Spezielle Kapitel Sozialer Web – Technologien - Schleifen

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Higher Order Functions
  - Anonyme Funktionen, Anwendung in der List-Klasse
  - Generisches map, filter
  - flatten und flatMap
- For-Schleifen
  - Aufbau
  - Beispiele
  - Umsetzung mittels map, flatMap und filter
- Neue Datentypen
  - Dictionaries/Map
  - Option

Higher Order Functions

- Funktionen sind in Funktionalen Programmiersprachen “first class values”
- Funktionen können wie jeder andere Wert als Parameter einer Funktion übergeben werden und können auch Ergebnis einer Funktion sein

Funktionen, die Funktionen als Parameter haben oder Funktionen zurück geben werden *Higher Order Functions* genannt

Realisierung über Vererbung

- Aufteilung der Funktion in einen variablen Anteil und einen festen Anteil
  - Fest: Durchlaufen der Liste
  - Variabel: Funktion, die den aktuellen Wert quadriert, … (Das ist eine Funktion, die ein Integer auf einen Integer abbildet)

```
abstract class Function{

   def apply(i:Int):Int
}
```

```
class id extends Function{

   def apply(i:Int):Int= i
}
```

```
class square extends Function{

   def apply(i:Int):Int= i*i
}
```

```
class cube extends… {

   def apply(i:Int)…
}
```

Fixer Anteil

```
def map(l:List[Int], f:Function[Int,Int]):List[Int]= l match{
   case Nil => Nil
   case x::xs => f.apply(x)::map(xs,f)
 }
```

Verdoppeln der Listelemente: map (l, new double)

Quadrieren der Listelemente: map (l, new square)

Schlüsselwort apply kann beim Aufruf weggelassen werden!

Funktionstypen in Scala

- Der Typ A => B ist eine Funktion, die ein Argument vom Typ A bekommt und ein Ergebnis vom Typ B liefert

Beispiel:

```
def square(x:Int):Int=x*x
def map(l:List[Int], f:Int=>Int):List[Int]= l match{
   case Nil => Nil
   case x::xs => f.apply(x)::map(xs,f)
 }
```

- Die Funktion map bekommt als zweiten Parameter eine Funktion vom Typ Int => Int – also eine Funktion die ein Integer auf einen Integer abbildet
- Aufruf map(List(1,2,3,4),square)
- Problem im Beispiel – Code wird durch die separate Funktion aufgebläht

Syntax von Anonymen Funktionen

Beispiel: Funktion zur Berechnung eines Cubes (x^3)

(x: Int) => x * x * x

(x: Int) ist der Parameter der Funktion

x*x*x ist die Berechnungsfunktion (der Body)

- Der Typ der Funktion kann weggelassen werden – er wird vom Compiler aus dem Kontext bestimmt
- Wenn mehr als ein Parameter in der Funktion ist, so werden diese durch ein Komma getrennt

  z.B.(x: Int, y: Int) => x + y

Anonyme Funktionen sind syntaktischer Zucker

Jede Anonyme Funktion $(x_1 : T_1,\ldots,\ x_n : T_n) \Rightarrow E$ kann ausgedrückt werden als normale Definition:

$$\text{def } f\ (x_1 : T_1,\ldots,\ x_n : T_n) = E;\ f$$

wobei f ein name ist, der bisher noch nicht benutzt wurde.

Anonyme Funktionen bieten also keine zusätzliche Funktionalität sondern sind nur syntaktischer Zucker

Übungsaufgabe map-Funktion

- Klasse Cons:

```
def mapI(f:Int=>Int):IntList = Cons(f(head),tail.mapI(f))
```

- Klasse Empty:

```
def mapI(f:Int=>Int):IntList = this
```

- Klasse IntList:

```
def mapO(f:Int=>Int, l:IntList):IntList = l match{
    case Cons(elem,rest) => Cons(f(elem), mapO(f,rest))
    case Empty => Empty
}
```

Übungsaufgabe filter-Funktion

- Klasse Cons:

```
def filterI(f:Int=>Boolean) = f(head) match{
    case true => Cons(head,tail.filterI(f))
    case _ => tail.filterI(f)
}
```

