Reaktive Programmierung

Vorlesung 4 vom 20.04.17: The Scala Collection Library

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Heute: Scala Collections

- Sind nicht in die Sprache eingebaut!
- Trotzdem komfortabel

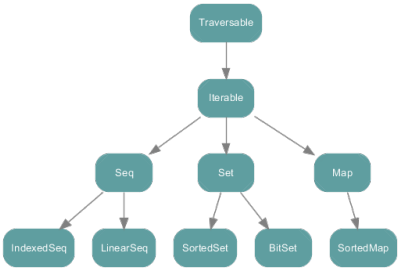
```
val ages = Map("Homer" → 36, "Marge" → 34)
ages("Homer") // 36
```

- Sehr vielseitig (Immutable, Mutable, Linear, Random Access, Read Once, Lazy, Strict, Sorted, Unsorted, Bounded...)
- Und sehr generisch

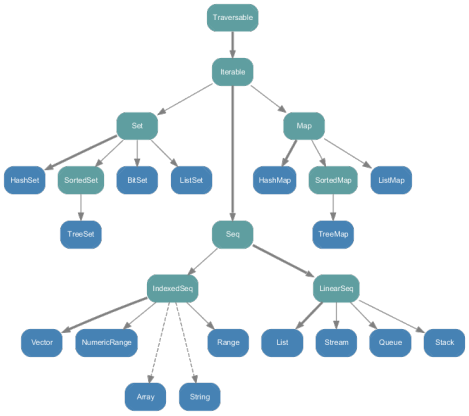
```
val a = Array(1,2,3) ++ List(1,2,3)
a.flatMap(i ⇒ Seq(i, i+1,i+2))
```


Scala Collections Bücherei

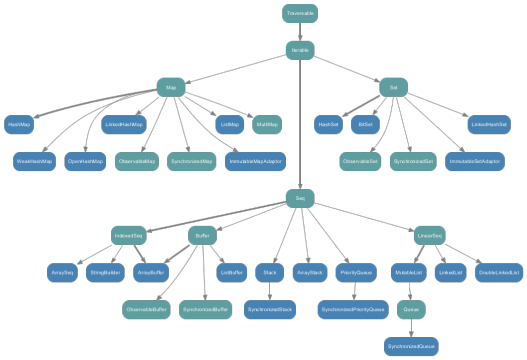
Sehr einheitliche Schnittstellen aber komplexe Bücherei:



Scala Collections Bücherei - Immutable



Scala Collections Bücherei - Mutable



Konstruktoren und Extraktoren

- Einheitliche Konstruktoren:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" → 24, "y" → 25, "z" → 26)
Set(Color.red, Color.green, Color.blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
...
```

- Einheitliche Extraktoren:

```
val Seq(a,b,c) = Seq(1,2,3)
// a = 1; b = 2; c = 3

...
```


Exkurs: Funktionen in Scala

- Scala ist rein Objektorientiert.
  - jeder Wert ist ein Objekt
  - jede Operation ist ein Methodenaufruf
- Also ist eine Funktion ein Objekt
- und ein Funktionsaufruf ein Methodenaufruf.

```
trait Function1[−T1,+R] {
  def apply(v1: T1): R
}
```

- Syntaktischer Zucker: `f(5)` wird zu `f.apply(5)`

Exkurs: Konstruktoren in Scala

- Der syntaktische Zucker für Funktionen erlaubt uns Konstruktoren ohne **new** zu definieren:

```
trait Person {
  def age: Int
  def name: String
}

object Person {
  def apply(a: Int, n: String) = new Person {
    def age = a
    def name = n
  }
}

val homer = Person(36,"Homer")
```

- Vgl. Case Classes

Exkurs: Extraktoren in Scala

- Das Gegenstück zu apply ist unapply.
  - apply (Konstruktor): Argumente ⟶ Objekt
  - unapply (Extraktor): Objekt ⟶ Argumente
- Wichtig für Pattern Matching (Vgl. Case Classes)

```
object Person {
  def apply(a: Int, n: String) = <...>
  def unapply(p: Person): Option[(Int,String)] =
    Some((p.age,p.name))
}

homer match {
  case Person(age, name) if age < 18 ⇒ s"hello young
      $name"
  case Person(_, name) ⇒ s"hello old $name"
}

val Person(a,n) = homer
```


scala.collection.Traversable[+A]

- Super-trait von allen anderen Collections.
- Einzige abstrakte Methode:

```
def foreach[U](f: Elem ⇒ U): Unit
```

- Viele wichtige Funktionen sind hier schon definiert:
  - `++[B](that: Traversable[B]): Traversable[B]`
  - `map[B](f: A => B): Traversable[B]`
  - `filter(f: A => Boolean): Traversable[A]`
  - `foldLeft[B](z: B)(f: (B,A) => B): B`
  - `flatMap[B](f: A => Traversable[B]): Traversable[B]`
  - `take, drop, exists, head, tail, foreach, size, sum, groupBy, takeWhile ...`
- Problem: So funktionieren die Signaturen nicht!
- Die folgende Folie ist für Zuschauer unter 16 Jahren nicht geeignet...

Die wahre Signatur von map

```
def map[B,That](f: A ⇒ B)(implicit bf:
  CanBuildFrom[Traversable[A], B, That]): That
```

Was machen wir damit?

- Schnell wieder vergessen
- Aber im Hinterkopf behalten: Die Signaturen in der Dokumentation sind "geschönt"!

Seq[+A], IndexedSeq[+A], LinearSeq[+A]

- Haben eine länge (`length`)
- Elemente haben feste Positionen (`indexOf, indexOfSlice, ...`)
- Können Sortiert werden (`sorted, sortWith, sortBy, ...`)
- Können Umgedreht werden (`reverse, reverseMap, ...`)
- Können mit anderen Sequenzen verglichen werden (`startsWith, ...`)
- Nützliche Subtypen: `List`, `Stream`, `Vector`, `Stack`, `Queue`, `mutable.Buffer`
- Welche ist die richtige für mich? `http://docs.scala-lang.org/overviews/collections/performance-characteristics.html`

Set[+A]

- Enthalten keine doppelten Elemente
- Unterstützen Vereinigungen, Differenzen, Schnittmengen:

```
Set("apple","strawberry") ++ Set("apple","peach")
> Set("apple", "strawberry", "peach")

Set("apple","strawberry") -- Set("apple","peach")
> Set("strawberry")

Set("apple", "strawberry") & Set("apple", "peach")
> Set("apple")
```

- Nützliche Subtypen: `SortedSet`, `BitSet`

Map[K,V]

- Ist eine Menge von Schlüssel-Wert-Paaren: `Map[K,V] <: Iterable[(K,V)]`
- Ist eine partielle Funktion von Schlüssel zu Wert: `Map[K,V] <: PartialFunction[K,V]`
- Werte können "nachgeschlagen" werden:

```
val ages = Map("Homer" → 39, "Marge" → 34)

ages("Homer")
> 39

ages isDefinedAt "Bart" // ages contains "Bart"
> false

ages get "Marge"
> Some(34)
```

- Nützliche Subtypen: `mutable.Map`

Array

- Array sind "special":
  - Korrespondieren zu Javas Arrays
  - Können aber auch generisch sein Array[T]
  - Und sind kompatibel zu Sequenzen
- Problem mit Generizität:

```
def evenElems[T](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i ← 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
```

- *Type erasure* zur Laufzeit — daher: Class manifest benötigt

```
def evenElems[T](xs: Vector[T])(implicit m:
    ClassManifest[T]): Array[T] = ...
def evenElems[T: ClassManifest](xs: Vector[T]): Array[T] =
    ...
```

- Generische Arrays erzeugen overhead: *"You can expect accesses to generic arrays to be three to four times slower than accesses to*

String

- Scala-Strings sind `java.lang.String`
- Unterstützen aber alle Sequenz-Operationen
- Beste aller Welter: effiziente Repräsentation, viele Operationen
  - Vergleiche Haskell: **type** String= [Char] bzw. ByteString
- Wird erreicht durch implizite Konversionen String to WrappedString und String to StringOps

Vergleiche von Collections

- Collections sind in Mengen, Maps und Sequenzen aufgeteilt.
- Collections aus verschiedenen Kategorien sind niemals gleich:

```
Set(1,2,3) == List(1,2,3) // false
```

- Mengen und Maps sind gleich wenn sie die selben Elemente enthalten:

