

Inhaltsverzeichnis

1	Einführung	3
1.1	Einfache Programme	3
2	Eigenschaften zur Sprache	5
2.1	Paradigmen	10
2.2	Operationen sind Objekte	11
2.3	Varablendeklarationen	12
2.4	if/else und while	12
2.5	for-Schleife	13
2.6	try-catch, throw und finally	13
2.7	Enumerations	13
3	Ausdrücke und Funktionen	15
3.1	Funktionen	15
3.1.1	Verschachtelte Funktionen und Parameter	15
3.1.2	Parameter	16
3.1.3	Callback	16
3.1.4	Anonyme Funktionen	17
3.2	Schwanzrekursion	17
3.3	Currying	18
4	Klassen und Objekte	19
4.1	Klassen	21
4.2	Klassenimport	21
4.3	Klassenhierarchie	21
4.4	Overriding	22
4.5	Abstraktion	22
4.6	Companion Objekte	23
4.7	Komposition	24
4.8	Dekomposition	25
4.9	Case-Klassen	26
4.10	Pattern Matching	27
4.11	traits	29
5	Generische Typen und Methoden	30
5.1	Annotationen Varianz	31
5.2	Tuples	31
5.3	Wann man explizite Typannotationen braucht	32
6	Listen und der Spaß mit der Unveränderlichkeit	33
6.1	Scala Listen, Tupel and Map-Klassen	33
6.2	List[T]	34
6.3	Transformation	35
6.4	Tupel	37
6.5	Map[K,V]	38
6.6	Option[T]	39

7 Spaß mit Funktionen	39
7.1 Partielle Anwendungen und Funktionen	39
7.2 Funktionen und Typparameter	40
7.3 Funktionen in Container packen	41
8 Pattern-Matching	41
9 Varianz	43
9.1 Invariante Parametertypen	43
9.2 Kovariante Parametertypen	43
9.3 Kovariante Methoden	43
9.4 Varianzregeln	44
10 Scalas Objekt System	45
10.1 Typhierarchie	45
10.2 Linearisierung der Objekthierarchie	45
11 FP in Scala	48
11.1 Was FP ist	48
11.2 FP in Scala	48
11.3 Rekursion	48
11.4 Tail Calls und Tail-Call Optimierung	49
11.5 Funktionale Datenstrukturen	49
11.6 Traversieren, Mapping, Filtering, Folding und Reducing	49
11.7 Pattern Matching	50
11.8 Partielle Funktionen	50
11.9 Currying	51
11.10 Implicits	51
11.11 Lazy Vals	52
12 Scala Typsystem	53
12.1 Parametrisierte Typen	53
12.2 Typen Bounds	53
12.3 Nothing und Null	53
13 Fragen	53
13.1 Temporäres für DA	54
13.1.1 Statische versus Dynamisch typisierte Sprachen	54
13.1.2 Typinferenz	54
14 Sachen für MT	54

1 Einführung

- **Kompilierung:** `scalac`¹ bzw. `fsc`²
- **Ausführung:**³ `scala`
- **Interpreter:** nur `scala` in Konsole eingeben

1.1 Einfache Programme

- Hello Helex:

```
1 object HelloWorld {  
  
    def main(args: Array[String]) = {  
        println("Hello Helex!");  
5    }  
}  
  
// kuerzer durch Verwendung des Mixins  
9 object HelloWorld extends Application {  
    println("Hello Helex!");  
}
```

⇒ Datei muss wie das Objekt heißen, damit es ausgeführt werden kann

- Zahlen:

```
for {i <- 1 to 10  
    j <- 1 to 10}  
    println(i * j)
```

- ein Programm, dass ein String in Int parsed und dabei alle Zahlen der Eingabe aufsummiert:

```
1 import scala.io._  
  
def toInt(in: String): Option[Int] =  
    try {  
5        Some(Integer.parseInt(in.trim))  
    } catch {  
        case e: NumberFormatException => None  
    }  
  
9 def sum(in: Seq[String]) = {  
    val ints = in.flatMap(s => toInt(s))
```

¹Resultat sind JVM Klassen-Dateien, welche man in JARs packen kann, hierbei wird jedoch class, trait od. object-Definition verlangt

²schnellere Kompilierung

³Programm wird kompiliert u. danach gleich ausgeführt

```

    ints.foldLeft(0)((a,b) => a + b)
13 }

println("Enter some numbers and press ctrl-D")

17 val input = Source.fromInputStream(System.in)

    val lines = input.getLines().collect

21 println("Sum "+sum(lines))

```

– *Option*:

- * *Option* ist Container, der *Null* (dann ist *None*) od. ein Element dann ist *Some(theElement)* enthält
- * durch *Option* verhindert man *null Pointer Exceptions*
 ⇒ gut wenn man Business-Logik schreibt und diesen Fall nicht in jeder Abfrage, sondern einfach am Ergebnistyp der Funktion festlegt⁴
- * Parsing des Strings: sollte eben keine Zahl eingegeben werden, so wird keine Exception geworfen, sondern die Ausgabe einfach auf *None* gemappt

– *sum*

- * in der Methode *sum* definieren wir keinen return-Wert
- * *in* Parameter ist vom Typ *Seq*⁵, was ein *trait*⁶ ist
- * *traits*⁷ können implementierte Methoden beinhalten u. sind am Besten mit mixins aus Ruby zu vergleichen
- * mit *flatMap* ruft die Methode *toInt* für jedes Element der Sequenz in *in* auf
- * mit *s => toInt(s)* definieren wir eine anonyme Funktion, die einen einzelnen Parameter *s* nimmt u. diesen an die Funktion *toInt* weitergibt
- * *foldLeft*⁸ nimmt einen einzelnen Parameter als *seed* u. und schreibt das Ergebnis der inneren Funktion an diesen *seed* zurück. Dabei wird der *seed* solange weiter ↑, bis alle Elemente der Sequenz durchlaufen wurden

⁴denke an Adminbill aus *pictrs*

⁵*Seq* ist *supertrait* von *Array*, *List* u. andere *Collections*

⁶denke Interface aus Java

⁷*traits* beheben das Diamanten-Prob. der multiplen Vererbung, da eine Klasse beliebig viele *traits* haben

⁸kann man gut verwenden, wenn man die Werte einer Sequenz aufsummieren will

2 Eigenschaften zur Sprache

- Wofü Scala geeignet ist: *language ideal for today's scalable, distributed, component-based applications that support concurrency and distribution*
- **Merke:** *Is a statically typed⁹, mixed-paradigm, JVM language with a succinct, elegant, and flexible syntax, a sophisticated type system, and idioms that promote scalability from small, interpreted scripts to large, sophisticated applications.*
- Programmiersprachen für Softwarekomponenten müssen **skalierbar** sein ⇒ Konzentration bei Scala auf Abstraktion, Komposition und Dekomposition
- skalierbare Unterstützung für Komponenten kann nur erreicht werden, wenn OOP generalisiert und mit funktionalen Aspekten (FP) einer Programmiersprache vereinigt werden
- Scala arbeitet gut mit Java und C# zusammen
- Typsystem von Scala hat folgende Vorteile:
 1. Abstrakte Typdefinitionen und vom Pfad abhängige Typen¹⁰ unterstützen
 2. modulare mixing Komposition
 3. *views*¹¹
- Scala Klassen u. Objekte können von Java-Sachen erben u. Java- Interfaces implementieren ⇒ man kann Scala-Code in einem Java-Framework¹² verwenden
- **high-order functions:** sind Funktionen, die Funktionen als Argumente nehmen od. Funktionen als Ergebnis zurückliefern u. diese werden von Scala unterstützt
- **scope** verschachtelte Funktionen können auf alle Parameter u. lokalen Variablen innerhalb ihrer Umgebung zugreifen
- Funktionen-Def mit nur einer Zeile benötigen keine geschweiften Klammern
- **id: type**-Syntax wird von Scala verwendet
- **unit** wird statt void in Scala verwendet
- Scala hat eine Darstellung von Javas void, nämlich Unit¹³
- alle Kontrollstrukturen von Java sind auch in Scala¹⁴ vorhanden
- in Scala hat alles einen öffentlichen Zugriff, es sein denn es wird anders definiert
- **Klassen mit Argumenten:** Argumente dienen als Konstruktoren für die Klasse

⁹der Typ einer Variable ist für die gesamte Lebenszeit der Variable fest

¹⁰"vObj calculus"

¹¹ermöglichen Komponenten-Adaption in einem modularen Weg

¹²wicket

¹³kann man explizit zurückgeben, wenn man einfach () beim Returnwert einer Funktion hinschreibt

¹⁴for-Schleifen wurde stark vereinfacht :)

```

class Succ(n: Nat){
  def isZero: boolean = false
3  def pred: Nat = n
  override def toString: String = "Succ("+n+")"

```

- Scala-Objekt ist Instanz einer Klasse u. kann deswegen als Parameter von Methoden agieren kann
- Klassen, Objekte u. *traits*¹⁵ können innere Klassen, Objekte u. traits haben, welche Zugriff auf *private* Methoden, Variablen und so weiter haben
- die Import-Methode kann innerhalb von Blöcken verwendet werden ⇒ können dadurch feingranular den scope festlegen
- Scala ist statisch typisiert
- **Nothing**: eine Methode mit diesen Rückgabewert, wird normalerweise niemals laufen
- **Any** ist die Mutter aller Klassen in Scala
- **AnyRef** bedeutet dasselbe wie Javas Object, jedoch mit den Unterschied, dass mit == die inhaltliche Gleichheit von Objekten gemeint ist - will man die Referenz von Objekten beurteilen, dann nimmt man lieber die Methode eq
- der return-Wert von Funktionen ist per default die letzte Zeile einer Methode¹⁶
- **call-by name** kann man in Scala durch => für Funktionsaufrufe anfordern:

```

def nano()= {
  println("Dwarf Warriors:")
  12
4 }

def delayed(t: => Long) = {
  println("Delayed method")
8  println("Count: " + t)
}

def notDelayed(t: Long) = {
12  println("not delayed method")
    println("Count: " + t)
}

16 delayed(nano)

/*

```

¹⁵um es in Java-Sprache auszudrücken sind *traits* Interfaces mit einer Superklasse, die nicht-abstrakte Methoden beinhalten dürfen

¹⁶man kann aber auch explizit return angeben

```

20     Delayed method
    Getting nano
    Count: 12
    */

24 notDelayed(nano)
    /*
    Getting nano
    not delayed method
28     Count: 12
    */

```

⇒ es kommen verschiedene Zeiten heraus ⇒ in *delayed* wird bereits reingegangen bevor *nano* aufgerufen wird und somit wird *nano* zweimal aufgerufen