- Klasse Empty:

```
def filterI(f:Int=>Boolean)=this
```

- Klasse IntList:

```
def filterO(f:Int=>Boolean, l:IntList):IntList = l match{
    case Cons(elem,rest) if f(elem) => Cons(elem,filterO(f,rest))
    case Cons(_,_) => filterO(f,l.tail)case _ => Empty
}
```

Übungsaufgabe reduce-Funktion

- Klasse Cons:

```
def reduceIn(reduceFun: (Int, Int)=>Int): Int= tail match{
    case Empty => head
    case _ => reduceFun(head,tail.reduceIn(reduceFun))
}
```

- Klasse Empty:

```
def reduceIn(reduceFun: (Int, Int)=>Int):Int = throw new Exception("Liste leer!")
```

- Klasse IntList:

```
def reduceIn(reduceFun: (Int, Int)=>Int): Int= tail match{
    case Empty => head
    case _ => reduceFun(head,tail.reduceIn(reduceFun))
}
```

Generisches Map

```
def mapG[T,G] (f:T=>G, list:List[T]):List[G] = list match{
  case Nil => Nil
  case x::xs => f(x)::mapG(f,list.tail)
 }
```
[1]

Wie würden Sie mit der Map-Funktion eine Liste von Tupeln in eine Liste von Zahlen umwandeln, in dem Sie die Werte innerhalb des Tupels addieren? Also: List((1,2),(3,4),(4,5)) → List(3,7,9)

```
mapG[(Int,Int),(Int)]((X) => X._1+ X._2,List((1,2),(3,4),(4,5)));
```

Filter-Funktion

```
def filter[T](f:T=>Boolean, list:List[T]):List[T]= list match{
  case Nil => Nil
  case x::xs => if (f(x)) x::filter(f,list.tail)
  else filter(f,list.tail)
}
```

Wie würde Sie alle die Elemente aus der List von Tupeln herausfiltern, deren Summe eine ungerade Zahl ergibt?

```
filter[(Int,Int)] ((X)=>((X._1+X._2) % 2==0), List((1,2),(3,5),
    (4,5)))
```

flatten

Implementieren Sie eine Funktion, die eine beliebige Liste flach zieht, d.h. innere Verschachtelungen mit List-Elementen auflöst.

Beispiel: flatten(List(List(1,List(2),4, List(), List(5))))

→ List(1,2,4,5)

```
def flatten(l: List[Any]): List[Any] = l match {
    case Nil => Nil
    case (head: List[_]) :: tail => flatten(head) ++ flatten(tail)
    case head :: tail => head :: flatten(tail)
 }
```

flatMap in Scala

Die Funktion flatten fasst eine Liste von gleichartigen Elementen vom Typ List[B] in eine Liste vom Typ[B] zusammen

Beispiele:

val liste1= List(List(1,2),List(3),List(4,5,6))

liste1.flatten → List(1,2,3,4,5,6)

val liste2= List(List(1,2),List(List(3)),List(4,5,6))

liste2.flatten → List(1,2,List(3),4,5,6)

val liste3= List(List(1,2),3,List(4,5,6))

liste3.flatten → Fehler, da 3 in keiner Liste

val liste4= List(List(1,List("hello")),List(4,5,6))

liste4.flatten → List(List(1, List(hello)), List(4, 5, 6))

flatMap

Schreiben Sie eine Funktion flatMap, die ein Map ausführt und die Ergebnisse in einer Liste zusammenzieht.

Beispiele:

val l= List(1,2,3)

l. flatMap(X=>List(X)++ (1 to X)) → List(1, 1, 2, 1, 2, 3, 1, 2, 3) aber

l.flatMap(X=>List(X,(1 to X).toList)) → List(1, List(1), 2, List(1, 2), 3, List(1, 2, 3))

```
def flatMap[A, B](list: List[A])(f: A => List[B]): List[B] = list match {
  case (x::xs) => f(x) ++ flatMap(xs)(f)
  case _ => Nil
}
```

Wozu werden die Funktionen map, flatMap und filter benötigt? Warum sind die Funktionen so wichtig?

- Map, flatMap und filter bilden sogenannte Monaden (Kategorientheorie) und sind Grundlage für jede Schleife in Scala.
- Jede Schleife in Scala lässt sich durch die Aneinanderkettung der Funktionen map, flatMap und filter ausdrücken.

