

Lösungsansatz Aufgabe 1

Die Modifier-Funktion kann beliebig einfach sein, z.B.:

```
val test3 = new ListCalculations( "Dreier" , x => x *3)
val test4 = new ListCalculations( "Negativ" , x => -x)
```

Damit `test3` auch über Nachrichten informiert wird, müssen Sie ihn ebenfalls beim `Broadcaster` registrieren:

```
Broadcaster ! Register (test3)
```

Sie können weitere Nachrichten mit `MapListProcessors` an den `Broadcaster` senden:

```
val lp2 = new MapListProcessorAdd100()
Broadcaster ! BroadcastMsg (SimpleCalculation (lp2 ))
Broadcaster ! BroadcastMsg (AsyncCalculation (lp2, 4) )
Broadcaster ! BroadcastMsg (FutureCalculation (lp2, 4) )
```

Da sowohl `test1`, `test2` und `test3` beim `Broadcaster` registriert sind, empfangen sie alle auch den Auftrag das Resultat für ihre Daten mit `MapListProcessorAdd100` zu berechnen.

`MapListProcessorAdd100` ist ein Beispiel für eine zusätzliche Implementierung des `MapListProcessor`:

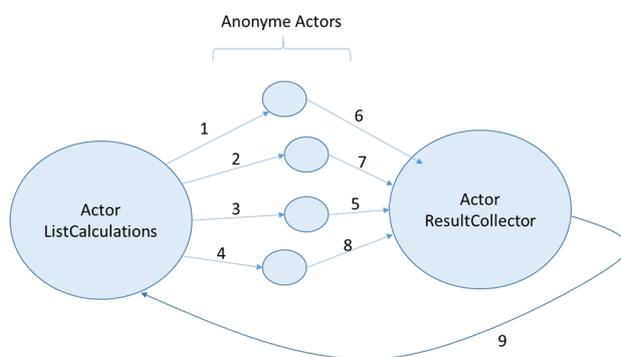
```
class MapListProcessorAdd100 extends MapListProcessor {
  override def process (l : List[Int]) : List[Int] = for (e1 <- l) yield e1 + 100
  override val name = "Add 100"
}
```

Lösungsansatz Aufgabe 2 für asynchrones Abarbeiten

Die Lösung teilt sich auf in folgende Schritte:

- Die Gesamtliste `data` muss aufgeteilt werden in Teillisten
- Für jede Teilliste muss ein eigener Actor gestartet werden, der jeweils diese Teilliste berechnet
- Für das Sammeln der Teillisten wird ein weiterer Actor benötigt. Aufgrund der parallelen Berechnung treffen die Ergebnisse der Teillisten in asynchroner Reihenfolge ein. Es kann sein, dass die erste Teilliste als letztes vorliegt
- Wenn alle Teilergebnisse vorliegen, dann können sie zum Gesamtergebnis zusammengesetzt werden. Die Ausgabe kann entweder direkt erfolgen oder man sendet eine Nachricht an den `ListCalculation Actor`.

Schematisch sieht dies so aus:



`ListCalculations` empfängt die Nachricht `AsyncCalculation` und verteilt die Gesamtliste auf Teilliste. Jede Teilliste wird der Reihe nach (1,2,3,4) an anonyme Actors geschickt. Diese haben nur die Aufgabe die Teilliste zu berechnen. Sobald das Ergebnis vorliegt wird es an `ResultCollector` geschickt. Aufgrund der asynchronen Berechnung treffen die Teillisten in unterschiedlicher Reihenfolge bei `ResultCollector` ein (5,6,7,8). Wenn alle Teilergebnisse vorliegen werden diese zu einer Gesamtliste zusammengesetzt und zurück an `ListCalculations` geschickt (9) oder direkt ausgegeben.

