Subscription {} })

// Observable error: returns an error
def apply[T](error: Throwable): Observable[T] =
    Observable[T](observer => {
    observer.onError(error)
    Subscription {}
  }
)

// Observable startWith: prepends some elements in front of an Observable
def startWith(ss: T*): Observable[T] = {
    Observable[T](observer => {
    for(s <- ss) observer onNext(s)
    subscribe(observer)
  })
})

// filter: filters results based on a predicate
def filter(p: T=>Boolean): Observable[T] = {
  Observable[T](observer => {
    subscribe(
      (t: T) => { if(p(t)) observer.onNext(t) },
      (e: Thowable) => { observer.onError(e) },
      () => { observer.onCompleted() }
    )
  })
}

// map: create an observable of a different type given a mapping function
def map(f: T=>S): Observable[S] = {
  Observable[S](observer => {
    subscribe(
      (t: T) => { observer.onNext(f(t)) },
      (e: Thowable) => { observer.onError(e) },
      () => { observer.onCompleted() }
    )
  }
}

// Turns a Future into an Observable with just one value
def from(f: Future[T])(implicit execContext: ExecutionContext): Observable[T] = {
    val subject = AsyncSubject[T]()
    f onComplete {
      case Failure(e) => { subject.onError(e) }
      case Success(c) => { subject.onNext(c); subject.onCompleted() }
    }
    subject
```

```
}
```

Blocking Observables

`Observable.toBlockingObservable()` returns a blocking observable (to use with care). Everything else is non-blocking.

```
val xs: Observable[Long] = Observable.interval(1 second).take(5)
val ys: List[Long] = xs.toBlockingObservable.toList
```

Schedulers

Schedulers allow to run a block of code in a separate thread. The Subscription returned by its constructor allows to stop the scheduler.

```
trait Observable[T] {
  def observeOn(scheduler: Scheduler): Observable[T]
}

trait Scheduler {
  def schedule(work: => Unit): Subscription
  def schedule(work: Scheduler => Subscription): Subscription
  def schedule(work: (=>Unit)=>Unit): Subscription
}

val scheduler = Scheduler.NewThreadScheduler
val subscription = scheduler.schedule { // schedules the block on another thread
  println("Hello world")
}



// Converts an iterable into an observable
// works even with an infinite iterable
def from[T](seq: Iterable[T])
           (implicit scheduler: Scheduler): Obserable[T] = {
   Observable[T](observer => {
     val it = seq.iterator()
     scheduler.schedule(self => {     // the block between { ... } is run in a separate thread
       if (it.hasnext) { observer.onNext(it.next()); self() } // calling self() schedules the block of code to be executed
again
       else { observer.onCompleted() }
     })
   })
}
```

Actors

Actors represent objects and their interactions, resembling human organizations. They are useful to deal with the complexity of writing multi-threaded applications (with their synchronizations, deadlocks, etc.)

```
type Receive = PartialFunction[Any, Unit]

trait Actor {
  def receive: Receive
}
```

An actor has the following properties:

- It is an object with an identity
- It has a behavior
- It only interacts using asynchronous message

Note: to use Actors in Eclipse you need to run a Run Configuration whose main class is `akka.Main` and whose Program argument is the full Main class name

An actor can be used as follows:

```
import akka.actor._
```

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
    case ("get", customer: ActorRef) => customer ! count // '!' means sends the message 'count' to the customer
    case "get" => sender ! count // same as above, except sender means the sender of the message
  }
}
```

The Actor’s Context

The Actor type describes the behavior (represented by a Receive, which is a PartialFunction), the execution is done by its ActorContext. An Actor can change its behavior by either pushing a new behavior on top of a stack or just purely replace the old behavior.

```
trait ActorContext {
  def become(behavior: Receive, discardOld: Boolean = true): Unit // changes the behavior
  def unbecome(): Unit                                             // reverts to the previous behavior
  def actorOf(p: Props, name: String): ActorRef                    // creates a new actor
  def stop(a: ActorRef): Unit                                      // stops an actor
  def watch(target: ActorRed): ActorRef                            // watches whenever an Actor is stopped
  def unwatch(target: ActorRed): ActorRef                          // unwatches
  def parent: ActorRef                                             // the Actor's parent
  def child(name: String): Option[ActorRef]                        // returns a child if it exists
  def children: Iterable[ActorRef]                                 // returns all supervised children
}

class myActor extends Actor {
   ...
   context.parent ! aMessage // sends a message to the parent Actor
   context.stop(self)        // stops oneself
   ...
}
```

The following example is changing the Actor’s behavior any time the amount is changed. The upside of this method is that 1) the state change is explicit and done by calling `context.become()` and 2) the state is scoped to the current behavior.

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case "incr" => context.become(counter(n + 1))
    case "get" => sender ! n
  }
  def receive = counter(0)
}
```

Children and hierarchy

Each Actor can create children actors, creating a hierarchy.