- [B >: T] heißt, dass B mindestens von derselben Klasse wie T
- **impliziten Konversion:** man fügt einer eigentlichen als `final` deklarierten Klasse noch zusätzliche Methoden hinzu¹⁷
- Scala ist FP, d.h. in dem Sinne, dass jede Funktion einen Wert hat
- Scala ist statisch Typisiert u. das Typsystem unterstützt:
 - **generische Klassen**
 - **Varianz**-Annotationen
 - obere u. untere Schranken für Typen
 - *compound types*
 - polymorphe Methoden
 - Scala ist erweiterbar: man kann leicht neue Sprachkonstrukte zur Sprache ergänzen
- Scala kompiliert in normalen Java Bytecode
- Scala arbeitet gut mit Java u. .NET an
- jede Java Klasse kann als eine normale Scala-Klasse verwendet werden ⇒ deswegen sind alle Java-Bibos auch direkt in Scala verwendbar
- gewöhnlicher Java-Code ist kein valider Scala-Code, aber wenn der Java-Code einmal kompiliert wurde, dann kann er vom Scala-Code verwendet werden
- Scala behandelt das auftauchen von Bezeichnern zwischen zwei Ausdrücken als Methodenaufruf
- Scala läuft wie Java auf derselben JVM u. sie teilen sich deshalb den gleichen Garbage-Collector
 - auf Java Bytecode kann Scala:
 - * Objekte instanziiieren

¹⁷einfach `implicit` vor Methodendef schreiben

- * Methoden aufrufen
 - * exceptions werfen/abfangen
 - * Klassen erweitern
 - * Interfaces implementieren
- Java-Klassen als Mixins, wenn diese als Quellcode vorhanden sind
- Java “locking u. concurrency model” wird unterstützt, wird aber normalerweise von Scala gewrapped
- Imports sind wie in Java, nur mit mehr Features (denke ans Alias) u. imports können allen Stellen des Programms gemacht werden
- wenn Typen offensichtlich sind, dann muss man diese nicht angeben, kann aber zu bösen Fehlern führen
- Generics¹⁸:
 - Klassen u. traits können generisch gemacht werden
 - via *Typparameter* (Nonvariant, Covariant, Contravariant)
 - obere u. untere Schranken
- Scala erlaubt die Definition von parameterlosen Methoden u. jedesmal wird so eine Funktion aufgerufen, wenn dessen Name verwendet wird
- FP Eigenschaften: *Higher-order functions*; *Function closure support*; Rekursion als *flow control*; *pure Funktionen* ⇒ keine Seiteneffekte¹⁹; *Pattern Matching*
- immutable `val` müssen initialisiert werden!
- Klassen können überall in einem Programm auftauchen: top-level, innerhalb von anderen Klassen (*inner classes*), innerhalb von Codeblöcken (*local classes*) u. innerhalb von Ausdrücken (*anonymous classes*) ⇒ analog gilt das für Funktionen in Scala, nur das Funktionen nicht als Top-Level deklariert werden können
- Scala ist eine stark typisierte Sprache: *class types*, *variant class type parameters*, *virtual types*, *qualified class types*, *compound types*²⁰, *singleton types*²¹, *explicit self types*²²
- Scala unterstützt *Symbole* aus Ruby
- `null` gibts in Scala, aber ab in die Tonne damit u. besser `Option` verwenden
- wenn man **final** vor Klassen od. traits schreibt, dann verhindert man, dass davon Klassen abgeleitet werden können
- **super** ist analog zu `this`, aber es bindet an die Elternklasse
- **this** wie ein Objekt auf sich alleine zeigt
- das `$` verwendet Scala intern für irgendwas, also ebenso wie die keywords nicht als Variablennamen verwenden

¹⁸statt Generics sagt man in Scala zu dynamischen Datentypen von Funktionen *parameterized types*

¹⁹viele Datenstrukturen sind immutable, mit `val` sind immutable u. mit `var` sind mutable

²⁰kann man festlegen, dass ein Wert eine Instanz von einer Liste von Klassen ist

²¹für Typen gibt es genau einen Wert

²²sind Annotationen, die den Typ einer aktuellen Instanz einer Klasse festlegen

- Scala-Konvention: Klammern bei Methodenaufrufen vermeiden, wenn diese keine Seiteneffekte verursachen

- **Generatoren:**

```

val clans = List("Eshin", "Test", 2)

// one not so real generator
for(i <- clans
  if clans.contains("Eshin")
  ) println("Skaven!")

// a real generator
for(j <- clans
  if j == "Eshin"
  ) println("Skaven are here")

```

der *left-arrow* Operator wird eben Generator genannt, der er die einzelnen Elemente aus der Collection generiert

- der `=>` Operator gibt an, dass aktuelle Argumente für diesen Parameter unausgewertet übergeben werden.

die Argumente einer solchen Funktion werden jedesmal ausgewertet, wenn der formale Parameter erwähnt wird

```

def && (n: => Bool): Bool = this
def || (n: => Bool): Bool = n

```

- die Behandlung von Variablenzugriffen als Methodenaufrufen ermöglicht es in Scala **properties** zu definieren. Im folgenden Beispiel wird die Eigenschaft *degree* definiert, welche nur einen Wert entspricht, der größer od. gleich -273 ist

```

class Celsius {
  private var d: int = 0
  def degree: int = d
  def degree_=(x: int): unit = if (x >= -273) d = x
}

```

- für jede Variable **var** *x*: *T* definiert Scala die folgenden *setter* und *getter* Methoden:

```

def x: T
def x_=(newval: T): unit

```

diese Methoden referenzieren und updaten die entsprechende Speicherzelle für die Variable, welche nicht direkt durch Scala-Programme beeinflussbar ist

- Kommentare:

```

/*

```

```

2  multiline
   */

   // single line

6  /*
   this is outer comment
   */
10 inner comment
   */
   a
   */

```

- **def** wird zur Festlegung von Funktionen verwendet
- **Array Typen** werden `Array[T]` geschrieben u. **Array-Zugriffe** werden mit `a(i)` statt `a[i]` geschrieben
- Scala unterscheidet nicht zwischen Identifier u. Operatorennamen, d.h. `xs filter (pivot >)` ist äquivalent zu `xs.filter(pivot >)`
- alle Funktionen haben die `apply`-Methode, welche die Funktion ausführen

2.1 Paradigmen

① OOP-Paradigma

- alles ist ein Objekt
- Scala hat die typischen Mechanismen von OOP, aber ergänzt das ganze noch durch *traits*, *mixin composition*
- es gibt keine primitiven Datentypen wie in Java, anstelle sind alle numerischen Typen Objekte
- Scala unterstützt *singleton object construct*

② FP-Paradigma

- FP sind gut für Design-Probs wie *concurrency*, da pure FP keine Δ Zustände erlaubt²³
- in puren FP kommunizieren Programme durch den Austausch von nebenläufigen autonomen Prozessen \Rightarrow Scala unterstützt das durch seine *Actors Library*, aber es unterstützt auch veränderliche Elemente, wenn man das will
- Funktionen sind *first class*²⁴ u. Scala bietet *closures*²⁵

²³um Synchronisation muss man sich nicht kümmern

²⁴d.h. sie können an Variablen, an andere Funktionen usw. ähnlich wie Werte übergeben werden

²⁵bezeichnet man eine Programmfunktion, die beim Aufruf einen Teil ihres vorherigen Aufrufkontexts reproduziert, selbst wenn dieser Kontext außerhalb der Funktion schon nicht mehr existiert \Rightarrow sind ein mächtiges Werkzeug zur Abstraktion

③ **Skalierbarkeit**²⁶ wird durch folgende Sachen gewährleistet:

1. explizite *self types*
2. abstrakte *type members* u. *generics*
3. verschachtelte Klassen
4. *mixin* Komposition durch Verwendung von *traits*

④ **Performanz**

- da Scala ja auf der JVM läuft, unterstützt auch die ganzen dafür entwickelten Optimierungsmethoden (Profilers, verteilter Cache, Clustering)

2.2 Operationen sind Objekte

Methoden sind funktionale Werte

- betrachten die folgende Funktion, welche überprüft, ob ein Array eine Element mit einer bestimmten Eigenschaft (Prädikat) hat:

```
// exist an element in xs with property p?
def exists[T](xs: Array[T], p: T => boolean) = {
  3   var i: int = 0
      while (i < xs.length && !p(xs(i))) i = i + 1
      i < xs.length
  }
  7
def forall[T](xs: Array[T], p: T => boolean) = {
  // nested function
  def not_p(x: T) = !p(x)
  11 !exists(xs, not_p)
  }

// shorter and eleganter than forAll (works with anonymous function)
15 def forallAnonymous[T](xs: Array[T], p: T => boolean) =
    !exists(xs, (x: T) => !p(x))
```

- der Elementtyp des Arrays ist beliebig, wird durch den Parameter [T] angegeben der exists-Methode angegeben
- die zu testende Eigenschaft ist beliebig u. dies wird durch den Parameter *p* der exists-Methode repräsentiert
- der Typ von *p* ist der *Funktionstyp* *T => boolean*, welche als Werte alle Funktionen mit der Domäne *T* und den Bereich von *boolean* hat
- Funktionsparameter können wie normale Funktionen angewendet werden (siehe im *p* in while-Schleife)

²⁶es wurde designt, um von kleinen, interpretierten Skripten zu großen, verteilten Anwendungen zu skalieren

- mithilfe der obigen Funktion können wir eine Funktion `forall` via Doppel-negation erstellen: Ein Prädikat gilt für alle Elemente eines Arrays, wenn es kein Argument gibt, dass nicht die Eigenschaft des Prädikats erfüllt
 - * `forall` definiert eine **geschachtelte Funktion** *not p*, welche den Parameter *p* negiert
 - * `forallAnonymous`: hier definiert $(x : T) \Rightarrow !p(x)$ eine anonyme Funktion, die alle Parameter vom Typ *T* nach $!p(x)$
- wenn Methoden Werte sind u. Werte Objekte, dann folgt, dass Methoden selbst Werte sind

Funktionen verfeinern

- da FunktionsTypen in Scala Klassen sind, kann man Sie in Unterklassen weiter verfeinern
- Klasse `Array[T]` erbt von der Funktion `Function1[Int, T]` u. fügt Methoden für Array-Update, Array-Länge usw. hinzu

```
class Array[T] extends Function1[Int, T] with Seq[T]
  def apply(index: Int): T = ...
  def update(index: Int, elem: T): Unit = ...
4  def length: Int = ...
```

2.3 Variablendeklarationen

- werde wie Methoden definiert beginne aber mit einen der folgenden *keywords*:
 - var**: können ihren Wert Δ (genauso wie Variablen in Java)
 - val**: definiert nur Werte \Rightarrow *read only*
 - lazy val**: wird erst dann zugewiesen, wenn die Variable verwendet wird

