Note: You might find it useful to take a look at the following:

Papers on *Java executors and futures, Scala actors, and Scala futures* can be found in [http://web.cs.iastate.edu/~smkautz/cs430s16/papers/](http://web.cs.iastate.edu/~smkautz/cs430s16/papers/)

In addition to those included in the directory above, there are some additional simple Scala examples that might be useful to try:

- Speak.scala - Steff's Cat and Dog example with Akka actors.
- InputTest.scala - Reading lines from standard input
- SortedRecordTest.scala - using the Ordered trait to sort a list of Record objects
- TestClient.scala - making a socket connection to the SimpleServer

These can be found in [http://web.cs.iastate.edu/~smkautz/cs430s16/examples/scala/](http://web.cs.iastate.edu/~smkautz/cs430s16/examples/scala/)

Remember that the examples and links from the original Scala presentation can be found here, [http://web.cs.iastate.edu/~smkautz/cs430s16/scala_references.html](http://web.cs.iastate.edu/~smkautz/cs430s16/scala_references.html).

In order to use Akka actors, you'll need to add two additional jars, `akka-actor_xxxx.jar` and `config-xxxx.jar`, to the build path. If you have the latest Scala distribution (2.11.7, I think), you can find them in the lib directory.

1. Take the original sample code for hw3a and reimplement it in Scala using Akka actors for the four components. (This means just translate the existing code, you don't have to actually re-do hw3a in Scala.) Use Scala libraries and idioms wherever possible (e.g., use Scala collections rather than Java collections, etc.). Use case classes for the message types. It is ok to create a new actor on the fly for the ProxyWorker. In the client, you don't have to exactly reproduce all the helper methods if you want to streamline it a bit, as long as the behavior is the same. Note, you will not need anything analogous to AbstractMessage, Component, SetTimeoutMessage, ThreadedComponent, or TimeoutMessage.

2. See the example Pi.java, also in the scala examples directory. It is a simple Master-worker problem to perform a randomized approximation of pi in parallel. Re-do this problem using Scala futures, in two ways:

   a) blocking, as in the Java version
b) with callbacks

_Important note on Random:_ There are two ways to generate random numbers in Scala.

a) Use the system-wide Random object

```scala
val rand = scala.util.Random
```

b) or, create a new Random object

```scala
val rand = new scala.util.Random()
```

When using multiple threads or a thread pool, it is critical to do (b), i.e., make sure each thread has its own instance. (Once you get the problem done, try using (a) and see how it performs.)