```
class Main extends Actor {
  val counter = context.actorOf(Props[Counter], "counter")  // creates a Counter actor named "counter"

  counter ! "incr"
  counter ! "incr"
  counter ! "incr"
  counter ! "get"

  def receive = {       // receives the messages from Counter
    case count: Int =>
      println(s"count was $count")
      context.stop(self)
    }
  }
}
```

Each actor maintains a list of the actors it created:

- the child is added to the list when context.actorOf returns
- the child is removed when Terminated is received
- an actor name is available IF there is no such child. Actors are identified by their names, so they must be unique.

Message Processing Semantics

There is no direct access to an actor behavior. Only messages can be sent to known adresses (`ActorRef`). Those adresses can be either be oneself (`self`), the address returned when creating a new actor, or when received by a message (e.g. `sender`)

Actors are completely insulated from each other except for messages they send each other. Their computation can be run concurrently. However, a specific actor is single-threaded - its messages are received sequentially. Processing a message is the atomic unit of execution and cannot be interrupted.

It is good practice to define an Actor's messages in its companion object. Here, each operation is effectively synchronized as all messages are serialized.

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

class BankAccount extends Actor {
  import BankAccount._

  var valance = BigInt(0)

  def receive = {
    case Deposit(amount) => balance += amount
                            sender ! Done
    case Withdraw(amount) if amount <= balance => balance -= amount
                                                  sender ! Done
    case _ => sender ! Failed
  }
}
```

Note that `pipeTo` can be used to foward a message (`theAccount deposit(500) pipeTo sender`)

Because communication is through messages, there is no delivery guarantee. Hence the need of messages of acknowledgement and/or repeat. There are various strategies to deal with this:

- at-most-once: send a message, without guarantee it will be received
- at-least-once: keep sending messages until an ack is received
- exactly-once: keep sending messages until an ack is received, but the recipient will only process the first message

You can call `context.setReceiveTimeout(10.seconds)` that sets a timeout:

```
def receive = {
  case Done => ...
  case ReceiveTimeout => ...
}
```

The Akka library also includes a scheduler that sends a message or executes a block of code after a certain delay:

```
trait Scheduler {
  def scheduleOnce(delay: FiniteDuration, target: ActorRef, msg: Any)
  def scheduleOnce(delay: FiniteDuration)(block: => Unit)
}
```

Designing Actor Systems

When designing an Actor system, it is useful to:

- visualize a room full of people (i.e. the Actors)
- consider the goal to achieve

- split the goal into subtasks that can be assigned to the various actors
- who needs to talk to whom?
- remember that you can easily create new Actors, even short-lived ones
- watch out for any blocking part
- prefer immutable data structures that can safely be shared
- do not refer to actor state from code running asynchronously

Consider a Web bot that recursively download content (down to a certain depth):

- one Client Actor, which is sending download requests
- one Receptionist Actor, responsible for accepting incoming download requests from Clients. The Receptionist forwards the request to the Controller
- one Controller Actor, noting the pages already downloaded and dispatching the download jobs to Getter actors
- one or more Getter Actors whose job is to download a URL, check its links and tell the Controller about those links
- each message between the Controller and the Getter contains the depth level
- once this is done, the Controller notifies the Receptionist, who remembers the Client who asked for that request and notifies it

Testing Actor Systems

Tests can only verify externally observable effects. Akka's TestProbe allows to check that:

```
implicit val system = ActorSystem("TestSys")
val myActor = system.actorOf(Props[MyActor])
val p = TestProbe()
p.send(myActor, "Message")
p.exceptMsg("Ack")
p.send(myActor, "Message")
p.expectNoMsg(1.second)
system.shutdown()
```

It can also be run from inside TestProbe:

```
new TestKit(ActorSystem("TestSys")) with ImplicitSender {
  val myActor = system.actorOf(Props[MyActor])
  myActor ! "Message"
  expectMsg("Ack")
  send(myActor, "Message")
  expectNoMsg(1.second)
  system.shutdown()
}
```

You can use dependency injection when the system relies from external sources, like overriding factory methods that work as follows:

- have a method that will call `Props[MyActor]`
- its result is called by context.actorOf()
- the test can define a "fake Actor" (`object FakeMyActor extends MyActor { ... }`) that will override the method

You should start first the "leaves" actors and work your way to the parent actors.

Logging Actor Systems

You can mix in your actor with `ActorLogging`, and use various log methods such as `log.debug` or `log.info`.

To see all the messages the actor is receiving, you can also define `receive` method as a `LoggingReceive`.

```
def receive: Receive = LoggingReceive {
  case Replicate =>
  case Snapshot =>
}
```

To see the log messages turn on akka debug level by adding the following in your run configuration.

```
-Dakka.loglevel=DEBUG -Dakka.actor.debug.receive=on
```

Failure handling with Actors

What happens when an error happens with an actor? Where shall failures go? With the Actor models, Actors work together in teams (systems) and individual failures are handled by the team leader.