2.4 if/else und while

- if/else wird eher selten verwendet u. verhalten sich eher wie der Ternary-Operator; while-Schleifen sind ebenso effizient wie Rekursion²⁷

```
if (exp) println("yes")

// ternary + if else
4 val i: Int = if (exp) 1 else 2

// while
while (exp) println("Working...")
8 while (exp) {
  println("Working...")
}
```

²⁷beachte hierbei die *tail-rekursion* bei funktionalen Sprachen

- das Ergebnis von if u. while ist immer Unit

2.5 for-Schleife

- einfache Variante ist wie in Java:

```
1 for {i <- 1 to 3} println(i)
```

- verschachtelte Variante:

```
for {i <- 1 to 3
    j <- 1 to 3} println(i * j)
```

- in for-Schleifen kann man auch guards packen:

```
def isOdd(in: Int) = in % 2 == 1
2 for {i <- 1 to 5 if isOdd(i)} println(i)
```

2.6 try-catch, throw und finally

- throws bzw. try/finally funzt wie in Java:

```
throw new Exception("Working...")
2
try {
    throw new Exception("Working...")
} finally {
6    println("This will always be printed")
}
}
```

- try/catch ist anders:

- es gibt immer ein einen Wert zurück
- es weist einen default Wert zu, sobald alle anderen Tests durchgefallen sind

```
1 try {
    file.write(stuff)
} catch {
    case e: java.io.IOException => // handle IO Exception
5    case n: NullPointerException => // handle null pointer
}
}
```

2.7 Enumerations

Einfach Klassen von Enumeration erben lassen u. es ist dann keine besondere Notation für die Elemente der Aufzählung nötig

```
object Breed extends Enumeration {  
  2   val moulder = Value("Master Moulders")  
    val pestilens = Value("Biological Weapon")  
    val eshin = Value("Assassins")  
    val skyre = Value("Gatling Gun and Warpstone")  
  6   val khazad = Value("Great race in Warhammer")  
    }  
  
  for (breed <- Breed) println(breed.id + "\t" + breed)  
  
  10 // print just the molders :)  
    println("\nJust Moulders:")  
    Breed.filter (_.toString.endsWith("Moulders")).foreach(println)
```

3 Ausdrücke und Funktionen

- Unterschied zwischen `def x = e` u. `val x = e`:
 - `def` - hier wird `e` nicht ausgewertet, sondern erst, wenn `x` verwendet wird
 - `val` - hier wird `e` sofort ausgewertet²⁸
- Frage: Wie werden Ausdrücke ausgewertet?
 - schnappe die die am meisten links stehende Operation
 - werte die Operanden aus
 - verwende die Operation entsprechend mit den Werten der Operanden

3.1 Funktionen

- Funktionen können in Scala Symbole wie `+`, `-`, `*`, `and`, `?` enthalten
- Funktionen können die folgende Form haben: `Function[A, B]`, wobei `A` der Parametertyp u. `B` der Rückgabewert ist
⇒ andere Schreibweise für `Function[A, B]`:

`A => B`

- Funktion, welche andere Funktionen als Parameter nehmen werden *high-order* Funktionen genannt
- falls man eine Variable Liste an Parametern haben möchte, dann muss `*` in die Parameterliste einer Funktionsdef setzen:

```
def largest(as: Int*): Int = as.reduceLeft((a,b) => a max b)
```

- man kann auch Schranken für Typen definieren. Im folgenden Beispiel müssen alle Typen vom Typ `Number` od. von einer Subklasse von `Number` sein:

```
def sum[T <: Number](as: T*): Double = as.foldLeft(0d)(_ + _.doubleValue)
```

3.1.1 Verschachtelte Funktionen und Parameter

- ---

```
def factorial(i: Int): Int = {  
  def fact(i: Int, accumulator: Int): Int = {  
3    if (i <= 1)  
      accumulator  
    else  
      fact(i - 1, i * accumulator)  
7  }  
  fact(i, 1)  
}
```

²⁸sobald man `x` verwendet, ist der Ausdruck `e` bereits vorhanden

fact kann man nur innerhalb des Scopes von factorial aufrufen, sonst kommt es zu einem Compilerfehler.

- weitergabe von Parametern innerhalb verschachtelten Funktionen:

```
def countTo(n: Int): Unit = {  
  2  def count(i: Int): Unit = {  
      if (i <= n) {  
        println(i)  
        count(i + 1)  
      }  
  6  }  
    }  
    count(1)  
  }  
10 countTo(5)
```

3.1.2 Parameter

- Funktionsparameter werden immer von Klammern eingeschlossen

call-by-value : Vorteil ist Vermeidung der wiederholten Auswertung von Argumenten

call-by-name : hat den Vorteil, dass es die Parameter nicht auswertet, sofern sie nicht in der Funktion verwendet werden

call-by-value ist effizienter als call-by-name, aber call-by-value kann in ∞ -loops geraten

- Scala benutzt per Def. call-by-value, aber kann auf call-by-name wechseln durch Voranstellung von `=>`:

```
1 def loop: Int = loop  
  
  def constOne(x: Int, y: => Int) = 1  
  
5 constOne(1, loop) // 1  
  constOne(loop, 1) // infy loop
```

3.1.3 Callback

Fkt. mit callback Fkt. haben folgende Syntax: `() => UNIT` und ist eine Funktion, die keine Parameter u. kein return-Wert hat

```
2 def oncePerSecond(callback: () => Unit) {  
    // danger, it's an endless loop :)  
    while (true) { callback(); Thread.sleep(10) }  
  }  
6
```



```
def timeFlies() {
  println("The dwarfs will pass away ...")
}
```

10

```
oncePerSecond(timeFlies) // start for 10 seconds the thread
```

3.1.4 Anonyme Funktionen

- doof, wenn man Funktionen einen Namen geben muss, wenn man sie nur einmal verwenden \Rightarrow hierfür sind anonyme Funktionen da :)
- die Anwesenheit von anonymen Funktion wird durch die Syntax: \Rightarrow gemacht

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread.sleep(1000) }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time to fly anonymous ..."))
  }
}
```

3.2 Schwanzrekursion

```
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
```

```
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n-1)
```

- gcd hat immer dieselbe Form, während bei factorial immer noch ein multiplikativer Faktor hinkommt
- bei Faktorial werden für die Multiplikatoren stets ein neuer Stack-Frame angelegt u. es braucht deshalb Platz proportional der Eingabe \Rightarrow ist deswegen keine Schwanzrekursion

\Rightarrow im folgenden Code wird nun gezeigt, wie man Schwanzrekursion erzeugen kann:

```
def factorial(i: Int): Int = {
  def fact(i: Int, accumulator: Int): Int = {
    if (i <= 1)
      accumulator
    else
      fact(i - 1, i * accumulator)
  }
  fact(i, 1)
}
```

```
}
```

3.3 Currying

- betrachten `sum`, bei der man die Grenzen `a` u. `b` nicht mehr angeben muss

```
def sum(f: Int => Int): (Int, Int) => Int = {  
2   def sumF(a: Int, b: Int): Int =  
    if (a > b) 0 else f(a) + sumF(a + 1, b)  
    sumF  
}
```

```
6 // nun kann man folgendes definieren  
def sumSquares = sum(x => x * x)
```

- wir werden nun Funktionen, die Funktionen zurückgegeben, behandelt?

```
sum(x => x * x)(1, 10)
```

hier wird `sum` zuerst zur Quadratfunktion (`x => x * x`) angewendet u. die resultierende Funktion wird dann auf die Argumentenliste `(1, 10)` angewendet

- für Funktionen, die Funktionen zurückgeben, hat Scala eine besondere Syntax u. die obige `sum`-Funktion kann auch wie folgt kürzer geschrieben werden

```
def sum(f: Int => Int)(a: Int, b: Int): Int =  
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

4 Klassen und Objekte

- Klasse für rationale Zahlen:

```
class Rational(n: Int, d: Int) {  
2  class RationalNumber(a: Int, b: Int) {  
    private def gcd(x: Int, y: Int): Int = {  
        if (x == 0) y  
        else if (x < 0) gcd(-x, y)  
6       else if (y < 0) gcd(x, -y)  
        else gcd(y%x, x)  
    }  
  
10    // defining a private variable which exists just in this class  
    private val g = gcd(a, b)  
    val number: Int = a/g  
    val denom: Int = b/g  
  
14    def +(juhu: RationalNumber) =  
        new RationalNumber(number + juhu.denom, denom + juhu.number)  
    }  
}
```

- **private members:** durch spezielles Keyword gekennzeichnet u. können nicht außerhalb der Klasse angesprochen werden

- **Objekterstellung**²⁹:

```
var test1 = new RationalNumber(3, 9)  
2 var test2 = new RationalNumber(3, 9)  
var test3: RationalNumber = test1.+(test2)  
var test4 = test1 + test2
```

⇒ mittels Punktnotation kann auf die Attribute zugegriffen werden

- **Vererbung:** jede Klasse erweitert eine Superklasse. Ist keine Klasse angegeben, so erbt es per *default* von `scala.AnyRef`

```
class RationalNumber(a: Int, b: Int) extends AnyRef {  
    ... // as before  
3 }
```

- eine Klasse erbt alle Methoden u. Variablen der Oberklasse, will man eine geerbte Methode überschreiben, so muss man das Schlüsselwort **override** verwenden

```
1 class Rational(n: Int, d: Int) extends AnyRef {  
    ... // as before  
    override def toString = "" + number + "/" + denom
```

²⁹erfolgt nach *lazy evaluation*

```
}
```

- **Abstrakte Klassen:**

```
abstract class IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
4 }
```

IntSet ist als abstrakte Klasse gekennzeichnet, d.h. von ihr können keine Objekte erzeugt werden

Implementierung einer abstrakten Klasse

```
class EmptySet extends IntSet {  
  def contains(x: Int): Boolean = false  
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new  
4   EmptySet)  
}
```

```
class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {  
  def contains(x: Int): Boolean =  
8    if (x < elem) left contains x  
    else if (x > elem) right contains x  
    else true  
  def incl(x: Int): IntSet =  
12   if (x < elem) new NonEmptySet(elem, left incl x, right)  
    else if (x > elem) new NonEmptySet(elem, left, right incl x)  
    else this  
}
```

- **traits:** traits sind wie abstrakte Klassen, nur dass sie dafür geschaffen wurde, um an andere Klassen ergänzt zu werden

```
1 trait IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
}
```

