Exercise 1 : Implementing `map` and `filter` on Futures

In this exercise, you will come up with an implementation of the `map` and `filter` methods of Futures. First of all, spend some time as a group to make sure that you understand what those methods are supposed to do. Then, complete the following code to implement the two methods:

```
trait Future[T] { self =>
  def map[S](f: T => S): Future[S] =
    new Future[S] {
      def onComplete(callback: Try[S] => Unit): Unit = ???
    }

  def filter(f: T => Boolean): Future[T] =
    new Future[T] {
      def onComplete(callback: Try[T] => Unit): Unit = ???
    }
}
```

Solution

```
trait Future[T] { self =>
  def map[S](f: T => S): Future[S] =
    new Future[S] {
      def onComplete(callback: Try[S] => Unit): Unit = self.onComplete {
        case Success(v) => callback(Success(f(v)))
        case Failure(e) => callback(Failure(e))
      }
    }

  def filter(f: T => Boolean): Future[T] =
    new Future[T] {
      def onComplete(callback: Try[T] => Unit): Unit = self.onComplete {
        case Success(v) =>
         if f(v) {
           callback(Success(v))
          }
          else {
           callback(Failure(new NoSuchElementException("...")))
          }
        case Failure(e) => callback(Failure(e))
      }
    }
}
```

Exercise 2 : The Josephus Problem

In this exercise, we will revisit the famous *Josephus problem*. In this problem, a group of soldiers trapped by the enemy decide to commit suicide instead of surrendering. In order not to have to take their own lives, the soldiers devise a plan. All soldiers are arranged in a single circle. Each soldier, when comes their turn to act, has to kill the next soldier alive next to them in the clockwise direction. Then, the next soldier that is still alive in the same direction acts. This continues until there remains only one soldier in the circle. This last soldier is the lucky one, and can surrender if he decides to. The *Josephus problem* consists in finding the position in the circle of this lucky soldier, depending on the number of soldiers.

In this exercise, you will implement a *simulation* of the mass killing of the soldiers. Each soldier will be modeled by an actor. Soldiers are arranged in a circle and when their turn comes to act, they kill the next alive soldier in the circle. The next soldier that is still alive in the circle should act next. The last soldier remaining alive does not kill himself but prints out its number to the standard output.

The code on next page covers the creation of all actors and the initialisation of the system. Your goal is to implement the `receive` method of the actors.

Hint: Think about what state the soldier must have.

Solution

```
import akka.actor._

class Soldier(number: Int) extends Actor {

  import Soldier._

  def receive: Receive = behavior(None, None, false)

  def behavior(next: Option[ActorRef],
               killer: Option[ActorRef],
               mustAct: Boolean): Receive = {

    case Kill => next match {
      case Some(myNext) => {
        sender ! Next(myNext)
        myNext ! Act
        println("Soldier " + number + " dies.")
        self ! PoisonPill
      }
      case None => {
        context.become(behavior(None, Some(sender), mustAct))
      }
    }

    case Next(newNext) => {
      if (newNext == self) {
        println("Soldier " + number + " is last !")
      }
      else if (!killer.isEmpty) {
        killer.get ! Next(newNext)
        newNext ! Act
        println("Soldier " + number + " dies.")
        self ! PoisonPill
      }
      else if (mustAct) {
        newNext ! Kill
        context.become(behavior(None, None, false))
      }
      else {
        context.become(behavior(Some(newNext), None, false))
      }
    }
    case Act => next match {
      case Some(myNext) => {
        myNext ! Kill
        context.become(behavior(None, killer, false))
      }
      case None => {
        context.become(behavior(None, killer, true))
      }
    }
  }
}


object Soldier {
  // The different messages that can be sent between the actors:

  // The recipient should die.
  case object Kill

  // The recipient should update its next reference.
  case class Next(next: ActorRef)

  // The recipient should act.
  case object Act

  def props(number: Int): Props = Props(new Soldier(number))
}


object Simulation {

  import Soldier._

  // Initialization
  val system = ActorSystem("mySystem")

  def start(n: Int) {
    require(n >= 1)

    // Creation of the actors.
    val actors = Seq.tabulate(n) { (i: Int) =>
      system.actorOf(Soldier.props(i), "Soldier" + i)
    }

    // Inform all actors of the next actor in the circle.
    for (i <- 0 to (n - 2)) {
      actors(i) ! Next(actors(i + 1))
    }
    actors(n - 1) ! Next(actors(0))

    // Inform the first actor to start acting.
    actors(0) ! Act
  }
}
```