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 = ???
    }
}
```

In the case of filter, if the original `Future` successfully returns a value which does not satisfy the predicate, the new `Future` should return a `Failure` containing a `NoSuchElementException`.

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.

```
class Master extends Actor {
  ???

  override def receive = ???
}

class Slave(master: Master) extends Actor {
  ???

  override def receive = ???
}
```

An example system using the `Master` and `Slave` actors is shown below.

```
object MasterSlave extends App {
  val masterProps: Props = Props(new Master())
  def slaveProps(master: Master): Props = Props(new Slave(master))

  val system = ActorSystem("master/slave")

  val master = system.actorOf(masterProps)
  val slaves = Seq.fill(10) {
    system.actorOf(slaveProps(master))
  }

  // Now, clients should be able to send requests to the master...
  master ! Order(println(3 + 5))
  master ! Order(println(67 * 3))
  // And so on...
}
```

Hint: In order to fulfill its job, the master should remember which slaves are ready and what requests are still to be allocated to a slave.