- **objects:** statt class kann man auch objects davor schreiben u. dadurch ist das **Singleton-Pattern** sichergestellt:

```
object EmptySet extends IntSet {  
  def contains(x: Int): Boolean = false  
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)  
4 }
```

- jede Deklaration ohne ein Sichtbarkeits/Scopewort ist per default *public*³⁰
- mit `equal` vergleicht man, ob Objekte den gleichen Werte besitzen
- mit `==` u. `!=` vergleicht man Wertgleichheit

4.1 Klassen

- werden in Paketen definiert u. spielen eine ähnliche Rolle wie in Java
- jedes Java-Paket ist auch eine Scala-Paket (vice versa)
- jede Klasse kann via *mixin* von mehr als einer Klasse erben, denn per default kann nur von einer Klasse gererbt werden
- jede Java-Klasse wird als gewöhnliche Scala-Klasse angesehen u. jedes Java-Interface kann als *Scala-trait* angesehen werden
- ein Beispiel für eine verschachtelte Klasse:

```

abstract class Widget {
  class Properties {
    import scala.collection.immutable.HashMap
4    private var values: Map[String, Any] = new HashMap
    def size = values.size
    def get(key: String) = values.get(key)
    def update(key: String, value: Any) = {
8      // Do some preprocessing, e.g., filtering .
      values = values.update(key, value)
      // Do some postprocessing.
    }
12  }
  val properties = new Properties
}

```

4.2 Klassenimport

- Syntax:

```

1 import scala.io._

```

- mehrere Klassen od. Objekte können vom selben Paket importiert werden, indem sie einfach in *brackets* geschrieben werden

4.3 Klassenhierarchie

- in Scala ist alles, bis auf eine Methode eine Instanz von einer Klasse \Rightarrow alle Primitiven aus Java (wie z.B. `int`) werden als Instanzen behandelt u. dies wird bei Kompilierung gemacht

³⁰dafür gibts kein Schlüsselwort

- Any ist die Top-Klasse, es hat zwei Unterklassen: AnyVal u. AnyRef
- AnyVal basiert auf *value classes*, also **boolean, byte, short, char, int, long, float, double**
- Mutterklasse aller Scala-Klassen ist Scala.Any
- am untersten Ende der Scala-Typen steht scala.Null u. scala.Nothing
- scala.Null ist ein *subtype* von allen Referenztypen
⇒ einzige Instanz ist die **null** Referenz
- scala.Nothing ist *subtypen* von jeden anderen Typen
⇒ von diesen Typen existieren keine Instanzen
- wenn man sich an die Namenskonventionen hält, dann sind die Scala-Repräsentanten der Primitiven Datentypen der JVM:
 - Int
 - Long
 - Double
 - Float
 - Boolean
 - Char
 - Byte

alle Unterklassen von der Klasse AnyVal

4.4 Overriding

- muss man dann hinschreiben, wenn abgeleitete Klassen Methoden, Feldern Variablen usw. von ihren Elternklassen überschreiben wollen
- überschreibt man etwas, ohne keyword **override** zu verwenden gibts einen Fehler
⇒ potentielle Fehler werden dadurch abgefangen
- Sachen, die als final deklariert sind, kann man nicht **overriden**

4.5 Abstraktion

- eine wichtige Aufgabe von Komponentensystemen ist, wie man von den erforderlichen Komponenten abstrahiert
- es gibt folgenden Formen der Abstraktion in Progg-Sprachen:
 1. *Parametrisierung* (typisch Funktional)
 2. *abstract members* (typisch objekt-orientiert)
- die folgende Klasse GenCell ist generisch

```

class GenCell[T](init: T) {
  private var value: T = init
3  def get: T = value
    def set(x: T): unit = {value = x}
}

```

- ebenso wie Klassen können auch Methoden Typenparameter besitzen.
die folgende Methode vertauscht den Inhalt von zwei Zellen:

```

def swap[T](x: GenCell[T], y: GenCell[T]): unit = {
  val t = x.get; x.set(y.get); y.set(t)
3 }

```

- Anwendung von swap:

```

1 val x: GenCell[int] = new GenCell[int](1)
  val y: GenCell[int] = new GenCell[int](2)
  swap[int](x,y)

```

Scala hat jedoch ein hochentwickeltes *type inference system*, welches die korrekten Typen anhand der Argumente erkennt \Rightarrow im obigen Codeschnipsel zur Anwendung der swap-Methode kann man die Typangaben in den *square brackets* auch weglassen

- **Varianz:**
 - Scala erlaubt die Varianz von Typparametern durch die Zeichen + u. -
 - + ... vor einem Parameter sagt aus, dass der Konstruktor *covariant* ist
 - - ... vor einem Parameter sagt aus, dass der Konstruktor *contravariant* ist

4.6 Companion Objekte

- wenn eine Klasse u. ein Objekte innerhalb einer Datei, im selben Packet den gleichen Namen haben, werden diese *Companion Objekte* genannt
- **Apply:** wird als *factory* Methode verwendet

```

1 type Pair[+A, +B] = Tuple2[A, B]
  object Pair {
    def apply[A, B](x: A, y: B) = Tuple2(x, y)
    def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
5 }

// neues Paar ohne new Konstruktor erstellen
val p = Pair(1, "one")

```

- **Unapply:** wirkt irgendwie als Extraktionmechanismus von bestimmten Werten einer Instanz \Rightarrow Pattern Matching benutzt diesen Mechanismus ausführlich

```

class Button(val label: String) extends Widget with Clickable {
3   def click () = {
      // Logic to give the appearance of clicking a button ...
    }
    def draw() = {
7      // Logic to draw the button on the display, web page, etc.
    }
    override def toString() = "(button: label="+label+", "+super.toString()+")"
  }
11
  object Button {
    def unapply(button: Button) = Some(button.label)
  }

```

4.7 Komposition

⇒ hat flexible modulare Mixin-Komposition Konstrukte für Klassen-Komposition

- fangen einfach mal mit einem kleinen Beispiel an:

```

1  traits AbsIterator[T] {
    def hasNext: boolean
    def next: T
  }

```

- **traits** ist eine spezielle Form einer abstrakten Klasse, welche keine Werte für den Parameter für den Konstruktor hat
- traits können in allen Kontexten verwendet werden, in denen abstrakte Klassen auftauchen
- nur traits können als mixins verwendet werden

- **mixin-class composition:** betrachten folgende Iteratoren

```

trait RichIterator[T] extends AbsIterator[T] {
    def foreach(f: T => unit): unit =
      while (hasNext) f(next)
4  }

```

```

# ein konkreter Iterator, der sukzessive die Zeichen eines Strings returnd
8  class StringIterator(s: String) extends AbsIterator[Char] {
    private var i = 0
    def hasNext = i < s.length
    def next = {val x = s.charAt(i); i = i + 1; x}
12 }

```

- nun wollen die Funktionen des RichIterators und des StringIterators in einer Klasse verwenden \Rightarrow mit Einfachvererbung u. Interfaces kann man das nicht machen
- Idee: *mixin-class composition*

```

object Test {
  def main(args: Array[String]): unit = {
    class Iter extends StringIterator(args(0)) with RichIterator[Char]
4    val iter = new Iter
    iter foreach System.out.println
  }
}

```

4.8 Dekomposition

von Objekten via Pattern-Matching

- wollen einen simplen Taschenrechner für algebraische Berechnungen u. der Plus-Operation implementieren:

```

1 abstract class Term {
  def eval: Int
}

5 class Num(x: Int) extends Term {
  def eval: Int = x
}

9 class Plus(left: Term, right: Term) extends Term {
  def eval: Int = left.eval + right.eval
}

```

- so ein Ansatz verlangt, dass alle Operationen zu einer bestimmten Struktur durchwandert werden
 - intern definierte Methoden müssen deswegen ebenfalls ungewollt durch die ganze Struktur gelegt werden \Rightarrow DRY wird verletzt

Pattern Matching über Klassenhierarchie

- in FP sind Datenstrukturen von ihren Operationen getrennt
- während Datenstrukturen gewöhnlich durch algebraische Datenstrukturen definiert sind, benutzen Operationen auf solchen Datentypen *pattern matching* als Grundprinzip der Dekomposition
- durch *pattern matching* kann man eine einzelne eval-Funktion implementieren, ohne das künstliche Zusatzfunktion aufzusetzen
- Klassen werden mit **case** "getagt":

```

1 abstract class Term
  case class Num(x: Int) extends Term
  case class Plus(left: Term, right: Term) extends Term

5 # dann ist folgendes möglich
  Plus(Plus(Num(1), Num(2)), Num(3))

```

- nun folgt die Implementierung der eval-Funktion nach dem Pattern-Matching Prinzip:

```

object Interpreter {
2   def eval(term: Term): Int = term match {
        case Num(x) => x
        case Plus(left, right) => eval(left) + eval(right)
    }
6 }

```

- der matchende Ausdruck x **match** $\{ \text{case } pat_1 \Rightarrow e_1 \text{ case } pat_2 \Rightarrow e_2 \dots \}$ matcht den Wert x gegen die Muster pat_1, pat_2, \dots
 \Rightarrow dadurch können neue Funktionen leicht zu einem bestehenden System hinzugefügt werden

4.9 Case-Klassen

- es wird einfach das Schlüsselwort **case** vor Klassen bzw. Objekten geschrieben
- Case-Klassen haben implizit eine Konstruktor-Funktion, welche denselben Namen wie die Klasse trägt
- Case-Klassen u. Case-Objekte haben implizit die Methoden `toString`, `equals` u. `hashCode` implementiert u. überschreiben die Methoden von `AnyRef`; sie haben implizite getter-Methoden, um an die Argumente der Konstruktoren zu gelangen
- Instanzen von Case-Klassen können ohne die `new`-Anweisung erzeugt werden

```

case class Stuff(name: String, age: Int)

2 # erzeuge eine Instanz
  val s = Stuff("Arsch", 24)

6 # equals-Methode anwenden
  s == Stuff("Arsch", 24)

# Zugriff auf die Member-Variablen
10 s.name; s.age;

# eigene Klasse schreiben, die das gleiche leistet wie die case-Klasse
class Stuff(val name: String, val age: Int) {

```

```

14  override def toString = "Stuff("+name+","+age+)"
    override def hashCode = name.hashCode + age
    override def equals(other: AnyRef) = other match {
        case s: Stuff => this.name == s.name && this.age == s.age
18  case _ => false
    }
}

