Scala Übung 3: Generics

1) In dieser Aufgabe soll eine Queue implementiert werden. Die Queue soll persistent (unveränderbar) sein, d.h. die Methoden `enqueue` und `dequeue` sollen jeweils eine neue Queue-Instanz zurückliefern.

   Gehen Sie wie folgt vor:

   a) Definieren Sie die Klasse `Queue` mit einem Typparameter `A`.

   b) Definieren Sie in dieser Klasse einen Konstruktor, der ein Feld vom Typ `List[A]` deklariert und mit dem die Elementliste gesetzt werden kann. Listen vom Typ `List` sind immutable.

   c) Deklarieren Sie den Konstruktor privat:
   ```
   class Queue[A] private (…) { … }
   ```

   d) Definieren Sie ein „Companion-Objekt“ mit einer parametrisierten Methode `apply` mit einem Typparameter `A` und Parametertyp `A*` (steht für eine beliebige Anzahl Argumente, wie `A…` in Java). Ein Parameter vom Typ `A*` kann mit der Methode `toList` in eine Liste vom Typ `List[A]` konvertiert werden. Eine neue Queue kann somit mit der Anweisung
   ```
   scala> Queue(1,2,3)
   res0: Queue[Int] = Queue@3c1908c8
   ```
   erzeugt werden.

   e) Definieren Sie eine Methode `dequeue` welche ein Tupel mit dem ersten Element und einer neuen Queue ohne das erste Element zurückgibt. Auf das erste Element einer Liste kann mit der Methode `head` zugegriffen werden, auf den Rest mit der Methode `tail`. Falls die Methode `dequeue` auf einer leeren Queue aufgerufen wird, so soll eine `NoSuchElementException` geworfen werden (wird automatisch geworfen falls auf einer leeren Liste `head` aufgerufen wird).

   f) Verändern Sie den Typparameter so, dass eine Instanz von `Queue[Bird]` einer Variablen vom Typ `Queue[Animal]` zugewiesen werden kann:
   ```
   val q: Queue[Animal] = Queue[Bird]()
   ```

   g) Definieren Sie eine Methode `enqueue` welche das neue Element an das Ende der Queue anfügt und als Resultat diese neue Queue zurückgibt. Tipp: Lower Bounds! Für das Anfügen eines Elementes an eine Liste kann die Methode `:+` verwendet werden.

   Die klasse kann dann wie folgt verwendet werden:

```
scala> val q = Queue[Bird]()
q: Queue[Bird] = Queue@5c50d2c4

scala> val q1 = q.enqueue(new Bird)
q1: Queue[Bird] = Queue@7fa00fdf

scala> val q2 = q1.enqueue(new Animal)
q2: Queue[Animal] = Queue@4fe93738

scala> q2.dequeue
res1: (Animal, Queue[Animal]) = (Bird@2551e7fb,Queue@245522e1)
```

2) Gegeben sind die beiden folgenden Implementierungen für eine unveränderbare und eine veränderbare Klasse Pair:

```
// immutable:
class Pair[+T,+S](val first: T, val second: S) {
    def replaceFirst[U >: T](newFst: U): Pair[U,S] = new Pair(newFst, second)
    def replaceSecond[U >: S](newSnd: U): Pair[T,U] = new Pair(first, newSnd)
}

// mutable:
class PairM[T,S](private[this] var f: T, private[this] var s: S) {
    def first = f
    def second = s
    def replaceFirst (newFst: T) { f = newFst }
    def replaceSecond (newSnd: S) { s = newSnd }
}
```

Aufgaben:

a) Definieren Sie eine generische Methode `swap`, die ein unveränderbares `Pair` nimmt und als Resultat ein neues `Pair` mit vertauschten Komponenten zurückliefert.

b) Definieren Sie eine analoge generische Methode für veränderbare Pairs welches den swap inplace ausführt, also kein neues Objekt erzeugt. Überlegen Sie sich dabei zuerst, wie die Signatur dieser Methode aussehen muss.

c) Die Typparameter der Klasse `PairM` können nicht kovariant definiert werden. Geben Sie ein Beispiel an das aufzeigt, dass es nicht typsicher wäre, wenn die Klasse `PairM` kovariant deklariert wäre.

d) Zeigen Sie analog mit einem Gegenbeispiel dass die Typparameter auch nicht kontravariant deklariert werden können.

Ausgabe: 23. 05. 2018
Abgabe: 30. 05. 2018