Syntax einer for-Schleife (List Comprehensions)

For-Schleifen haben die folgende Form:

for (s) yield e

Wobei s eine Sequenz von **Generatoren** und **Filtern** ist, während e der Ausdruck ist, der bei den Iterationen zurückgegeben wird

- Ein Generator hat die Form p <- e wobei p ein Pattern ist und e ein Ausdruck ist, der eine Collection zurück gibt
- Ein Filter ist eine boolescher Ausdruck – wenn er false ergibt, wird die Iteration abgebrochen
- Jede Sequenz startet mit einem Generator
- Anstatt (s) kann auch {s} benutzt werden - erlaubt mehrere Zeilen für eine for-Schleife
- For-Schleife kann auch ohne yield-Anweisung ausgeführt werden

Beispiele

Definieren Sie eine Funktion mittels der for-Schleife, die aus einer beliebigen Liste von Zahlen, eine Liste mit allen Quadraten bildet – entspricht also:

list map (X=>X*X)

for (X <- list) yield X*X

Filtern Sie alle Zahlen heraus deren Quadrate ungerade sind.

list map (X=>X*X) filter (X=>X%2 ==0)

for {X <- list if ((X*X)%2==0)} yield X*X

Filtern Sie alle Zahlen heraus deren Quadrate ungerade sind.

list map (X=>X*X) filter (X=>X%2 ==0)

Übung

Generieren Sie mittels einer for-Schleife alle möglichen Tupel, die sich aus den Zahlen 1-3 (1. Stelle) und den Buchstaben a,b,c (2. Stelle) bilden lassen.

Imperativ:

```
var result:List[(Int,Char)] = Nil
for (i <- 1 to 3){
   for (j <- List('a','b','c')){
      result = (i,j)::result
   }
}
```

Wie würde das Problem funktional gelöst werden?

```
for (i <- 1 to 3; j <- List('a','b','c')) yield (i,j)
```

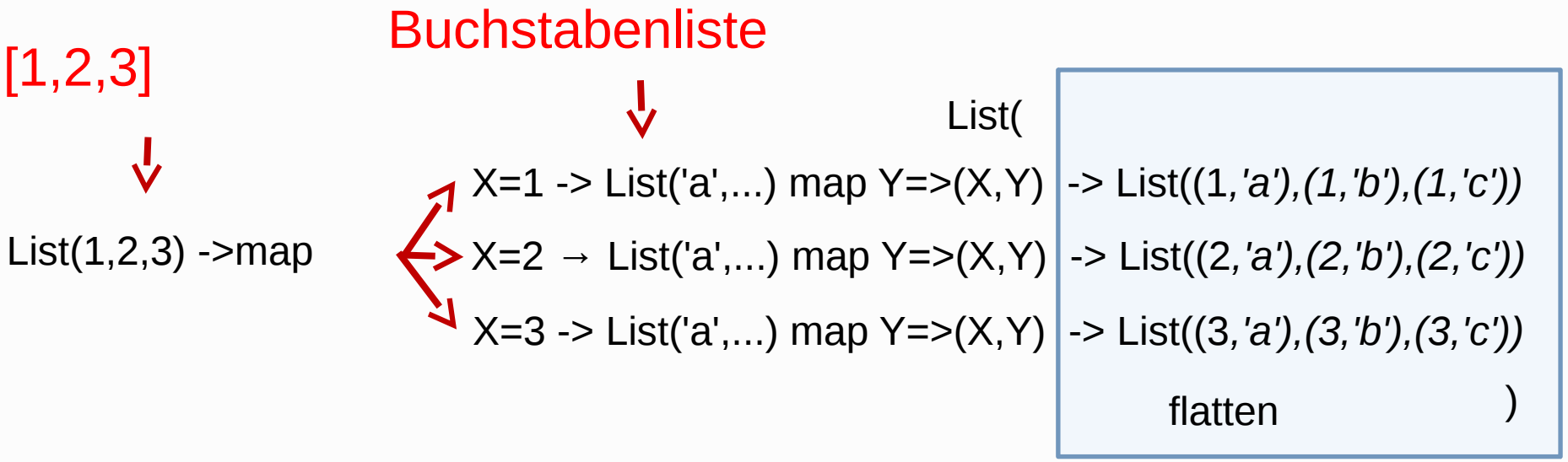
Und wie kann das mit map, flatMap und filter ausgedrückt werden?

