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 : Master / Slave

In this exercise, you will have to implement a Master / Slave actor system, in which one actor, the *master*, dispatches work to other actors, the *slaves*. Between the master and the slaves, only two kinds of messages are sent: `Order` and `Ready` messages.

```
case class Order(computation: => Unit)
case object Ready
```

The master actor sends `Order` messages to slaves to order them to perform some computation (passed as an argument of `Order`). Upon reception of an `Order`, a slave should perform the computation. Slaves should send a `Ready` message to their master whenever they finish executing the requested computation, and right after they are created.

The master actor itself receives requests through `Order` messages from clients. The master actor should then dispatch the work to slave actors. The master should however never send an order to a slave which has not declared itself ready via a `Ready` message beforehand.

Implement the `Master` and `Slave` classes.

Solution on the next page.

Solution

```
class Master extends Actor {
  var availableSlaves: List[ActorRef] = Nil
  var pendingOrders: List[Order] = Nil

  def receive: Receive = {
    case Ready =>
      if (pendingOrders.isEmpty) {
        availableSlaves = availableSlaves :+ sender
      }
      else {
        val order = pendingOrders.head
        pendingOrders = pendingOrders.tail
        sender ! order
      }
    case order: Order => availableSlaves match {
      case slave :: rest => {
        slave ! order
        availableSlaves = rest
      }
      case Nil => {
        pendingOrders = pendingOrders :+ order
      }
    }
  }
}

class Slave(master: ActorRef) extends Actor {
  master ! Ready

  def receive: Receive = {
    case Order(f) =>
      f()
      master ! Ready
  }
}
```