```
TreeSet(3,2,1) == HashSet(2,1,3) // true
```

- Sequenzen sind gleich wenn sie die selben Elemente in der selben Reihenfolge enthalten:

```
List(1,2,3) == Stream(1,2,3) // true
```


Scala Collections by Example - Part I

- Problem: Namen der erwachsenen Personen in einer Liste

```
case class Person(name: String, age: Int)
val persons = List(Person("Homer",39), Person("Marge",34),
                   Person("Bart",10), Person("Lisa",8),
                   Person("Maggie",1), Person("Abe",80))
```

- Lösung:

```
val adults = persons.filter(_.age ≥ 18).map(_.name)

> List("Homer", "Marge", "Abe")
```


Scala Collections by Example - Part II

- Problem: Fibonacci Zahlen so elegant wie in Haskell?

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- Lösung:

```
val fibs: Stream[BigInt] =
  BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(
    n ⇒ n._1 + n._2)

fibs.take(10).foreach(println)
> 0
> 1
> ...
> 21
> 34
```


`Option[+A]`

- Haben maximal 1 Element

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some(get: A) extends Option[A]
```

- Entsprechen `Maybe` in Haskell
- Sollten dort benutzt werden wo in Java `null` im Spiel ist

```
def get(elem: String) = elem match {
  case "a" ⇒ Some(1)
  case "b" ⇒ Some(2)
  case _ ⇒ None
}
```

- Hilfreich dabei:

```
Option("Hallo") // Some("Hallo")
Option(null) // None
```


`Option[+A]`

- An vielen Stellen in der Standardbücherei gibt es die Auswahl:

```
val ages = Map("Homer" → 39, "Marge" → 34)

ages("Bart") // NoSuchElementException
ages.get("Bart") // None
```

- Nützliche Operationen auf `Option`

```
val x: Option[Int] = ???

x getOrElse 0

x foldLeft ("Test")(_.toString)
x exists (_ == 4)
...
```


Ranges

- Repräsentieren Zahlensequenzen

```
class Range(start: Int, end: Int, step: Int)
class Inclusive(start: Int, end: Int, step: Int) extends
    Range(start,end + 1,step)
```

- `Int` ist "gepimpt" (`RichInt`):

```
1 to 10 // new Inclusive(1,10,1)
1 to (10,5) // new Inclusive(1,10,5)
1 until 10 // new Range(1,10)
```

- Werte sind berechnet und nicht gespeichert
- Keine "echten" Collections
- Dienen zum effizienten Durchlaufen von Zahlensequenzen:

```
(1 to 10).foreach(println)
```


For Comprehensions

- In Scala ist `for` nur syntaktischer Zucker

```
   for (i ← 1 to 10) println(i)
⇒ (1 to 10).foreach(i ⇒ println(i))

   for (i ← 1 to 10) yield i * 2
⇒ (1 to 10).map(i ⇒ i * 2)

   for (i ← 1 to 10 if i > 5) yield i * 2
⇒ (1 to 10).filter(i ⇒ i > 5).map(i ⇒ i * 2)

   for (x ← 1 to 10, y ← 1 to 10) yield (x,y)
⇒ (1 to 10).flatMap(x ⇒ (1 to 10).map(y ⇒ (x,y)))
```

- Funktioniert mit allen Typen die die nötige Untermenge der Funktionen (`foreach`,`map`,`flatMap`,`withFilter`) implementieren.

Scala Collections by Example - Part III

- Problem: Wörter in allen Zeilen in allen Dateien in einem Verzeichnis durchsuchen.

```
def files(path: String): List[File]
def lines(file: File): List[String]
def words(line: String): List[String]

def find(path: String, p: String ⇒ Boolean) = ???
```

- Lösung:

```
def find(path: String, p: String ⇒ Boolean) = for {
  file ← files(path)
  line ← lines(file)
  word ← words(line) if p(word)
} yield word
```

Zusammenfassung

- Scala Collections sind ziemlich komplex
- Dafür sind die Operationen sehr generisch
- Es gibt keine in die Sprache eingebauten Collections:
  Die Collections in der Standardbücherei könnte man alle selbst implementieren
- Für fast jeden Anwendungsfall gibt es schon einen passenden Collection Typ
- `for`-Comprehensions sind in Scala nur syntaktischer Zucker