Zusammenziehen von Generatoren

val l= List(1,2,3)

Funktion ohne flatten:

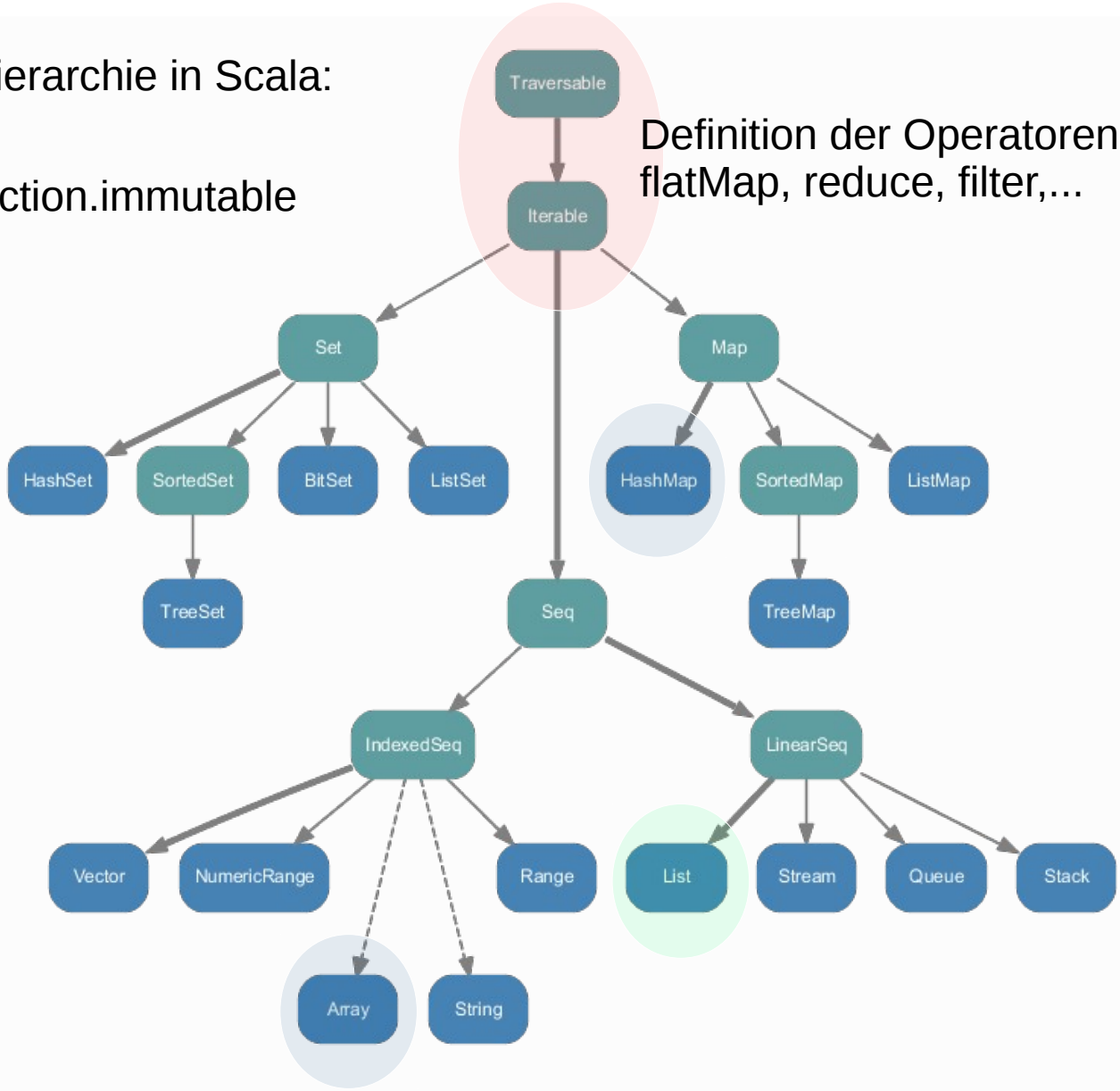
l.map(X=> List('a','b','c') map (Y=> (X,Y)))



Für jeden zusätzlichen Generator muss das einmal das Ergebnis aus dem vorigen Durchlauf flach gezogen werden!

