Name: _____ Vorname: _____

Matrikelnummer: _____ Testat:_____

Paradigmen der Programmierung
Praktikum 3

Hinweise für die Informationssuche:

1. Schauen Sie ins Skript und in die Vorlesungsfolien
2. Nutzen Sie das Buch „Actors ins Scala“: http://www.softwareresearch.net/fileadmin/src/docs/teaching/SS13/ST/ActorsInScala.pdf
3. Fragen Sie während des Praktikums
4. Nutzen Sie die Dateien Broadcaster.scala, ListCalculations.scala, MapListProcessor.scala und Main.scala (auf www.kohls.de/lehre oder hier als ZIP-Datei: http://www.kohls.de/wp-content/uploads/2014/11/listactors.zip)

In dieser Praktikumsaufgabe geht es um die Entwicklung nebenläufiger Berechnungen von Teillisten mit Hilfe von Actors.

Um einen Actor zu erzeugen, können Sie eine neue Klasse entwerfen, die von Actors abgeleitet ist, und Objekte dieser Klasse erzeugen. Sie können auch direkt anonyme Actor-Objekte erzeugen:

```
val neuerActor = actor {
  while (true) {
     receive {
     case msg  => // do something with msg
     }
  }
}
```

Die Klasse ListCalculations ist ein Beispiel für eine Actor-Implementierung. Die Klasse enthält zwei Listen `data` und `resultData`. Die Werte aus `data` sollen nach `resultData` gemappt werden. Objekte der Klasse empfangen Nachrichten, in denen festgelegt wird

- mit welcher Rechenvorschrift das Mapping geschehen soll,
- ob das Mapping nebenläufig geschehen soll.

`ListCalculations`-Objekte reagieren als Actors auf verschiedene Nachrichten:

- Wenn Int Objekte empfangen werden, dann werden diese der Liste `data` vorangestellt. Zuvor wird der empfangene Int-Werte aber noch über eine modifier-Funktion verändert.
- Die Case-Klassen `SimpleCalculation, AsyncCalculation` und `FutureCalculation` legen fest, mit welchem Verfahren (nebenläufig oder nicht) die Berechnung durchgeführt werden soll und überliefern zudem die Rechenvorschrift (durch konkrete Implementierungen der `process`-Funktion der abstrakten Klasse `MapListProcessor`).

Aufgabe 1 (Akteure und Nachrichten erzeugen)
Das `Broadcaster`-Objekt ist ein Actor und empfängt Nachrichten, um diese wiederum an mehrere andere Actors zu verteilen. Dies wird genutzt, um mehrere `ListCalculations`-Objekte mit den gleichen Nachrichten zu versorgen.

Im `Main`-Objekt werden bereits zwei Actors `test1` und `test2` erzeugt und beim `Broadcaster`-Objekt registriert (über eine `Register` Nachricht).

Erstellen Sie einen dritten Actor `test3` und verwenden Sie eine eigene modifier-Funktion, um die eingehenden Int-Werte der Liste zu verändern. Durch die Registierung beim `Broadcaster`-Objekt empfängt `test3` die gleichen Nachrichten wie `test1` und `test2`. Was geschieht dabei?

Senden sie weitere Nachrichten mit einem anderen `MapListProcessor` an das `Broadcaster`-Objekt.

Aufgabe 2 (Nebenläufiges Mappen einer Liste durch mehrere Actors)
Die `receive`-Funktion von `ListCalculations` hat bisher nur nicht-nebenläufige Berechnung implementiert (`SimpleCalculation`), d.h. die gesamte Liste `data` wird an den `MapListProcessor` geschickt. Dies geschieht zudem sehr unsauber (was ist hier besonders kritisch?).

Implementieren Sie nebenläufige Berechnungen, indem Sie die Gesamtliste in Teillisten aufteilen und jede Teilliste durch einen Actor berechnen lassen. Sie sollen also sinnvolle Funktionen für Reaktion auf die Nachrichten `AsyncCalculation` und/oder `FutureCalculation` definieren. Die mit `AsyncCalculation` oder `FutureCalculation` übermittelten Nachrichten enthalten einen `MapListProcessor` und die Anzahl der Actors, die gleichzeitig Teilergebnisse berechnen sollen.

Tipp: Eine Liste lässt sich mit splitAt an einer bestimmten Position in zwei Teile aufteilen.

Hinweis: die map-Funktion der List-Implementierung von Scala ist sicherlich effizienter als unsere Parallelisierungsstrategie. Hier geht nur um das generelle Verständnis, wie eine Berechnung parallel durchgeführt werden kann.

Aufgabe 3 (Leichtgewichtige Actors mit react)
Tipp: Speichern Sie alle offenen Dokumente und Dateien bevor Sie mit dieser Aufgabe beginnen!

Schreiben Sie ans Ende der `main`-Funktion eine for-Schleife, in der sehr viele Actors (z.B. `ListCalculation`-Objekte) erzeugt werden. Da wir bislang die Actors „schwergewichtig“ implementiert haben, wird für jeden Actor ein eigener Thread gestartet. Wann stößt der Rechner an seine Ressourcengrenzen?

Sie können eine leichtgewichtige (event-basierte) Implementierung der Actors erreichen, indem Sie die `receive` durch `react` ersetzen. Was muss dabei beachtet werden?