Zunächst benötigen wir einen `ResultCollector`, der das Ergebnis sammelt. Auch der `ResultCollector` ist ein Actor. Wir erzeugen ihn sobald wir die Nachricht erhalten, die Liste asynchron zu bearbeiten:

```
case ac: AsyncCalculation => {  
  val resultCollector = new ResultCollector (ac.numberOfActors , ac.lp.name , this)  
  resultCollector.start()  
  ...  
}
```

Zum Aufsplitten der Gesamtliste gehen wir hier iterativ vor (auch wenn dies nicht der funktionalen Herangehensweise von Scala entspricht):

```
val segmentSize = if (ac.numberOfActors > 0 && ac.numberOfActors <= data.length)
    data.length / ac.numberOfActors else data.length;

var tmpData = data;
var position = 0;

while (tmpData.length > segmentSize && position < ac.numberOfActors - 1) {
    val split = tmpData.splitAt(segmentSize)
    // Erster Teil wird zum Berechnen geschickt:
    startAsyncCalc ( split._1 , position , ac.lp , resultCollector )

    // zweiter Teil wird nun Rest der Liste:
    tmpData = split._2
    position += 1
}
// Restliste auch noch berechnen:
startAsyncCalc ( tmpData , ac.numberOfActors - 1 , ac.lp , resultCollector )
```

Wir senden also jeweils Teillisten und deren korrekte Position an die Methode `startAsyncCalc(..)`. Darin wird der eigentliche Actor erzeugt:

```
private def startAsyncCalc(partList : List [Int] , position : Integer, mlp:
MapListProcessor , collector : ResultCollector ) {

    val neuerActor = actor {
        val partResult = mlp.process(partList);
        collector ! PartResult (partResult , position)
    }
}
```

Der Actor beginnt sofort zu arbeiten. Er blockiert dabei nicht, so dass die Methode direkt zurückspringt und weitere Teilaufträge platziert werden können.

Wenn aus der nebenläufigen Berechnung das Ergebnis vorliegt, dann wird unser `ResultCollector` benachrichtigt:

```
collector ! PartResult (partResult , position)
```

In der Klasse `ResultCollector` definieren wir die `act()` Methoden und reagieren auf die eben verschickte `PartResult` – Nachricht:

```
def act () {

    while (missingResults > 0) {

        receive {
            case res : PartResult => {
                collector(res.position) = res.data ;
                missingResults -= 1
            }
        }
    }
}
```

Solange nicht alle Resultate eingetroffen sind warten wir auf weitere Nachrichten. Erst wenn `missingResults` den Wert 0 erreicht wird nicht auf weitere Nachrichten gewartet.

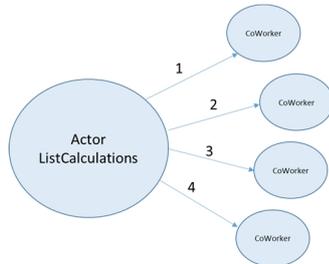
Die in `collector` gesammelten Teilergebnisse werden jetzt in eine flache Liste umgewandelt und danach als Gesamtergebnis an den `ListCalculations`-Actor (hier via `notifyOnResult` referenziert) als Nachricht übermittelt:

```
val fullResult = collector.toList.flatten;
notifyOnResult ! setResultList (fullResult , listName)
```

Lösungsansatz Aufgabe 2 für die Arbeit mit Futures

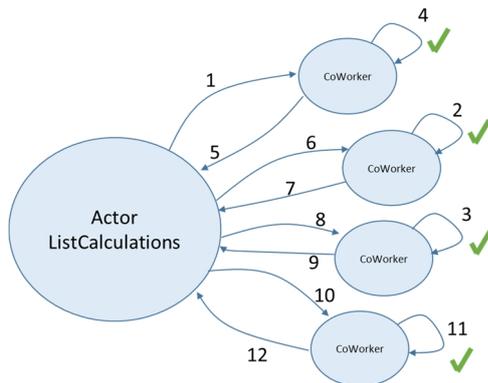
Auch bei dieser Lösung wird die Gesamtliste in Teillisten aufgeteilt. Zusätzlich wird aber eine Liste mit Future-Objekten erstellt, über die dann später das Ergebnis abgerufen wird. Die Berechnungen können nebenläufig erfolgen, aber beim Zugriff auf das Ergebnis wird blockiert falls dieses noch nicht vorliegt.

Schematisch sieht dies so aus:



Zunächst wird die Arbeit auf mehrer Actors verteilt (1,2,3,4). Dabei verwenden wir in diesem Falle eine neue Klasse `CoWorker`, die von `Actor` abgeleitet ist.

Sobald alle `CoWorker` mit Arbeit versorgt sind, also die Berechnung aller Teillisten begonnen hat, können über die `Future`-Objekte die Teilergebnisse abfragen. Dies geschieht in der gleichen Reihenfolge.