Collectionhierarchie in Scala:

scala.collection.immutable

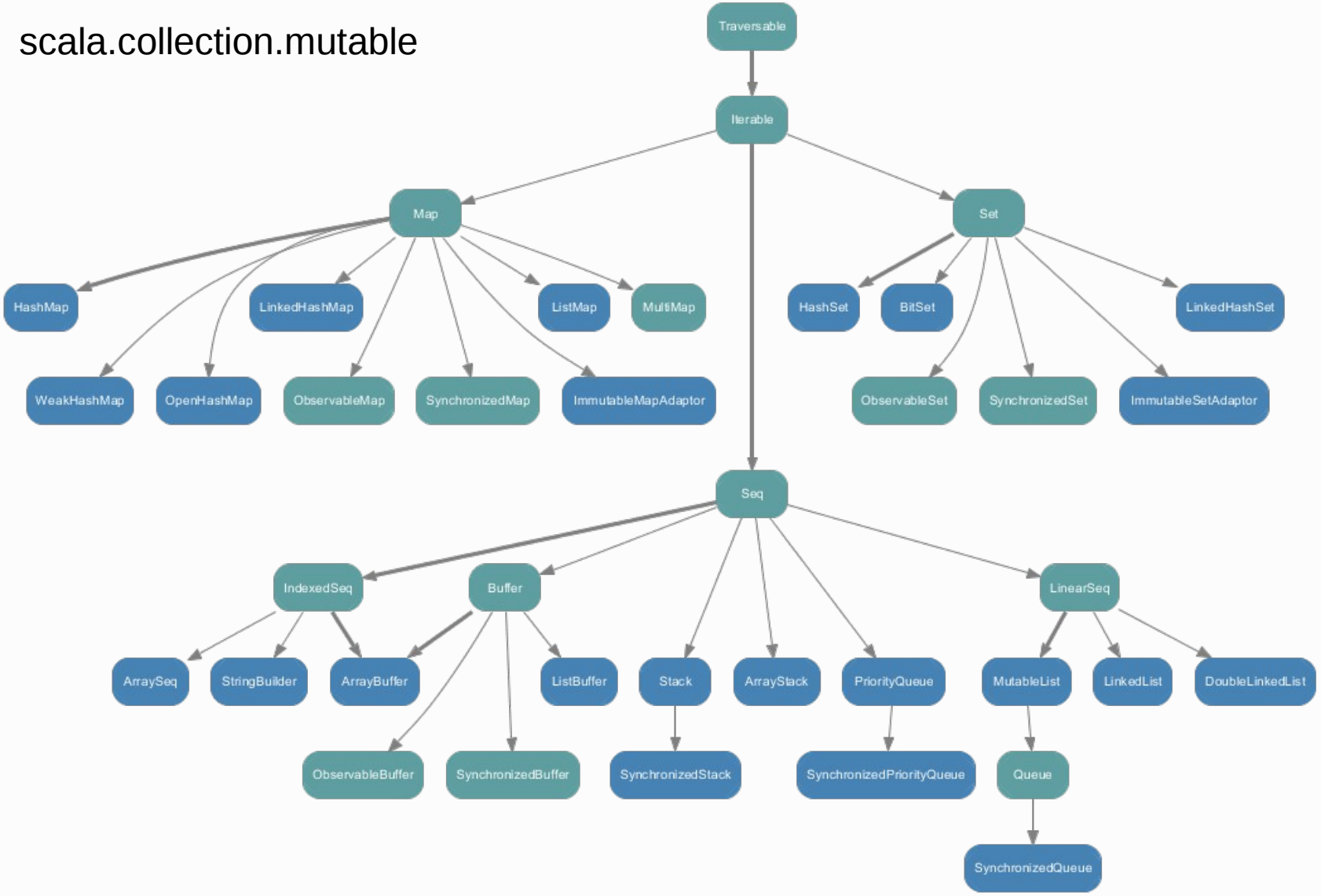


Definition der Operatoren map, flatMap, reduce, filter,...

Arrays sind nur logisch Teil der Hierarchie

Collectionhierarchie in Scala:

scala.collection.mutable



Unterschied val/var – immutable/mutable

- val/var:
  - Aussage über die Veränderbarkeit der Objektreferenz
  - Hat keine Auswirkungen auf die Veränderbarkeit der Werte
- Mutable/Immutable
  - Collections haben Operationen zur Veränderung: z.B. add, delete, update, etc.