```

- Case-Klassen erlauben die Erstellung von *patterns*, welche zu case-class Konstruk-toren gehören:

```

case class Point(x: Double, y: Double)
abstract class Shape() {
3  def draw(): Unit
}

case class Circle(center: Point, radius: Double) extends Shape() {
7  def draw() = println("Circle.draw: " + this)
}

case class Rectangle(lowerLeft: Point, height: Double, width: Double) extends
    Shape() {
11  def draw() = println("Rectangle.draw: " + this)
}

case class Triangle(point1: Point, point2: Point, point3: Point)
15  extends Shape() {
    def draw() = println("Triangle.draw: " + this)
}

19  val shapesList = List(
    Circle(Point(0.0, 0.0), 1.0),
    Circle(Point(5.0, 2.0), 3.0),
    Rectangle(Point(0.0, 0.0), 2, 5),
23  Rectangle(Point(-2.0, -1.0), 4, 3),
    Triangle(Point(0.0, 0.0), Point(1.0, 0.0), Point(0.0, 1.0)))

val shape1 = shapesList.head // grab the first one.
27  println("shape1: "+shape1+". hash = "+shape1.hashCode)
    for (shape2 <- shapesList) {
        println("shape2: "+shape2+". 1 == 2 ? "+(shape1 == shape2))
    }

```

4.10 Pattern Matching

- ist eine generalisierte switch-Anweisung für Klassenhierarchien
- anstelle der switch-Anweisung gibt es eine **match**-Operation

```

44 match {
    case 44 => true // if we match 44, the result is true
    case _ => false // otherwise the result is false
4 }

// pattern-Match fuer Klassen
8 Stuff("David", 45) match {
    case Stuff("David", 45) => true
    case _ => false
}

12 // koennen den Namen testen, wobei uns der zweite Parameter (age) rille ist
Stuff("David", 45) match {
    case Stuff("David", _) => "David"
16 case _ => "Other"
}

// koennen das age field extrahieren und in die howold-Variable schreiben
20 Stuff("David", 45) match {
    case Stuff("David", howOld) => "David, age: "+howOld
    case _ => "Other"
}

24 // koennen einen Guard setzen
Stuff("David", 45) match {
    case Stuff("David", age) if age < 30 => "young David"
28 case Stuff("David", _) => "old David"
    case _ => "Other"
}

```

- man kann gegen match so ziemlich alles testen, was es da so gibt

4.11 traits

- in Scala kann man traits in Objekte erst bei Instanziierung einbinden
- traits können auch als abstrakt deklariert haben
- traits haben ansonsten den gleiche Aufbau wie eine normale Klasse
- traits unterstützen nicht beliebige Konstruktoren u. nehmen auch keine Argumente für ihre Konstruktoren auf
- traits können keine Argumente an ihre Elternklassen weitergeben
- **Erstellung von traits:**

```

trait T1 {

```

```

2   println( "    in T1: x = " + x )
   val x=1
   println( "    in T1: x = " + x )
}

6

trait T2 {
   println( "    in T2: y = " + y )
   val y="T2"
10  println( "    in T2: y = " + y )
}

class Base12 {
14  println( " in Base12: b = " + b )
   val b="Base12"
   println( " in Base12: b = " + b )
}

18

class C12 extends Base12 with T1 with T2 {
   println( " in C12: c = " + c )
   val c="C12"
22  println( " in C12: c = " + c )
}

println( "Creating C12:" )
26 new C12
   println( "After Creating C12" )
   /*
   Creating C12:
30   in Base12: b = null
      in Base12: b = Base12
      in T1: x = 0
      in T1: x = 1
34   in T2: y = null
      in T2: y = T2
      in C12: c = null
      in C12: c = C12
38 After Creating C12
   */

```

⇒ Reihenfolge der Abarbeitung der traits ist von links nach rechts

- **Klassen od. traits?** falls ein traits mehr als einmal als Elternteil von anderen Klassen dient, so als Klasse machen; vermeide konkrete Felder in traits, welche nicht mit geeigneten default-Werten initialisiert werden können ⇒ verwende lieber abstrakte Felder

5 Generische Typen und Methoden

- haben einen Stack für Integer

```
abstract class IntStack {  
    def push(x: Int): IntStack = new IntNonEmptyStack(x, this)  
    def isEmpty: Boolean  
4    def top: Int  
    def pop: IntStack  
}  
class IntEmptyStack extends IntStack {  
8    def isEmpty = true  
    def top = error("EmptyStack.top")  
    def pop = error("EmptyStack.pop")  
}  
12 class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {  
    def isEmpty = false  
    def top = elem  
    def pop = rest  
16 }
```

- kann man diesen Stack generisch auch für andere Typen machen? Jo, dazu benutzen wir einfach **Typen Parameter**

```
abstract class Stack[A] {  
    def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)  
    def isEmpty: Boolean  
4    def top: A  
    def pop: Stack[A]  
}  
class EmptyStack[A] extends Stack[A] {  
8    def isEmpty = true  
    def top = error("EmptyStack.top")  
    def pop = error("EmptyStack.pop")  
}  
12 class NonEmptyStack[A](elem: A, rest: Stack[A]) extends Stack[A] {  
    def isEmpty = false  
    def top = elem  
    def pop = rest  
16 }
```

```
val x = new EmptyStack[Int]  
val y = x.push(1).push(2)
```

- dasselbe Prinzip kann man auch bei Methoden anwenden u. generische Methoden sind auch ein Ausdruck von Polymorphi:

```
1 def isPrefix [A](p: Stack[A], s: Stack[A]): Boolean = {
```

```

    p.isEmpty ||
    p.top == s.top && isPrefix[A](p.pop, s.pop)
  }

```

5.1 Annotationen Varianz

- schreiben wir + vor einen Typ Parameter so ist dies ein Zeichen dafür, dass Subtyping Kovariant in diesem Parameter ist

```
class Stack[+A]
```

5.2 Tuples

- manchmal möchte man, dass eine Funktion mehr als ein Ergebnis zurückgibt

```

case class Tuple2[A, B](_1: A, _2: B)
def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
3 val xy = divmod(x, y)
  println("quotient: " + xy._1 + ", rest: " + xy._2)

```

- ein anderes Beispiel mit der tupleator Methode, die aus den Argumenten ein Tupel entsprechend der Anzahl der Argumente generiert

```

def tupleator(x1: Any, x2: Any, x3: Any) = (x1, x2, x3)
3 val t = tupleator("Hello", 1, "2.3")
  println( "Print the whole tuple: " + t )
  println( "Print the first item: " + t._1 )
  println( "Print the second item: " + t._2 )
7 println( "Print the third item: " + t._3 )

val (t1, t2, t3) = tupleator("World", '!', 0x22)
println( t1 + " " + t2 + " " + t3 )

```

- um auf Elemente von Tuppeln zuzugreifen verwendet man `t._N`, wobei N für das gewünschte Element steht
- n

5.3 Wann man explizite Typannotationen braucht

- bei einer reinen Variablendeklaration, ohne Wertzuweisung
- Alle Methodenparameter

6 Listen und der Spaß mit der Unveränderlichkeit

- Listen sind ähnlich wie Arrays in anderen Sprachen

```
val fruits = List("apples", "oranges")  
2 val nums = List (1,2,3,4,5,6,7,8,9,10)  
val div = List(List (1,2) , List("String"), 1)
```

- Listen in Scala unterscheiden sich aber in folgenden Punkten von anderen Sprachen:
 1. Listen sind unveränderbar
 2. Listen haben eine rekursive Struktur
 3. Listen haben viel mehr Operationen als Arrays
- Listen sind nicht build in, sondern werden durch die Abstrakte Klasse List definiert
- benefits Unveränderlichkeit:
 - es wird weniger globale Zustände geben
 - weniger Dinge können geändert werden
 - Funktionen werden weniger anfällig für globale Zustände von Variablen u. Funktionen werden mehr transformativ \Rightarrow Methoden referenzieren viel weniger auf den externen Zustand von Variablen
 - solche Methoden sind leichter mit automatischen Tests durchzuführen (ScalaCheck)

6.1 Scala Listen, Tupel and Map-Klassen

- Collections (Listen) sind Container für Dinge

```
1 val x = List (1,2,3,4)  
x. filter (a => a % 2 == 0) // List[Int] = List(2, 4)  
  
val a = Array(1,2,3) // Arrays fangen bei null an  
5 a(1) // 2  
  
val m = Map("one" -> 1, "two" -> 2, "three" -> 3)
```

- es gibt lazy collections (z.B. Range), d.h. für diese wird erst dann Speicherplatz angelegt, wenn auf diese das erste mal zugegriffen wird

```
1 0 to 10  
// Range.Inclusive = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

6.2 List[T]

- List[T] stellen eine verkettete Liste vom Typ T dar, d.h. ist eine sequentielle Liste welche Javas primitive Datentypen beinhaltet (Int, Float, Double), da sich um **boxing** (also die Umwandlung von primitiven Datentypen in Objekten) kümmert
- **Listen-Konstruktoren:** Nil ist Repräsentant für eine leere Liste, :: (cons genannt) z.B. x :: xs, d.h. ist eine Liste, bei der das erste Element x ist u. gefolgt wird aus (Elemente der) Liste xs

```
1 :: 2 :: 3 :: Nil
2 // List[Int] = List(1, 2, 3)

// obiger Code wird auf folgende weisse ausgewertet
new ::(1, new ::(2, new ::(3, Nil)))
```

⇒ das letzte Element einer "cons" muss immer einer Liste sein (mit Nil drücken wir die leere Liste aus)

- Listen sind *homogen*, d.h. innerhalb einer Liste müssen alle Elemente vom selben Typ sein

```
val fruits : List[Int] = List("apples", "oranges") // Error
val fruits : List[String] = List("apples", "oranges") // funzt
```

- Liste mit verschiedenen Typen erstellen: Da nehmen wir eine List mit der apply-Methode (welche wie folgt definiert ist: def apply[T](param: T*): List[T])

```
List(1, 44.5, 8d)
List[AnyVal] = List(1, 44.5, 8.0)
```

- angenommen wir wollen an eine beliebige Liste ein Item dranhängen:

```
val x = List(1,2,3)
2 99 :: x

// List[Int] = List(99, 1, 2, 3)
```

die alte Variable x bleibt unverändert und an die neue Liste mit den Wert 99 als head wird einfach die alte Liste x drangehangen ⇒ diese Operation läuft in O(1)

- Listen mergen geht mit den :::-Operator:

```
val x = List(1,2,3)
val y = List(99, 98, 97)
x ::: y
```

- *typische Listen-Operatoren:*

```
1 List (1,2,3) . filter (x => x % 2 == 1)
  // List[Int] = List(1, 3)
```

```
5 List (1,2,3) . remove(x => x % 2 == 1)
  // List[Int] = List(2)
```

- filter funzt bei jeder Collection, welche einen bestimmten Typ enthält:

```
"99 Red Balloons".toList. filter (Character.isDigit)
2 // List[Char] = List(9, 9)
```

⇒ wir konvertieren einen String in List [Char] u. filtern via Methode aus Java aus dieser Char-Liste die Zahlen heraus

6.3 Transformation

- map Funktion transformiert alle Elemente einer Collection basierend auf eine Funktion. Sollte die Funktion, welche an map übergeben wird, einen anderen Typ zurückgeben, als die ursprüngliche Collection, so wird der Typ der Funktion zurückgegeben

```
1 List("A", "Cat").map(s => s.toLowerCase)
  // kuerzer: List("A", "Cat").map(_.toLowerCase)
  // List[java.lang.String] = List(a, cat)

5 List("A", "Cat").map(_.length)
  // List[Int] = List(1, 3)
```

- mit Listen kann man ebenfalls komplexe Datenbankabfragen machen u. uns Elemente in einer Liste ausgeben lassen:

```
trait Person {def first : String }

2
val d = new Person {def first = "David" }
val e = new Person {def first = "Elwood"}
val a = new Person {def first = "Archer"}

6
List(d, e, a).map(n => <li>{n.first}</li>)
```

- List hat eine sort-Methode:

```
1 List(99, 2, 1, 45).sort(_ < _)
  // List[Int] = List(1, 2, 45, 99)

List("b", "a", "elwood", "archer").sort(_ < _)
5 //List[java.lang.String] = List(a, archer, b, elwood)
```

```
List("b", "a", "elwood", "archer").sort(_.length > _.length)
// List(archer, elwood, a, b)
```

nun noch eine runde komplexer: Wir wollen alle validen Persons-Records aus der DB, welche nach Alter sortiert sind und dabei den Namen ausgeben:

```
trait Person {
  def age: Int
  def first : String
4   def valid: Boolean
}

def validByAge(in: List[Person]) =
8   in. filter (_.valid).
    sort(_.age < _.age).
    map(_.first)
```

- **reduceLeft**: Operation auf adjazenten Elemente einer Collection durchführen, d.h. nehme mir zwei Elemente der Liste, führe die entsprechende Operation aus u. gehe dann zum nächsten Element und mache dasselbe nochmal solange bis die Liste keine Elemente mehr besitzt:

```
List(8, 6, 22, 2).reduceLeft(_ max _)
2 // Int = 22

// reduceLeft verwenden, um das laengste Wort zu finden
List("moose", "cow", "A", "Cat").
6 reduceLeft((a, b) => if (a.length > b.length) a else b)
// java.lang.String = moose
```

- **foldLeft** arbeitet wie **reduceLeft**, nur dass es einen *Seed* als Startpunkt nimmt, wobei der Seed-Typ den Rückgabewert von **foldLeft** u. nicht von der Funktion bestimmt

```
1 List (1,2,3,4) .foldLeft(2) (_ + _)
  // Int = 12

List (1,2,3,4) .foldLeft(1) (_ * _)
5 // Int = 24

List("b", "a", "elwood", "archer").foldLeft(0)(_ + _.length)
```

- Erstellung einer geschachtelten Collection:

```
1 val n = (1 to 3).toList
  //List[Int] = List(1, 2, 3)
```

```

n.map(i => n.map(j => i * j))
5 // List[List[Int]] = List(List(1, 2, 3), List(2, 4, 6), List(3, 6, 9))

```

wollen wir die Ergebnisse einer geschachtelten Schleife platten, so die flatMap-Methode verwenden

- **for-Comprehension**

- angenommen wir haben eine Liste von Personen mit namen u. age Feldern u. wir wollen die Namen aller Personen ausgeben, die alle über 20 sind

```

for (p <- persons if p.age > 20) yield p.name

```

- genereller Aufbau von for-comprehension:

```

for(s) yield e

```

s ... ist eine Sequenz von *Generatoren*, *Definitionen* u. *Filtern*

- ein *Generator* hat die Form `val x <- e`, wobei e eine Liste mit Werten ist u. an x werden sukzessiv die Elemente aus e gehangen
- eine *Definition* hat die Form `val x = e`, d.h. x ist ein Name für die Werte von e
- ein *Filter* ist ein Ausdruck f vom booleschen Typ
- angenommen wir wollen das Produkt der Zahlen von 1 bis 10 zwischen den geraden u. ungeraden Zahlen bilden:

```

def isOdd(in: Int) = in % 2 == 1
def isEven(in: Int) = !isOdd(in)
3 val n = (1 to 10).toList

// dirty-Variante
n. filter (isEven).flatMap(i => n. filter (isOdd).map(j => i * j))

7 // for-comprehension
for {i <- n if isEven(i); j <- n if isOdd(j)} yield i * j
// List[Int] = List(2, 6, 10, 14, 18, 4, 12, 20, 28, 36, 6, 18, 30, 42, 54,
// 8, 24, 40, 56, 72, 10, 30, 50, 70, 90)

```

⇒ for-Comprehension ist keine Schleifenkonstrukt sondern einfach nur eine syntaktische Vereinfachung

6.4 Tupel

- wollen eine Funktion schreiben, die 3 return-Werte hat: einen Zähler, die Summe u. die Quadrate:

```

1 def sumSq(in: List[Double]): (Int, Double, Double) =
    in.foldLeft((0, 0d, 0d))((t, v) => (t._1 + 1, t._2 + v, t._3 + v * v))

// koennen obige Funktion durch Scalas pattern-Matching lesbarer machen

```

5

```
def sumSqReadable(in: List[Double]) : (Int, Double, Double) =
  in.foldLeft ((0, 0d, 0d)) {
    case ((cnt, sum, sq), v) => (cnt + 1, sum + v, sq + v * v) }
```

- der Compiler übersetzt (Int, Double, Double) in Tuple3[Int, Double, Double]
- foldLeft hat zwei Parameter: t... steht für Tuple3[Int, Double, Double]
- der Rückgabewert der Funktion ist ein neues Tupel

- man kann Tupel auf viele verschiedene Arten anlegen:

```
Tuple2(1,2, 2.0, "ww") == Pair(1,2)
(1,2) == (1,2)
(1,2) == 1 -> 2
```

6.5 Map[K,V]

- die Map Klasse in Scala ist unveränderlich
- ein Map ist eine Sammlung von key/value Paaren
- jeder beliebige value kann von einem eindeutigen Schlüssel beschrieben werden

```
var p = Map(1 -> "David", 9 -> "Elwood")
// Map[Int,java.lang.String] = Map(1 -> David, 9 -> Elwood)

4 // an eine Map noch ein Element dranhaengen
  p + 2 -> "test"
  // Map[Int,java.lang.String] = Map(1 -> David, 9 -> Elwood, 2 -> test)

8 p(1)           // funzt
  p(88)          // exception
  p.get(88)       // Option[java.lang.String] = None
  p -= 9          // key-value entfernen
12 p.contains(1) // true
```

- wenn wir auf einen Schlüssel zugreifen wollen, den es nicht gibt wird eine Exception geworfen (was auch Sinn macht, denn man kann ja nicht auf was zugreifen, was es gar nicht gibt):

macht man den Map-Zugriff mit get, so wird die Option (Some od. None) zurückgegeben

- mit der -= key kann man Elemente aus einer Map entfernen
 - mit .contains kann man testen, ob ein Schlüssel in der der Map enthalten ist
- java.util.NoSuchElementException: key not found: ...

6.6 Option[T]

- ist eine mächtige Alternative zu Javas null
- Option hat nur die Werte Some[T] od. None haben
- None ist ein Objekt u. in einem Scala Programm gibt es nur eine Instanz von None

7 Spaß mit Funktionen

- Funktionen sind in Scala eine Instanz von Klassen:

```
// erstellen eine Funktion und weisen dieser einer Variable zu
val f: Int => String = x => "Dude: " + x
// f: (Int) => String = <function>

4 // rufen eine Methode auf dieser Methode auf
  f.toString
  // java.lang.String = <function>

8 // nun vergleichen wir Methoden
  f == f

12 f(24)
```

- Funktionen als Parameter übergeben

- im folgenden Beispiel definieren wir eine Methode w42, die eine Funktion als Parameter übernimmt, welche Int als Input hat u. String zurückgibt

```
def w42(f: Int => String) = f(42)
// w42: ((Int) => String)String

4 // fm nimmt einen Int als Input und gibt String zurueck
  def fm(i: Int): String = "fm:" + i
  // def fm(i: Int): String = "fm:" + i

8 // nun erstellen wir eine Funktion, die w42 uebergeben wird und als
  Ergebnis die Rueckgabe von Funktion fm hat
  w42((i: Int) => fm(i))
  // String = fm:42
```

7.1 Partielle Anwendungen und Funktionen

- Methoden u. Funktionen sind in Scala etwas verschiedenes
- in Scala ist alles bis auf Methoden eine Instanz \Rightarrow Methoden sind keine Funktionen
- Methoden werden an Instanzen angehängen u. können auf Instanzen angewendet werden

- in Scala können wir partiell angehauchte Funktionen von Methoden her ableiten - betrachten folgende Funktionen:

```

def plus(a: Int, b: Int) = "Result is: " + (a+b)
2
val p = (b: Int) => plus(42, b)

```

p braucht einen zweiten Parameter, um die Anforderungen an die Funktion plus 42 zu erfüllen u. wir sagen p ist partielle Anwendung von plus

- partielle Methodendefinitionen können mit der folgenden Syntax besser beschrieben werden:

```

1 def add(a: Int)(b: Int) = "Result is: " + (a + b)
  // add: (Int)(Int)java.lang.String

```

durch diese Notation kann man Codeblöcke als Parameter übergeben:

```

def add(a: Int)(b: Int)(c: String) = "Result is: " + (a + b) + " " + c
2

add(1){
  val r = new java.util.Random
6  r.nextInt(100)
  }("Arsch")

```

7.2 Funktionen und Typparameter

- Methoden können Typparameter haben
- Typparameter definieren den Typ von Parametern od. den Rückgabewert der Funktion:

```

1 val f: Int => String = x => "Dude: " + x
  val g: Int => Double = x => 20.0
  def t42[T](f: Int => T): T = f(42)
  // t42: [T]((Int) => T)T
5
  t42(f)
  // String = Dude: 42

9  t42(g)
  // Double = 20.0

  t42(1 +)
13 // Int = 43

```

7.3 Funktionen in Container packen

- Funktionen sind Instanzen u. deswegen kann man alles, was man mit Instanzen machen kann auch mit Funktionen machen
- Im folgenden Erstellen wir ein Array von Funktionen:

```
def bf: Int => Int => Int = i => v => i + v
// (Int) => (Int) => Int
3
val fs = (1 to 100).map(bf).toArray
// fs: Array[(Int) => Int] = Array(<function>, <function>, <function>,
    <function>, <function>, <function>, ...
```

8 Pattern-Matching

- bisher haben wir die Ecksteine der funktionalen Programmierung abgegrast: unveränderliche Datentypen u. Funktionen als Parameter von Funktionen
- als erstes Beispiel gucken wir uns die Fibonaccizahlen in verschiedenen Varianten an:

```
// normale Fibos
def fibonacci(in: Int): Int = in match {
3   case 0 => 0
    case 1 => 1
    case n => fibonacci(n - 1) + fibonacci(n - 2)
}
7
// fibo mit guards
def fib2(in: Int): Int = in match {
    case n if n <= 0 => 0
11  case 1 => 1
    case n => fib2(n - 1) + fib2(n - 2)
}
```

- **Matching Any Type:**

```
def myMules(name: String) = name match {
    case "Elwood" | "Madeline" => Some("Cat")
3   case "Archer" => Some("Dog")
    case "Pumpkin" | "Firetruck" => Some("Fish")
    case _ => None
}
```

- **Datentypen testen:** schreiben eine Methode, die testet, ob ein hereinkommendes Objekt ein String, Integer od. was anderes ist

```

def test2(in: Any) = in match {
2   case s: String => "String, length "+s.length
    case i: Int if i > 0 => "Natural Int"
    case i: Int => "Another Int"
    case a: AnyRef => a.getClass.getName
6   case _ => "null"
}

```

- Case-Klassen gehören auch hierunter u. wurden an anderer Stelle in diesem Buch bereits erklärt. Hier wird nun beschrieben wie man match anwenden kann:

```

1 case class Person(name: String, age: Int, valid: Boolean)

def older(p: Person): Option[String] = p match {
    case Person(name, age, true) if age > 35 => Some(name)
5   case _ => None
}

```

- **Pattern-Matching in Listen:**

```

def sumOdd(in: List[Int]): Int = in match {
2   case Nil => 0
    case x :: rest if x % 2 == 1 => x + sumOdd(rest)
    case _ :: rest => sumOdd(rest)
}

6

def noPairs[T](in: List[T]): List[T] = in match {
    case Nil => Nil
10   case a :: b :: rest if a == b => noPairs(a :: rest)
        // the first two elements in the list are the same, so we will
        // call noPairs with a List that excludes the duplicate element
    case a :: rest => a :: noPairs(rest)
14   // return a List of the first element followed by noPairs
        // run on the rest of the List
}

18 noPairs(List (1,2,3,3,3,4,1,1) )
    // List[Int] = List(1, 2, 3, 4, 1)

```

- **verschachteltes Pattern-Matching in case-Klassen:**

```

1 // override val ist mit de haesslichste Syntax in Scala
case class MarriedPerson(override val name: String,
    override val age: Int,
    override val valid: Boolean,
5   spouse: Person) extends Person(name, age, valid)

```

```

def mOlder(p: Person): Option[String] = p match {
9   case Person(name, age, true) if age > 35 => Some(name)
    case MarriedPerson(name, _, _, Person(_, age, true))
      if age > 35 => Some(name)
    case _ => None
13 }

```

9 Varianz

⇒ es legt Regeln fest, nach denen parametrisierte Typen als Parameter übergeben werden können

9.1 Invariante Parametertypen

- in Scala ist `Array[T]` *invariant*, d.h. man kann nur `Array[String]` an `foo(a: Array[String])` übergeben
- invariante Typparameter schützen uns, wenn wir mit veränderlichen Datentypen arbeiten

9.2 Kovariante Parametertypen

- sind gekennzeichnet durch ein `+` vor dem Typparameter, z.B. `List[+T]`
- ein kovarianter Typ ist nützlich, wenn wir es mit einem *read-only* container zu tun haben

```

class Getable[+T](val data: T)
2 def get(in: Getable[Any]) {println("It's " + in.data)}
  val gs = new Getable("String")

```

9.3 Kovariante Methoden

- **kovariant** heißt, dass der return-Wert von Funktionen in abgeleiteten Klassen geändert werden kann:

```

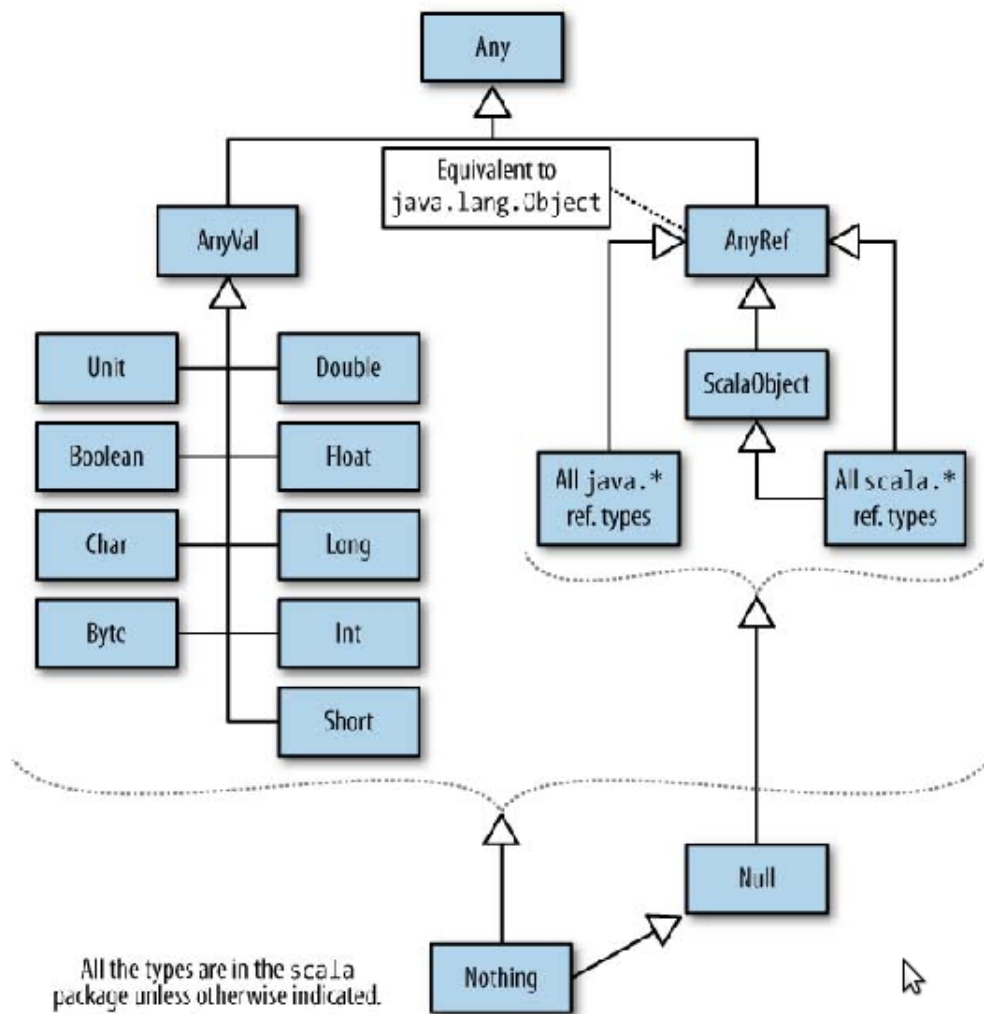
abstract class A { def f(a: A): Any; }
class B extends A { def f(a: A): A = a; }
class C extends B { override def f(a: A): C = this; }

```

- Methoden sind aber niemals in ihren Parametern kontravariant, d.h. es können keine weiteren Parameter ergänzt werden

9.4 Varianzregeln

- veränderliche Container sollten invariant sein
- unveränderliche Container sollten kovariant sein
- die Inputs von Transformationen sollten kontravariant sein u. die Outputs von Transformationen sollten kovariant sein



10 Scalas Objekt System

- das sogenannte **Predef Objekt** lädt automatisch wichtige Sachen in ein Scala Programm
- Objekte werden automatisch und *lazy* zur Laufzeit instanziiert
-

10.1 Typhierarchie

siehe folgendes Bild

- alle **AnyVal** Instanzen sind immutable u. alle **AnyValue** Typen sind *abstract final*

10.2 Linearisierung der Objekthierarchie

betrachte folgendes Beispiel:

```

1  class C1 {
    def m = List("C1")
  }

5  trait T1 extends C1 {
    override def m = { "T1" :: super.m }
  }

9  trait T2 extends C1 {
    override def m = { "T2" :: super.m }
  }

13 trait T3 extends C1 {
    override def m = { "T3" :: super.m }
  }

17 class C2 extends T1 with T2 with T3 {
    override def m = { "C2" :: super.m }
  }

21 val c2 = new C2
    println(c2.m)

// List(C2, T3, T2, T1, C1)

25

class C1 {
    def m(previous: String) = List("C1("+previous+")")
29 }
trait T1 extends C1 {
    override def m(p: String) = { "T1" :: super.m("T1") }
  }
33 trait T2 extends C1 {
    override def m(p: String) = { "T2" :: super.m("T2") }
  }
trait T3 extends C1 {
37   override def m(p: String) = { "T3" :: super.m("T3") }
  }
class C2 extends T1 with T2 with T3 {
    override def m(p: String) = { "C2" :: super.m("C2") }
41 }
val c2 = new C2
    println(c2.m(""))

45 // List(C2, T3, T2, T1, C1(T1))
    // haetten eigentlich mit C1(T3) gerechnet

```

hier ist der Linearisierungsalso, den Scala verwendet:

1. Put the actual type of the instance as the first element.
2. Starting with the rightmost parent type and working left, compute the linearization of each type, appending its linearization to the cumulative linearization. (Ignore `ScalaObject`, `AnyRef`, and `Any` for now.)
3. Working from left to right, remove any type if it appears again to the right of the current position.
4. Append `ScalaObject`, `AnyRef`, and `Any`.

11 FP in Scala

11.1 Was FP ist

- haben keine Seiteneffekte, d.h. Analyse, Testen u. debuggen werden leichter
- *referential transparency*, d.h. man kann eine Funktion in jeden beliebigen Kontext aufrufen u. muss keine Sorgen um den Kontext machen in dem die Funktion aufgerufen wird
- in FP sind Variablen immutable

11.2 FP in Scala

- Merke: eine Funktion, die `Unit` zurückgibt, hat pure Seiteneffekte, denn ansonsten ist die Funktion sinnlos, da sie ja nichts zurückgibt

```
List(1, 2, 3, 4, 5) map { _ * 2 }
```

```
// _ * 2 ist ein shortcut fuer i => i * 2!!!!
```

man kann die anonyme Funktion auch auf ein `val` drücken.