Für die erste Ergebnisabfrage (1) wird solange gewartet bis das Ergebnis (4) vorliegt. Zwar sind andere `CoWorker` sehr viel schneller fertig (2,3), aber das Ergebnis wird noch nicht abgefragt. Sobald das angefragte Teilergebnis dann vorliegt (4) kann es von `ListCalculations` verwendet werden (5). Danach wird das nächste Ergebnis abgefragt (6). Dieses liegt schon lange vor (2) und kann daher sofort geliefert werden (7). Gleiches gilt für die nächste Anfrage (8). Sie kann sofort beantwortet werden (9), da das Ergebnis schon vorher verfügbar war (3). Beim letzten Future (10) liegt das Ergebnis noch nicht vor. Sobald es vorliegt (11) wird es auch zurück geliefert (12).

Wichtig ist dabei folgendes: Die Ergebnisberechnung erfolgt weiterhin parallel und asynchron (2,3,4,11), d.h. die Ergebnisse liegen nicht in der Reihenfolge vor, in der die `CoWorker` gestartet worden sind.

Sobald die Bearbeitung jedoch angestoßen ist, erfolgt die Abfrage der Ergebnisse in synchroner Reihenfolge. Die Abfrage der Ergebnisse erfolgt z.B. in (1,6,8,10), die `CoWorker` antworten in gleicher Reihenfolge (5,7,9,12).

Im Code sieht dies dann wie folgt aus. Zunächst nutzen wir eine Liste mit Futures:

```
var futures : List [Future[Any]] = List();
```

Beim Verteilen der Aufgaben erzeugen wir eine neue Liste mit weiteren Futures, daher ist `futures` variabel. Da ein Actor mit verschiedenen Nachrichtentypen antworten kann, speichern wir `Future[Any]` in der Liste. Der Cast auf unseren eigentlichen Antwortdatentyp, nämlich `List[Int]` erfolgt dann später.

Für jede Teilliste starten wir einen `CoWorker`. Dem `CoWorker` geben wir die Teilliste und einen `MapListProcessor`. Der `CoWorker` arbeitet los sobald er die `Task`-Nachricht erhält. Da wir noch nicht auf das Ergebnis zugreifen wird noch nicht blockiert.

```
val cw = new CoWorker()  
val fut = cw !! Task ( splitted._1 , fc.lp)
```

Damit wir später auf das Ergebnis zugreifen können, müssen wir uns alle Futures in einer Liste merken, dazu erzeugen wir eine neue Liste mit dem zusätzlichen Future-Objekt:

```
futures = futures:+fut;
```

Sobald alle Futures erzeugt und die Berechnungen gestartet sind können wir mit dem Einsammeln der Werte beginnen. Für jedes Future-Objekt fragen wir das Ergebnis mit `future()` ab. Der Aufruf blockiert solange bis das Ergebnis auch wirklich vorliegt. Da wir die futures in der Liste in der richtigen Reihenfolge gespeichert haben, müssen wir uns nicht mehr um die richtige Positionierung kümmern. Da Scala den Ergebnistyp eines Futures nicht kennen kann (je nach Nachricht könnten unterschiedliche Werte zurückgeschickt werden) müssen wir hier einen expliziten Cast vornehmen:

```
val preRes = for ( future <- futures ) yield future().asInstanceOf [List[Int]]
```

Für jedes `future`-Objekt in `futures` wurde also `future()` aufgerufen und als `List[Int]` behandelt. Mit `yield` sammeln wir die Ergebnisse der einzelnen Schleifendurchläufe wiederum in einer Liste. Somit enthält `preRes` nun eine Liste mit den Teilergebnissen (jeweils wieder Listen). Jetzt machen wir daraus eine flache Liste:

```
val res = preRes.flatten // Aus ( (1,2) , (3,4) ) wird z.B. (1,2,3,4)
```

Wir müssen uns bei den Futures also nicht um die richtige Positionierung der Teilergebnisse kümmern, da garantiert ist, dass die Ergebnisse in der richtigen Reihenfolge abgefragt werden – auch wenn nicht bekannt ist, in welcher Reihenfolge die Ergebnisse eintreffen.