Resilience demands containment (i.e. the failure is isolated so that it cannot spread to other components) and delegation of failure (i.e. it is handled by someone else and not the failed component)

In the Supervisor model, the Supervisor needs to create its subordinates and will handle the exceptions encountered by its children. If a child fails, the supervisor may decide to stop it (`stop` message) or to restart it to get it back to a known good state and initial behavior (in Akka, the ActorRef stays valid after a restart).

An actor can decide a strategy by overriding `supervisorStrategy`, e.g.

```
class myActor extends Actor {
  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 5) {
     case _: Exception => SupervisorStrategy.Restart
  }
}
```

Lifecycle of an Actor

- An Actor will have its context create a child Actor, and gets `preStart()` called.
- In case of a failure, the supervisor gets consulted. The supervisor can stop the child or restart it (a restart is not externally visible). In case of a restart, the child Actor's `preRestart()` gets called. A new instance of the actor is created, after which its `postRestart()` method gets called. No message gets processed between the failure and the restart.
- An actor can be restarted several times.
- An actor can finally be stopped. It sends Stop to the context and its `postStop()` method will be called.

An Actor has the following methods that can be overridden:

```
trait Actor {
  def preStart(): Unit
  def preRestart(reason: Throwable, message: Option[Any]): Unit // the default behavior is to stop all children
  def postRestart(reason: Throwable): Unit                      // the default behavior is to call preStart()
  def postStop(): Unit
}
```

Lifecycle Monitoring

To remove the ambiguity where a message doesn't get a response because the recipient stopped or because the network is down, Akka supports Lifecycle Monitoring, aka DeathWatch:

- an Actor registers its interest using `context.watch(target)`
- it will receive a `Terminated(target)` message when the target stops
- it will not receive any direct messages from the target thereafter

The watcher receives a `Terminated(actor: ActorRef)` message:

- It is a special message that our code cannot send
- It comes with two implicit boolean flags: `existenceConfirmed` (was the watch sent when the target was still existing?) and `addressTerminated` (the watched actor was detected as unreachable)
- Terminated messages are handled by the actor context, so cannot be forwarded

The Error Kernel pattern

Keep important data near the root, delegate the risk to the leaves

- restarts are recursive
- as a result, restarts are more frequent near the leaves
- avoid restarting Actors with important states

EventStream

Because Actors can only send messages to a known address, the EventStream allows publication of messages to an unknown audience

```
trait EventStream {
  def subscribe(subscriber: ActorRef, topic: Class[_]): Boolean
  def unsubscribe(subscriber: ActorRef, topic: Class[_]): Boolean
  def unsubscribe(subscriber: ActorRef): Unit
  def publish(event: AnyRef): Unit
}



class MyActor extends Actor {
  context.system.eventStream.subscribe(self, classOf[LogEvent])
  def receive = {
    case e: LogEvent => ...
  }
  override def postStop(): Unit = {
```

```
    context.system.eventStream.unsubscribe(self)
  }
}
```

Unhanlded messages are passed to the Actor's `unhandled(message: Any)` method.

Persistent Actor State

The state of an Actor can be stored on disk to prevent data loss in case of a system failure.

There are two ways for persisting state:

- in-place updates mimics what is stored on the disk. This solution allows a fast recovery and limits the space used on the disk.
- persist changes in append-only fashion. This solution allows fast updates on the disk. Because changes are immutable they can be freely be replicated. Finally it allows to analyze the history of a state.
    - Command-Sourcing: persists the messages before processing them, persist acknowledgement when processed. Recovery works by sending the messages to the actor. A persistent Channel discards the messages already sent to the other actors
    - Event-Sourcing: Generate change requests (events) instead of modifying the local state. The events are sent to the log that stores them. The actor can either update its state when sending the event to the log or wait for the log to contact it back (in which case it can buffer any message while waiting for the log).
    - In both cases, immutable snapshots can be made at certain points of time. Recovery only applies recent changes to the latest snapshot.

Each strategy have their upsides and downsides in terms of performance to change the state, recover the state, etc.

The `stash` trait allows to buffer, e.g.

```
class MyActor extends Actor with Stash {
  var state: State = ...
  def receive = {
    case NewState(text) if !state.disabled =>
      ... // sends the event to the log
      context.become(waiting, discardOld = false)
  }
  def waiting(): Receive = {
    case e: Event =>
      state = state.updated(e)  // updates the state
      context.unbecome();       // reverts to the previous behavior
      unstashAll()              // processes all the stashed messages
    case _ => stash()           // stashes any message while waiting
  }
}
```

Created Wed 13 Nov 2013 8:59 PM MSK

Last Modified Mon 16 Dec 2013 11:14 PM MSK