```
1 var factor = 3
  val multiplier = (i: Int) => i * factor
  val l1 = List(1, 2, 3, 4, 5) map multiplier
```

```
5 factor = 5
  val l2 = List(1, 2, 3, 4, 5) map multiplier
```

```
println(l1)
9 println(l2)
```

```
// List(3, 6, 9, 12, 15)
// List(5, 10, 15, 20, 25)
```

⇒ Faktor ist kein formaler Parameter sondern eine Referenz zu einer Variable in einem bestimmten scope, d.h. der Compiler kreiert ein *closure*

11.3 Rekursion

- verhindert mutable Variablen
- aber Performanz-Overhead u. Risiko des Stackoverflows
- Performanz-Prosb können mit *memorization* u. Stackoverflows können durch Umwandlung in eine spezielle Schleife (Tail Calls u. Tail-Call Optimierung) verbessert werden

11.4 Tail Calls und Tail-Call Optimierung

- Tail Call, d.h. wenn eine Funktion sich erst bei ihren finalen Durchlauf selbst aufruft \Rightarrow es ist die leichteste Art der Rekursion, die man in eine normale Schleife umwandeln kann
- Schleifen verhindern die Gefahr eines Stackoverflows

```
// ohne tail-Rekursion – Mult nach rek. Aufruf ist bloed
def factorial (i: BigInt): BigInt = i match {
3   case _ if i == 1 => i
    case _ => i * factorial (i - 1)
}

7 // mit tail-Rekursion – Mult vor rek. Aufruf ist gut
def factorial (i: BigInt): BigInt = {
    def fact(i: BigInt, accumulator: BigInt): BigInt = i match {
        case _ if i == 1 => accumulator
11    case _ => fact(i - 1, i * accumulator)
    }
    fact(i, 1)
}
```

11.5 Funktionale Datenstrukturen

- **Listen:** tauchen ganz oft in FP auf

```
val list1 = List("Programming", "Scala")
2 val list2 = "People" :: "should" :: "read" :: list1
val list2 = ("People" :: ("should" :: ("read" :: list1 )))
val list2 = list1 ::( "read") ::( "should") ::( "People")
println( list2 )

6 // List[java.lang.String] = List(People, should, read, Programming, Scala) –
    kommt bei allen Defs heraus
```

- **hash/dictionary/map**
- **Mengen:** sind wie Listen, aber ihnen kann jedes Element nur einmal vorkommen; Element Iteration geht in $O(n)$

11.6 Traversieren, Mapping, Filtering, Folding und Reducing

- traversieren geht mit der foreach:

```
List(1, 2, 3, 4, 5) foreach { i => println("Int: " + i) }
val stateCapitals = Map(
3   "Alabama" -> "Montgomery",
    "Alaska" -> "Juneau",
```



```

    "Wyoming" -> "Cheyenne")
stateCapitals foreach { kv => println(kv._1 + ": " + kv._2) }

7

// The signature of foreach is the following:
trait Iterable[+A] {
11  ...
    def foreach(f : (A) => Unit) : Unit = ...
    ...
}

```

- map: wandelt eine Collection in eine neue Collection um, in der die Anzahl der Elemente gleich der Anzahl der Ursprungscollection ist

```

1 val stateCapitals = Map(
    "Alabama" -> "Montgomery",
    "Alaska" -> "Juneau",
    // ...
5    "Wyoming" -> "Cheyenne")

```

- filter wird verwendet, um beim Traversieren durch eine Collection bestimmte Elemente herauszufiltern:

```

val stateCapitals = Map(
2    "Alabama" -> "Montgomery",
    "Alaska" -> "Juneau",
    "Wyoming" -> "Cheyenne")

6 val map2 = stateCapitals filter { kv => kv._1 startsWith "A" }

```

- folding u. reducing werden beide verwendet, um eine collection zu schrumpfen, wobei folding immer mit einem *seed* beginnt:

```

1 List (1,2,3,4,5,6) .reduceLeft(_ + _)
List (1,2,3,4,5,6) .foldLeft(10)(_ * _)

```

11.7 Pattern Matching

- ist das für FP, was Kapselung für OOP ist
- Pattern Matching ist ein eleganter Weg, um Objekte so zu zerlegen, dass sie in der erforderliche Verarbeitung passen

11.8 Partielle Funktionen

- wenn an eine Funktion ein Unterstrich übergeben wird, dann handelt es sich um eine partielle Funktionen

- partielle Funktionen sind Ausrücke, in denen nicht alle Argumente einer Funktion auch von der Funktion übernommen werden

```

1 def concatUpper(s1: String, s2: String): String = (s1 + " " + s2).toUpperCase

val c = concatUpper _
println(c("short", "pants"))

5 val c2 = concatUpper("short", _: String)
  println(c2("pants"))

```

11.9 Currying

- stammt vom Mathematiker Haskell Curry u. transformiert eine Funktion, die mehrere Parameter nimmt in eine Kettenfunktion, die nur einen Parameter nimmt
- in Scala werdem curried Funktion mit mehreren Parameterlisten definiert:

```

def multiplrier(i: Int)(factor: Int) = i * factor
val byFive = multiplrier(5) _
val byTen = multiplrier(10) _

4 // byFive underscore drueckt aus, dass das Ergebnis aus dieser Operation eben
  // zwischespeichert wird und dann von byTen verwendet werden kann

scala> byFive(2)
8 res4: Int = 10
scala> byTen(2)
res5: Int = 20

```

- Frage: Wann verwendets man? Wenn man eine spezialisierte Funktion hat, die nur für bestimmte Daten geeignet ist

11.10 Implicits

- es kann manchmal vorkommen, dass man eine Instanz eines Typs hat u. man diesen Typ in einen Kontext einsetzen möchte, in den es einen anderen aber vielleicht ähnlichen Typ geht

⇒ Schlüsselwort `implicit` verwenden

```

1 class FancyString(val str: String)

object FancyString2RichString {
  implicit def fancyString2RichString(fs: FancyString) =
5    new RichString(fs.str)
}

import FancyString2RichString._

```

9

```
val fs = new FancyString("scala")  
println(fs.capitalize.reverse)
```

11.11 Lazy Vals

- das Schlüsselwort *lazy* benutzen:

```
trait AbstractT2 {  
  println("In AbstractT2:")  
  val value: Int  
4  lazy val inverse = { println("initializing inverse:"); 1.0/value }  
  //println("AbstractT2: value = "+value+", inverse = "+inverse)  
}  
  
8 val c2d = new AbstractT2 {  
  println("In c2d:")  
  val value = 10  
}  
  
12 println("Using c2d:")  
println("c2d.value = "+c2d.value+", inverse = "+c2d.inverse)
```

- wenn man eine Variable als *lazy* deklariert, dann sollte man auch alle Verwendungen davon ebenfalls auf *lazy* stellen

12 Scala Typsystem

- Scala ist eine statisch typisierte Sprache u. das Typsystem ist hier am fortschrittlichsten von allen Progsprachen, da es FP u. OOP Benefits vereint

12.1 Parametrisierte Typen

- bla

12.2 Typen Bounds

- `A <: AnyRef` means any type A that is a subtype of AnyRef
- bla, wurde an anderer Stelle besser erklärt

12.3 Nothing und Null

- Null ist `final trait` u. kann nur eine Instanz haben
- Nothing ist `final trait` u. hat keine Instanz

13 Fragen

1. Was sind **traits**?
 - Ist eine besondere Variante eines *mixins*³¹. Ein Trait besitzt eine gemeinsame Basisklasse mit der Klasse, auf die das Trait angewendet wird.
2. Was ist der Unterschied zwischen `val` u. `var`?
 - Variablen können in Scala jeweils den Wert *assign-once* od. *assign-many* haben
 - *assign-once*: mit `val` (ähnlich `final` in Java)
 - *assign-many*: mit `var`
 - ⇒ am besten immer `val` verwenden, denn je weniger sich Dinge Δ können, desto weniger Fehler können sich in den Code schleichen
3. Was drückt der folgende Ausdruck aus: `typeS <: Subject`? Also was macht das `<:?`
4. *subclassing* bei mixins?
5. **case-Klassen** generieren automatisch *factory method* mit denselben Argumenten als Primärkonstruktor
6. Was sind *Views*³²? Was ist im selben Zusammenhang der **implizit** Parameter³³?
7. Was versteht man unter kontravariante Parametertypen?

³¹zusammengehöriges, mehrfach verwendbares Bündel von Funktionalität bezeichnet, das zu einer Klasse hinzugefügt werden kann

³²*implicit conversions between types*; sie werden typischerweise angelegt, um neue Funktionalitäten für einen davor existierenden Typ zu ergänzen

³³Argumente hierfür können bei einem Methodenaufruf weggelassen werden

13.1 Temporäres für DA

13.1.1 Statische versus Dynamisch typisierte Sprachen

- Typing =_{df} *A type system is a tractable syntactic method for preserving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
- ein Typsystem erlaubt es dem System zu sagen, was alles nicht stattfinden soll
- ein Typsystem kann verschiedene Fehler abfangen: *like unsupported operations on particular data structures, attempting to combine data in an undefined way (e.g., trying to add an integer to a string), breaking abstractions, etc.*
- in **statischer Typisierung** ist die Variable für seine gesamte Lebenszeit an einen Typ gebunden u. kann diesen nicht ändern
- in **dynamischer Typisierung** ist der Typ zum Wert u. nicht zur Variable gebunden \Rightarrow *duck typing*; es wird deswegen von dynamik gesprochen, da der Typ einer Variable erst dann ausgewertet wird, wenn es zur Laufzeit verwendet wird
- Scala ist eine statisch typisierte Sprache

13.1.2 Typinferenz

- darunter versteht man, dass man den Typ einer Variable schon anhand des Kontextes bestimmen kann (z.B. $x = 1 + 3 \Rightarrow$ Compiler denkt sich also, dass x Integer ist)

14 Sachen für MT

- **Dienst-orientiertes Komponenten-Model**
 - Scalas Abstraktion kann als Basis für Dienst-orientiertes Komponenten-Model gesehen werden
 - Software-Komponenten sind Berechnungseinheiten, die eine wohlgeformte Menge von Diensten definieren
 - in Scala gehören Software-Komponenten zu Klassen u. *traits*
 - konkrete *Members* einer Klasse od. *traits* stellen die angebotenen Dienste dar, während *derefered Members* als benötigte Dienste angesehen werden können
 - die Komposition von Komponenten basiert auf *mixins*, welche es Programmern erlauben, größere Komponenten aus kleineren zu bauen
 - größte Vorteil gegenüber traditionelle black-box Komponenten ist, dass die Komponenten erweiterbare Entitäten sind \Rightarrow wegen *subclassing* u. *overriding*