Beispiel:

```
val l= scala.collection.mutable.MutableList(1,2,3)

l += 7   // erlaubt: anfuegen der 7 an die Liste l
```

Verboten:

```
l = scala.collection.mutable.MutableList(4,5,6)
```

Da l ein Wert ist und keine Variable.

Class Array

- Sind ähnlich den Java Arrays
- Indizierte Menge (Index beginnt mit 0)

val a= new Array(5)

→ Array(null, null, null, null, null)

val b= new Array[Int](5)

→ Array(0,0,0,0,0)

Val c= Array(1,2,3,4,5)

→ Array(1,2,3,4,5)

c(0)

→ 1

c(5)=10

→ Array(1,2,3,4,10)

Anwendung Higher Order Functions

Gegeben seien zwei n-elementige Vectoren, die durch einen Array repräsentiert werden, z.B.
v1= Array(1,2,3); v2= Array(4,5,6)

Schreiben Sie eine Funktion, die beide Vectoren miteinander addiert. (Tipp verwenden Sie die Funktion zip, die zwei Listen in eine Liste mit Tupeln mit den zusammengehörigen Elementen umwandelt:

zip[B](that: GenIterable[B]): Array[(A, B)])

```
def add_vec(v1: Array[Double], v2:Array[Double]):Array[Double]=

    v1 zip v2 map (X => X._1 + X._2)
```

Indexierte Mengen: Maps

Beispiel:

val m=Map('a'->1, 'b'->2,'c'->3)

→ m : scala.collection.immutable.Map[Char,Int] = Map(a -> 1, b -> 2, c -> 3)

Möglichkeiten des Auslesens der Elemente:

- Auslesen über Index:

  m('a')

  → 1

  m('q')

  → java.util.NoSuchElementException: key not found: q

  Exception Handling:

  try{ m('q')} catch{ case e:java.util.NoSuchElementException => 0 }

Map Auslesen

val m=Map('a'->1, 'b'->2,'c'->3)

- m.getOrElse('a',0)

  GetOrElse liefert den Wert zum Schlüssel von 'a'. Ist dieser Schlüssel nicht vorhanden, so wird der Defaultwert 0 zurückgegeben. Es wird keine Exception geworfen.

- m.get('a')

  getliefert ein Objekt vom Typ Option zurück. Option hat zwei Subtypen: None und Some. Wird der key nicht gefunden, so ist das Ergebnis None; ist er vorhanden, so ist er Some. Ein Objekt vom Typ Some enthält den Wert des Schlüssels

  ```
  m.get('q') match {
      case None => "Nichts"
      case Some(x) => x.toString }
  ```

Datentyp Option

- Scala beinhaltet keinen Wert null (nur für die Kompatibilität zu Java
- Option hat zwei Subklassen: None (kein Wert) oder Some[Typ]
- Methoden von Option:
  - get: gibt Wert der Option (oder NoSuchElementException)
  - isEmpty: ist Option leer?
- Vorteile des Datentyps Option:
  - Some und None können mittels Pattern Matching unterschieden werden
  - Mengenoperationen können ausgeführt werden, auch wenn manche Operationen kein Ergebnis liefern (z.B. Berechne für alle Zahlen einer Liste den reziproken Wert – was passiert, wenn die 0 enthalten ist? dazu später…)

Aggregationsfunktionen für Mengen

- Reduce:
  - Ohne Startwert, assoziative binäre Aggregationsoperation
- ReduceLeft, ReduceRight:
  - Ohne Startwert, beliebige Aggregationsoperation
- Fold
  - Mit Startwert, assoziative Aggregationsoperation
- FoldLeft, FoldRight
  - Mit Startwert, beliebige Aggregationsoperation

Assoziativität: a¤(b¤c)=(a¤b)¤c

¤ - binärer Operator

Beispiel Reduce-Funktionen

Ziel: Aufaddieren der Werte eines Arrays: Verwendung der Funktion reduce.

```
val l= Array(1,2,3,4)

l.reduce ((X,Y)=> X+Y)

l.reduceLeft ((X,Y)=> X+Y)

l.foldLeft (0)((X,Y)=> X+Y)
```

Warum ist der Funktionsaufruf l.reduce((X,Y)=>X-Y) nicht sinnvoll?

Zum Aufsummieren bieten Collections eine sum-Funktion.

