

Problem 1: Implementing map and filter on Futures

In this exercise, you will come up with an implementation of the `map` and `filter` methods of `MyFuture`. The `MyFuture` trait is a simplified version of the `Future` trait from the Scala standard library, with a single abstract method:

```
trait MyFuture[+T]:
  def onComplete(callback: Try[T] => Unit): Unit
```

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:

```
extension [T](self: MyFuture[T])
  def map[S](f: T => S): MyFuture[S] = ???
  def filter(p: T => Boolean): MyFuture[T] = ???
```

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

Problem 2: Coordinator / Worker

In this exercise, you will implement a `Coordinator / Worker` actor system, in which one actor, the coordinator, dispatches work to other actors, the workers. Between the coordinator and the workers, only two kinds of messages are sent: `Request` and `Ready` messages.

```
enum Message:
  case Request(computation: () => Unit)
  case Ready
```

The coordinator actor sends `Request` messages to workers to request them to perform some computation (passed as an argument of `Request`). Upon reception of a `Request`, a worker should perform the computation. Workers

should send a `Ready` message to their coordinator whenever they finish executing the requested computation, and also right after they are created.

The coordinator actor itself receives requests through `Request` messages from clients. The coordinator actor should then dispatch the work to worker actors. The coordinator should however never send a request to a worker which has not declared itself ready via a `Ready` message beforehand.

Implement the `Coordinator` and `Worker` classes.

```
class Coordinator extends Actor:
  ???
  override def receive = ???

class Worker(coordinator: Coordinator) extends Actor:
  ???
  override def receive = ???
```

An example system using the `Coordinator` and `Worker` actors is shown below.

```
@main def problem2 = new TestKit(ActorSystem("coordinator-workers"))
  with ImplicitSender:
    try
      val coordinator = system.actorOf(Props(Coordinator()))
      val workers = Seq.tabulate(4)(i =>
        system.actorOf(Props(Worker(coordinator)))
      )

      // Now, clients should be able to send requests to the
      coordinator...
      coordinator ! Request(() => println(3 + 5))
      coordinator ! Request(() => println(67 * 3))
      // And so on...
      finally shutdown(system)
```

Hint: In order to fulfill its job, the coordinator should remember which workers are ready and what requests are still to be allocated to a worker.