Beispiel reduceLeft

Mögliche Implementierung:

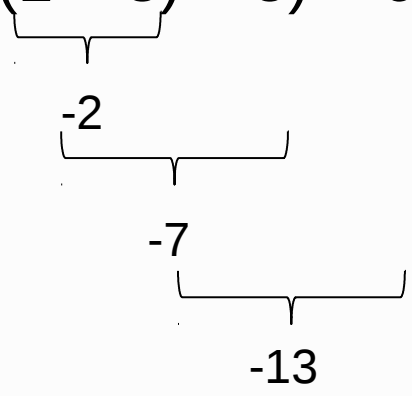
```
def reduceLeft[T](reduceFun:(T,T)=>T, li:List[T]):T= li match {
   case Nil => throw new Exception("List empty")
   case x::Nil => x
   case x::xs => reduceFun(x,reduceLeft(reduceFun,xs))
 }
  reduce[Int](_+_,l)
```

ReduceLeft reduziert die Menge von links nach rechts – Achtung: Die Funktion reduce stellt beim reduzieren keine Reihenfolge sicher!

Beispiel reduceLeft

Auswertung von: reduceLeft[Int](_-_ ,List(1,3,5,6))

(1 – 3) – 5) – 6)) = -15



Syntaktischer Zucker: Wenn Variablenanordnung der Funktion klar, können die Variablen durch Wildcards ersetzt werden
Hier für:
(x,y)=>x-y

Reduzieren mit beliebigen Reihenfolgen ergeben andere Ergebnisse.

Anwendung for-Schleife

Gegeben sei die folgende Liste:

```
val db =List(("francesco", "bloodsports"), ("simon", "jamesBond"), ("marcus",
    "jamesBond"), ("francesco", "die12KammernDerShaolin"))
```

Entwerfen Sie zwei Funktionen, die jeweils ermitteln, welche Filme „francesco“ gesehen hat. Verwenden Sie dabei einmal die for-Schleife und einmal die Operationen map, flatMap und filter.

```
for (x <- db if (x._1=="francesco")) yield x._2
```

```
db filter (X=>X._1=="francesco") map (_._2)
```

Anwendung for-Schleife

Gegeben sei die folgende Liste:

```
val db =List(("francesco", "bloodsports"), ("simon", "jamesBond"), ("marcus",
    "jamesBond"), ("francesco", "die12KammernDerShaolin"))
```

Entwerfen Sie zwei Funktionen, die jeweils ermitteln, wer den Film „james Bond“ gesehen hat. Verwenden Sie dabei einmal die for-Schleife und einmal die Operationen map, flatMap und filter.

```
for (x <- db if (x._2=="jamesBond")) yield x._1
```

```
db filter( X=> X._2=="jamesBond") map (_._1)
```

Anwendung for-Schleife

Gegeben sei die folgende Liste:

```
val db =List(("francesco", "bloodsports"), ("simon", "jamesBond"), ("marcus",
    "jamesBond"), ("francesco", "die12KammernDerShaolin"))
```

Entwerfen Sie zwei Funktionen, die jeweils ermitteln, wer mehr als zwei Filme gesehen hat. Verwenden Sie dabei einmal die for-Schleife und einmal die Operationen map, flatMap und filter.

```
for (x <- db; y <- db if (x._1==y._1 && x._2!=y._2)) yield x._1
```

```
db flatMap( x=> db map (y=> (x,y))) filter (z=>z._1._1== z._2._1 && z._1._2!
    =z._2._2) map (_._1._1)
```

Anwendung for-Schleife

Gegeben sei die folgende Liste:

```
val db =List(("francesco", "bloodsports"), ("simon", "jamesBond"), ("marcus",
    "jamesBond"), ("francesco", "die12KammernDerShaolin"))
```

Entwerfen Sie zwei Funktionen, die jeweils ermitteln, wer genau welche Filme gesehen hat. Ergebnis soll dabei eine Liste von Tupeln sein, die als erstes Element den Namen und dann die Liste aller Filme enthält. (Einträge können doppelt vorkommen.) Verwenden Sie dabei einmal die for-Schleife und einmal die Operationen map, flatMap und filter.

```
for (x <- db) yield (x._1, for (y <-db if (y._1==x._1 )) yield y._2)
```

```
db map (x=> (x._1, db filter (y=> x._1 == y._1) map (_._2)))
```

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner