Inhaltsverzeichnis

1 Einführung

- **Kompilierung**: `scalac`[1] bzw. `fsc`[2]
- **Ausführung**:[3] `scala`

1.1 Einfache Programme

- Hello Helex:

```
object HelloWorld {

  def main(args: Array[String]) = {
    println("Hello Helex!");
  }
}

// kuerzer durch Verwendung des Mixins
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:

```
import scala.io._

def toInt(in: String): Option[Int] =
  try {
    Some(Integer.parseInt(in.trim))
  } catch {
    case e: NumberFormatException => None
  }

def sum(in: Seq[String]) = {
  val ints = in.flatMap(s => toInt(s))
  ints.foldLeft(0)((a,b) => a + b)
```

[1] Resultat sind JVM Klassen-Datein, welche man in JARs packen kann, hierbei wird jedoch class, trait od. object-Definition verlangt
[2] schnellere Kompilierung
[3] Programm wird kompiliert u. danach gleich ausgeführt

```
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 ists `None`) od. ein Element dann ists `Some(theElement)` enthält
  - durch Option verhindert man `null` Pointer Exceptions
    $\Rightarrow$ gut wenn man Business-Logik schreibt und diesen Fall nicht in jeder Abfrage, sondern einfach am Ergebnistyp der Funktion festlegt[4]
  - 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 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 $\uparrow$, bis alle Elemente der Sequenz durchlaufen wurden

[4] denke an Adminbill aus pictrs
[5] `Seq` ist *supertrait* von `Array, List` u. andere *Collections*
[6] denke Interface aus Java
[7] traits beheben das Diamanten-Prob. der multiplen Vererbung, da eine Klasse beliebig viele traits haben
[8] kann man gut verwenden, wenn man die Werte einer Sequenz aufsummieren will

2 Eigenschaften zur Sprache

- Wofü Scala geeingnet ist:*language ideal for today's scalable, distributed, component-based applications that support concurrency and distribution*
- **Merke**: *Is a statically typed*[9]*, 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 Typdefinitonen und vom Pfad abhängige Typen[10] 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-Framework[12] 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
- alle Kontrollstrukturen von Java sind auch in Scala[13] 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
- Scala-Objekt ist Instanz einer Klasse u. kann deswegen als Parameter von Methoden agieren kann
- Klassen, Objekte u. traits können innere Klassen, Objekte u. traits haben, welche Zugriff auf *private* Methoden, Variablen und so weiter haben

[9] der Typ einer Variable ist für die gesamte Lebenszeit der Variable fest
[10] "νObj calculus"
[11] ermöglichen Komponenten-Adaption in einem modularen Weg
[12] wicket
[13] for-Schleifen wurde stark vereinfacht :)

- 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 Methode[14]
- **call-by name** kann man in Scala durch => für Funktionsaufrufe anfordern:

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

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

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

delayed(nano)

/*
  Delayed method
  Getting nano
  Count: 12
*/

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

[14] man kann aber auch explizit return angeben

⇒ 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 hinzu[15]
- 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 läuft wie Java auf derselben JVM u. sie teilen sich deshalb den gleichen Garbage-Collector
  - auf Java Bytecode kann Scala:
    - Objekte instanziieren
    - Methoden aufrufen
    - exceptions werfen/abfangen
    - Klassen erweitern
    - Interfaces implementieren
  - Java-Klassen als Mixins, wenn diese als Quellcode vorhanden sind
  - Java "locking u. concurency model" wird unterstützt, wird aber normalerweise von Scala gerwapped
- 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
- Generics[16]:

[15] einfach implicit vor Methodendef schreiben

[16] statt Generics sagt man in Scala zu dynamischen Datentypen von Funktionen ***parameterized types***

  - Klassen u. traits können generisch gemacht werden
  - via ***Typparameter*** (Nonvariant, Covariant, Contravariant)
  - obere u. untere Schranken

- FP Eigenschaften: *Higher-order functions*; *Function closure support*; Rekursion als *flow control*; *pure Funktionen* ⇒ keine Seiteneffekte[17]; *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*[18], *singleton types*[19], *explicit self types*[20]
- 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
- 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")
```

[17] viele Datenstrukturen sind immutable, mit `val` sind immutable u. mit `var` sind mutable
[18] kann man festlegen, dass ein Wert eine Instanz von einer Liste von Klassen ist
[19] für Typen gibt es genau einen Wert
[20] sind Annotationen, die den Typ einer aktuellen Instanz einer Klasse festlegen

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

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 primitven 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 erlaubt[21]
- in puren FP kommunizieren Programme durch den Autausch von nebenläufigen autonomen Prozessen $\Rightarrow$ Scala unterstützt dass durch seine `Actors Library`, aber es unterstützt auch veränderliche Elemente, wenn man das will
- Funktionen sind ***first class***[22] u. Scala bietet ***closures***[23]

③ Skalierbarkeit[24] 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)

[21] um Synchronisation muss man sich nicht kümmern
[22] d.h. sie können an Variablen, an andere Funktionen usw. ähnlich wie Werte übergeben werden
[23] bezeichnet 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
[24] es wurde designt, um von kleinen, interpretierten Skripten zu großen, verteilten Anwendungen zu skalieren

2.2 Klassen

- werden in Packeten definiert u. spielen eine ähnliche Rolle wie in Java
- jedes Java-Packet ist auch eine Scala-Pracket (vice versa)
- jede Klasse, ausser der Top-Klasse erbt von genau einer Klass
- jede Klasse kann via *mixin* von mehr als einer Klasse erben
- Scala hat keine Interfaces sondern *traits*, welche zustandslose abstrakte Klassen sind ⇒ um es in Java-Sprache auszudrücken sind *traits* Interfaces mit einer Superklasse, die knicht-abstrakte Methoden beinhalten dürfen
- jede Java-Klasse wird als gewöhnliche Scala-Klasse angesehen u. jedes Java-Interface kann als Scala-*trait* angesehen werden
- n

2.2.1 Klassenhierarchie

- in Scala ist alles, bis auf eine Methode eine Instanz von einer Klasse ⇒ alle Primitven aus Java (wie z.B. int) werden als Instanzen behandelt u. dies wird bei Kompilierung gemacht
- wenn man sich an die Namenskonventionen hält, dann sind die Scala-Repräsentanten der Primitiven Datentypen der JVM `Int, Long, Double, Float, Boolean, Char` u. `Byte` alle Unterklassen von der Klasse `AnyVal`
- Scala hat eine Darstellung on Javas `void`, nämlich `Unit`
- man kann `Unit` explizit zurückgeben, wenn man einfach () hinschreibt
- `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**
- `Unit`-Klasse entspricht dem `void` aus Java

2.2.2 Klassenimport

- Scala-Bibos werden auf die folgende Art u. Weise importiert:

```
import scala.io._
```

- *java.lang.package* importiert! (andere Pakete müssen dann explizit) mit in das System eingebunden werden
- mehrere Klassen od. Objekte können vom selben Paket importiert werden, indem sie einfach in *brackets* geschrieben werden

```
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
```

- wegen Zeile 3 kann man die GETDATEINSTANCE u. LONG direkt nutzen
- NOW erstellt eine Instanz von Javas Datumklasse
- Zeile 9: Methoden mit einen Argument können in der infix-Syntax geschrieben werden u. so ist obiger Ausdruck äquivalent zu DF.FORMAT(NOW)

2.3 Scala hat uniformes Objektmodel

⇒ d.h. jeder Wert ist ein Objekt u. jede Operation ist ein Methodenaufruf

- 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
- Scala behandelt das auftauchen von Bezeichnern zwischen zwei ausdrücken als Methodenaufruf
- Scala erlaubt die Definition von parameterlosen Methoden u. jedesmal wird so eine Funktion aufgerufen, wenn dessen Name verwendet wird
- man kann auch abstrakte Klassenvariablen anlegen, ohne dass man den *modifier* **abstract** davor schreiben muss
- In Scala folgen Konstruktor-Parameter den Klassennamen

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

- Scala braucht den **override** *modifier*, wenn konkrete Methoden einer geerbten Methode überschrieben werden sollen (sollte man eine Methode in einer Subklasse ohne den *override* überschreiben, so meckert der Compiler, sollte man in einer Oberklasse die Parameteranzahl einer Funktion ändern, welche in einer geerbten Klasse noch mit der vorherigen Parameterzahl besteht, so macht der Compiler aus dieser eventuell gewollt überschriebenden Methode einfach ein überladen)

- 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
```

- 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

- die Behandlung von Variablenzugriffen als Methodenaufrufen ermögicht 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
}
```

2.4 Operationen sind Objekte

⇒ kommt daher, dass Scala eine funktionale Sprache ist, d.h. jede Funktion hat einen Wert

Methoden sind funktionale Werte

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

```
def exists[T](xs: Array[T], p: T => boolean) = {
  var i: int = 0
  while (i < xs.length && !p(xs(i))) i = i +1
  i < xs.length
}

def forall[T](xs: Array[T], p: T => boolean) = { # nested functions
  def not_p(x: T) = !p(x)
  !exists(xs, not_p)
}

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[25] 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)
- mithilfe der obigen Funktion können wir eine Funktion `forall` via Doppelnegation 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 $(x : T) =>!p(x)$ ein anonyme Funktion, die alle Parameter vom Typ T nach $!p(x)$

Funktionen sind Objekte

- 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 = ...
  def length: int = ...
```

2.5 Varablendeklarationen

- werde wie Methoden definiert beginne aber mit einen der folgenden *keywords*: `val, var` od. `lazy val`
- mit `var` deklarierte Variablen können im Programmablauf ihren Wert ändern ähnlich wie es auch die Variablen in Java können
- mit `val` deklarierte Variablen werden erst dann ausgewertet, wenn der Block betreten wurde, in der diese Variable definiert wurde
- `lazy val` wird erst dann zugewiesen, wenn die Variable auch benutzt wird

[25] ist eben was generisches

2.6 if/else und while

- if/else wird eher selten verwendet:

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

- if/else verhält sich wie der *ternary*-Operator:

```
val i: Int = if (exp) 1 else 2
```

- das Ergebnis von if u. while ist immer `Unit`
- while-Schleifen sind ebenso effizient wie Rekursion[26]

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

2.7 for-Schleife

- einfache Variante ist wie in Java:

```
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
for {i <- 1 to 5 if isOdd(i)} println(i)
```

- for-Variante zur Umwandlung einer Collection:

```
val lst = (1 to 18 by 3).toList
```

2.8 throw und try/catch/finally

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

```
throw new Exception("Working...")

try {
```

[26] beachte hierbei die *tail-rekursion* bei funktionalen Sprachen

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

- try/finally geht analog
- try/catch ist anders:
  - es gibt immer ein einen Wert zurück
  - es weist einen default Wert zu, sobald alle anderen Tests durchgefallen sind

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

noch ein weiteres Beispiel:

```
import java.util.Calendar
val then = null
val now = Calendar.getInstance()
try {
  now.compareTo(then)
} catch {
  case e: NullPointerException => println("One was null!"); System.exit(-1)
  case unknown => println("Unknown exception " + unknown);
      System.exit(-1)
} finally {
  println("It all worked out.")
  System.exit(0)
}
```

2.9 Kommentare

```
/*
  multiline
*/

// single line

/*
  this is outer comment
  /*
    inner comment
```

```
   */
   a
*/
```

2.10 Enumerations

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

```
object Breed extends Enumeration {
  val doberman = Value("Doberman Pinscher")
  val yorkie = Value("Yorkshire Terrier")
  val scottie = Value("Scottish Terrier")
  val dane = Value("Great Dane")
  val portie = Value("Portuguese Water Dog")
}
// print a list of breeds and their IDs
println("ID\tBreed")
for (breed <- Breed) println(breed.id + "\t" + breed)
// print a list of Terrier breeds
println("\nJust Terriers:")
Breed.filter(_.toString.endsWith("Terrier")).foreach(println)
```

3 Unsortiert

```
object HelloWorld{
  def main(args: Array[String]){
    println("Hello, world!")
  }
}
```

- besteht aus eine main-Methode u. mit args werden jeweils Kommandozeilenparameter als ein Array von Strings entgegengenommen
- main-Methode ist hier eine Prozedur
- **object**-Deklaration enthält eine Main-Methode ⇒ *Singleton object*
- statische Member existieren in Scala nicht
- Scala interagiert mit Java u. es werden alle Klassen von
- Typ Any: ist der Super-Typ von allen anderen Typen u. ist etwas genereller als Javas Objekttyp
- **def** wird zur Festlegung von Funktionen verwendet
- **var** wird zur Festlegung von Variablen verwendet
- **val** definiert nur Werte (*read only*)
- **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 geben in Scala irgendwas zurück, aber der Wert `value ()` wird auf als **unit** bezeichnet
- der Rückgabewert einer Funktion ist per Def. die letzte Anweisung ⇒ es muss kein **return** angegeben werden
- per `scala` kann man in der Konsole einen Interpreter starten
- Klassen sind ähnlich wie Java, nur können Klassen in Scala Argumente haben
- die Argumente einer Klasse kann man sich einfach wie ein Konstruktor vorstellen, welche bei Anlegung einer Instanz immer mit Werten angegeben werden muss

```
class Complex(real: Double, imaginary: Double){
  def re = real
  def im = imaginary
}
```

```
object Bla{
  def main(arg: Array[String]) {
    val test = new Complex(1.5, 2.3)
    println("real      : " + test.re)
    println("imaginary: " + test.im)
  }
}
```

- in diesem Beispiel ist kein return-Wert angegeben u. dies macht für gewöhnlich der Compiler, aber manchmal schimpft er auch herum, sobald der Typ mal nicht zu bestimmen ist

3.1 Alles ist ein Objekt

- alles ist ein Objekt, d.h. sowohl Zahlen als auch Funktionen

Funktionen

- Funktionen sind ebenfalls Objekte u. dies ermöglicht funktionales Programmierung, d.h. Funktionen als Argumente übergeben, Funktionen in Variablen speichern u. Funktion als Rückgabewerte von anderen Funktionen
- Funktionen mit call-back Funktion haben folgende Syntax: () => UNIT und ist eine Funktion, die keine Parameter hat u. kein return-Wert

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

3.2 Anonyme Funktionen

- ist blöd, wenn man Funktionen einen Namen geben muss, wenn man sie nur einmal verwenden
- ausweg sind anonyme Funktionen
- die Anwesenheit von anonymen Funktion wird durch die Syntax: => gemacht

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
```

```
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time to fly anonymous ..."))
  }
}
```

4 Ausdrücke und einfache Funktionen

- Unterschied zwischen `def x = e` u. `val x = e`:
  - `def x = e` - hier wird e nicht ausgewertet, sondern erst, wenn x verwendet wird
  - `val x = e` - hier wird e sofort ausgewertet u. falls man x verwendet, so wird sofort e verwendet, ohne dass der Ausdruck ausgewertet werden muss
  -

- 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

4.1 Methodenaufrufe

- anders als in Java kann man Methoden ohne Parameter auch ohne Klammern aufrufen
- Methoden mit einem Parameter können ebenfalls ohne Klammern aufgerufen werden

```
instance.method()

instance.method

instance.method(param)

instance method param
```

- in Scala können Methoden Symbole wie $+, -, *, and, ?$ enthalten
- Methoden können auch mit dem Typparameter aufgerufen werden:

```
instance.method[TypeParam](p1, p2)
```

4.2 Funktionen

- alle Funktionen haben die `apply`-Methode, welche die Funktion ausführen
- Funktionen können die folgende Form haben: `Function[A, B]`, wobei `A` der Parametertyp u. `B` der Rückgabewert ist
- andere Schreibweise für `Function[A, B]` ist:

  `A => B`
- falls eine Klasse eine `update` Methode. Eine `update` Methode, die zwei Argumente nimmt wird aufgerufen, wenn der Compiler die eine Zuweisung parst

```
class Up {
  def update(k: Int, v: String) = println("Hey: "+k+" "+v)
}
```

- verschachtelte Funktionen:

```
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)
}

println( factorial (10))
```

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

Analog verhält es sich mit Parametern von v

- weitergabe von Parametern von verschachtelten Funktionen:

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

4.3 Parameter

- mit **def** kann man auch Funktionen definieren

```
1 def square(x: Double) = x * x
```

- Funktionsparameter werden immer von Klammern eingeschlossen
- **call-by-value** hat den Vorteil, dass es die wiederholte Auswertung bon Argumenten verhindert
- **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

```
def loop: Int = loop

3 def first(x: Int, y: Int) = x
```

`first(1, loop)` wird bei call-by-name auf 1 gemacht u. wirds hingegen per call-by-value ausgwertet, so erhalten wir ∞-loop

- Scala benutzt per Def. call-by-value, aber kann auf call-by-name wechseln, sofern ein => vorangestellt wird

```
1 def loop: Int = loop

def constOne(x: Int, y: => Int) = 1

5 constOne(1, loop) //ergibt 1
constOne(loop, 1) // ergibt unendliche Schleife
```

4.4 Bedingte Ausdrücke

- if-else wie gehabt
- **true** u. **false** sind ebenfalls da
- !, && || als boolesche Operatoren sind analog wie in Java

4.5 Verschachtelte Funktionen

- in Scala kann man mit *braces* einen Block definieren
- jede Definition in einen Block muss mit einen Semikolon abgeschlossen werden

4.6 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
- n

5 Erste-Klassen Funktionen

- eine Funktion ist in Scala ein "first-class value"
- wie jeder andere Wert in Scala können Funktionen als Parameter übergeben werden od. als das Ergebnis einer Operation
- Funktione, welche andere Funktionen als Parameter nehmen werden *high-order* Funktionen genannt

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

// helper
def id(x: Int): Int = x
def square(x: Int): Int = x * x

//
def sumInt(a: Int, b:Int): Int = sum(id, a, b)
def sumSquare(a: Int, b: Int): Int = sum(square, a, b)
```

der Typ `f: Int => Int` ist so ein Funktionstyp, der für jede beliebige Funktion steht

5.1 Deklaration von Funktionen

- bestehen aus dem Schlüsselwort `def`, einem Methodennamen, Parametern, einen optionalen return-Typ[27], = keyword u. den Methodenrumpf
- folgende Methode nimmt keinen Parameter und gibt einen String zurück:

```
def arsch(): String = "Penner"

def arschGanz() = "Rentner"
```

- Parameter in Funktionen:

```
def foo(a: Int, b: Boolean): String = if (b) a.toString else "false"
```

- Erstellung einer generischen Liste durch den Parameter T:

```
def list[T](p: T): List[T] = p :: Nil
```

- falls man eine Variable Liste an Parametern haben möchte, dann muss * in die Parameterliste einer Funktionsdef setzen:

[27] der Compiler zieht selber Rückschlüsse[28] aus dem return-Type, aber dies nur wirklich machen, wenn man sich sicher ist

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

- man kann Typparameter mit variabler Länge bezüglich der Argumente festlegen:

```
def mkString[T](as: T*): String = as.foldLeft("")(_ + _.toString)
```

- 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)
```

5.2 Anonyme Funktionen

- eine anonyme Funktion ist ein Ausdruck, der wie eine Funktion auswertet, ohne dass man für diese Funktion explizit einen Namen angeben muss
- der Teil vor => sind die Parameter und der Teil danach ist der Rumpf

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

```
def sumSquareAnonym(a: Int, b: Int): Int = sum((x: Int) => x * x, a, b)
```

-

5.3 Currying

- im vorherigen Abschnitt taucht bei der Funktionen-Def. auf, obwohl sie da eigentlich gar nicht benötigt werden
- wir schreiben `sum` nun so um, dass man die Grenzen a u. b nicht mehr angeben muss

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

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

- wir werden nun 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

- der obige Ausdruck ist äquivalent zu

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

- 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)
```

6 Klassen und Objekte

- Klasse für rationale Zahlen:

```
class Rational(n: Int, d: Int) {
  private def gcd(x: Int, y: Int): Int = {
    if (x == 0) y
    else if (x < 0) gcd(−x, y)
    else if (y < 0) −gcd(x, −y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)
  val numer: Int = n/g
  val denom: Int = d/g
  def +(that: Rational) =
    new Rational(numer ∗ that.denom + that.numer ∗ denom,
                 denom ∗ that.denom)
  def −(that: Rational) =
    new Rational(numer ∗ that.denom − that.numer ∗ denom,
                 denom ∗ that.denom)
  def ∗(that: Rational) =
    new Rational(numer ∗ that.numer, denom ∗ that.denom)
  def /(that: Rational) =
    new Rational(numer ∗ that.denom, denom ∗ that.numer)
}
```

- **private members**: solche gekennzeichneten Teile können nicht außerhalb der Klasse angesprochen werden
- **Erstellen u. Zugriff auf Objekte**

```
var i = 1
var x = new Rational(0,1)
x.denom // 1
x.numer // 0
```

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

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
}
```

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

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
```

```
  override def toString = "" + numer + "/" + denom
}
```

- **Parameterlose Funktionen** anders als in Java müssen hier keine Parameter angegeben werden

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
  def square = new Rational(numer*numer, denom*denom)
}
val r = new Rational(3, 4)
println(r.square)
```

- Unterschied einer rechten Seite von value u. parameterlose Funktion

  rechte Seite eines `values` wird ausgewertet, sobald das Objekt angelegt wurde u. der Wert wird danach nicht mehr Δ

  rechte Siete einer parameterlosen Funktion wird jedesmal ausgewertet, wenn die Funktion aufgerufen wird

- **Abstrakte Klassen**:

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

`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
      EmptySet)
}

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

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

- **dynamische Bindung**: betrachte den Ausdruck `s contains 7` u. welche Methode nun ausgeführt wird, hängt, davon ab, von welchen Typ `s` ist (`EmptySet` od. `NonEmptySet`)

- **objects**: statt class kann man auch objects davor schreiben u. dadurch ist das Singleton-Pattern sichergestellt, d.h. dieses Objekt gibt es nur einmal

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

$\Rightarrow$ Objekterzeugung erfolgt nach *lazy evaluation*

- jede Deklaration ohne ein Sichtbarkeits/Scopewort ist per default public[29]

6.1 Konstruktoren

- Scala unterscheidet zwischen ***primary constructor*** (ist der gesamte Klassenrumpf) u. null od. mehr ***auxiliary constructor*** (ist das, was in Klammern hinter den Klassennamen steht)

```
class ButtonWithCallbacks(val label: String,
    val clickedCallbacks: List[() => Unit]) extends Widget {
  require(clickedCallbacks != null, "Callback list can't be null!")

  def this(label: String, clickedCallback: () => Unit) =
    this(label, List(clickedCallback))

  def this(label: String) = {
    this(label, Nil)
    println("Warning: button has no click callbacks!")
  }

  def click() = {
    clickedCallbacks.foreach(f => f())
  }
}
```

[29] für public gibt es kein Schlüsselwort

6.2 verschachtelte Klassen

```
abstract class Widget {
  class Properties {
    import scala.collection.immutable.HashMap
    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) = {
      // Do some preprocessing, e.g., filtering .
      values = values.update(key, value)
      // Do some postprocessing.
    }
  }
  val properties = new Properties
}
```

6.3 Abstraktion

⇒ mächtiges werkzeug für Typen und Werte

- 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)

funktionale Abstraktion

- die folgende Klasse `GenCell` ist generisch

```
class GenCell[T](init: T) {
  private var value: T = init
  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)
}
```

- Anwendung von swap:

```
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 ⇒ im obigen Codeschnipsel zur Anwendung der swap-Methode kann man die Typangaben in den *square brackets* auch weglassen

- **Parameter-Bounds**: man kann einen Obertypen angeben, der als obere Schranke für bestimmte Subtypen agiert (Seite 7 ScalaOverview)
- Fragen: Was sind Typkonstruktoren mit der Eigenschaft ***covariant***?
- **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
- Scalas Typsystem garantiert, dass Varianzannotationen wohl formuliert sind, in dem die Verwendung der jeweiligen Variablen aufgezeichnet wird

6.4 Overriding

- muss man dann hinschreiben, wenn abgeleitete Klassen Methoden, Felderm 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***

6.5 Companion Objekte

- wenn eine Klasse u. ein Objekte innerhalb einer Datei, im selben Packet den gleichen Namen haben, werde diese *Companion Objekte* genannt
- **Apply** Methode: nicht so ganz verstanden

```
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)
}

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

⇒ Apply wird als *factory* Methode verwendet

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

```
class Button(val label: String) extends Widget with Clickable {

  def click() = {
    // Logic to give the appearance of clicking a button ...
  }
  def draw() = {
    // Logic to draw the button on the display, web page, etc.
  }
  override def toString() = "(button: label="+label+", "+super.toString()+")"
}

object Button {
  def unapply(button: Button) = Some(button.label)
}
```

6.6 Komposition

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

- fangen einfach mal mit einem kleinen Beispiel an:

```
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 Interatoren

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


# ein konkreter Iterator, der sukzessive die Zeichen eines Strings returnd
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 ⇒ 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
  }
}
```

- *Mixin-class composition* ist eine Form der Mehrfachvererbung

- **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 Proggern erlauben, größere Komponenten aus kleineren zu bauen
  - größte Vorteil gegenüber traditionelle black-box Komponenten ist, dass die Komponenten erweiterbare Entitäten sind ⇒ wegen *subclassing* u. *overriding*

6.7 Dekomposition

⇒ erlaubt Dekomposition von Objekten durch Pattern-Matching
Objekt-orientierte Dekomposition

- 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
  - durch diese Durchreichung wird es schwierig zu verstehen, was die Methode überhaupt macht u. Δ sind ebenfalls schwierig umusetzen

Pattern Matching über Klassenhierarchie

- in einer funktionalen Sprache 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[30]
- Klassen werden mit **case** getagt:

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term

# dann ist folgendes moeglich
Plus(Plus(Num(1), Num(2)), Num(3))
```

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

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

- der matchende Ausdruck x **match** $\{case\ pat_1 => e_1 case\ pat_2 => e_2 \ldots\}$ matchd den Wert x gegen die Muster $pat_1, pat_2, \ldots$

  $\Rightarrow$ dadurch können neue Funktionen leicht zu einem bestehenden System hinzugefügt werden

[30] without exposing artificial auxiliary functions

6.8 Gleichheit von Objekten

- mit `equal` vergleicht man, ob Objekte den gleichen Werte besitzen
- mit == u. != vergleicht man Wertgleichheit
- n

6.9 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`
- Case-Klassen 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)

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

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

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

# eigene Klasse schreiben, die das gleiche leistet wie die case-Klasse
class Stuff(val name: String, val age: Int) {
  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
    case _ => false
  }
}
```

- Case-Klassen erlauben die Erstellung von *patterns*, welche zu case-class Konstruktoren gehören
- weiteres Bsp.:

```
case class Point(x: Double, y: Double)
abstract class Shape() {
```

```
  def draw(): Unit
}

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

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

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

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),
  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.
println("shape1: "+shape1+". hash = "+shape1.hashCode)
for (shape2 <- shapesList) {
  println("shape2: "+shape2+". 1 == 2 ? "+(shape1 == shape2))
}
```

6.10 Pattern Matching

- ist eine generalsierte `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
}


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

```

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

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

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

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

6.11 traits

- in Scala kann man traits in Objekte erst bei Instanziiirung 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 {
  println( "  in T1: x = " + x )
  val x=1
  println( "  in T1: x = " + x )
}

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

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

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

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

```
  in C12: c = C12
38 After Creating C12
*/
```

⇒ Reihenfolge der Abarbeitung der trais 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 initialsiert werden können ⇒ verwende lieber abstrakte Felder

7 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
}

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:

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

7.1 Annotationen Varianz

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

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

7.2 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)

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)

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 )
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`

7.3 Wann man explizite Typannotationen braucht

- bei einer reinen Variablendeklaration, ohne Wertzuweisung
- Alle Methodenparameter

8 Listen und der Spaß mit der Unveränderlichkeit

- Listen sind ähnlich wie Arrays in anderen Sprachen

```
val fruits = List("apples", "oranges")
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 ⇒ Methoden referenzieren viel weniger auf den externen Zustand von Variablen
  - solche Methoden sind leichter mit automatischen Tests durchzuführen (ScalaCheck)

8.1 Scala Listen, Tupel and Map-Klassen

- Collections (Listen) sind Container für Dinge

```
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
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

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

8.2 List[T]

- List[T] stellen eine verkettete Liste vom Typ T dar, d.h. ist eine sequentielle Liste welche Javas primitve 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
// 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)
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***:

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

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)
// 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

8.3 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

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

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 }

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

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

- `List` hat eine `sort`-Methode:

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

List("b", "a", "elwood", "archer").sort(_ < _)
//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
    def valid: Boolean
}

def validByAge(in: List[Person]) =
  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 _)
// Int = 22

// reduceLeft verwenden, um das laengste Wort zu finden
List("moose", "cow", "A", "Cat").
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

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

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

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

- Erstellung einer geschachtelten Collection:

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

```
n.map(i => n.map(j => i * j))
// 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 wollem 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)
    val n = (1 to 10).toList

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

    // 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)
    ```

    $\Rightarrow$ for-Comprehension ist keine Schleifenkonstrukt sondern einfach nur eine syntaktische Vereinfachung

8.4 Tupel

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

```
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
```

8.5 Map[K,V]

- die `Map` Klasse in Scala ist unveränderlich
- ein `Map` ist eine Sammlung von key/value Paaren
- jeder beliebige value kann von einen 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 Schlüsselzugreifen 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: ...
```

8.6 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`

9 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>

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

// nun vergleichen wir Methoden
f == f

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

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

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

9.1 Partielle Anwendungen und Funktionen

- Methoden u. Funktionen sind in Scala etwas verschiedenes
- in Scala ist alles bis auf Methoden eine Instanz ⇒ Methoden sind keine Funktionen
- Methoden werden an Instanzen angehangen 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)

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 Methodendefs können mit der folgenden Syntax besser beschrieben werden:

```
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

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

9.2 Funktionen und Typparameter

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

```
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

t42(f)
// String = Dude: 42

t42(g)
// Double = 20.0

t42(1 +)
// Int = 43
```

9.3 Funktionen in Container packen

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

```
def bf: Int => Int => Int = i => v => i + v
// (Int) => (Int) => Int

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

10 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 {
  case 0 => 0
  case 1 => 1
  case n => fibonacci(n - 1) + fibonacci(n - 2)
}

// fibo mit guards
def fib2(in: Int): Int = in match {
  case n if n <= 0 => 0
  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")
  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 {
  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
  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:

```
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)
  case _ => None
}
```

- **Pattern-Matching in Listen**:

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


def noPairs[T](in: List[T]): List[T] = in match {
  case Nil => Nil
  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)
    // return a List of the first element followed by noPairs
    // run on the rest of the List
}

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

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

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

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

11 Varianz

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

11.1 Invariante Parametertypen

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

11.2 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)
def get(in: Getable[Any]) {println("It's "+in.data)}
val gs = new Getable("String")
```

11.3 Kovariante Methoden

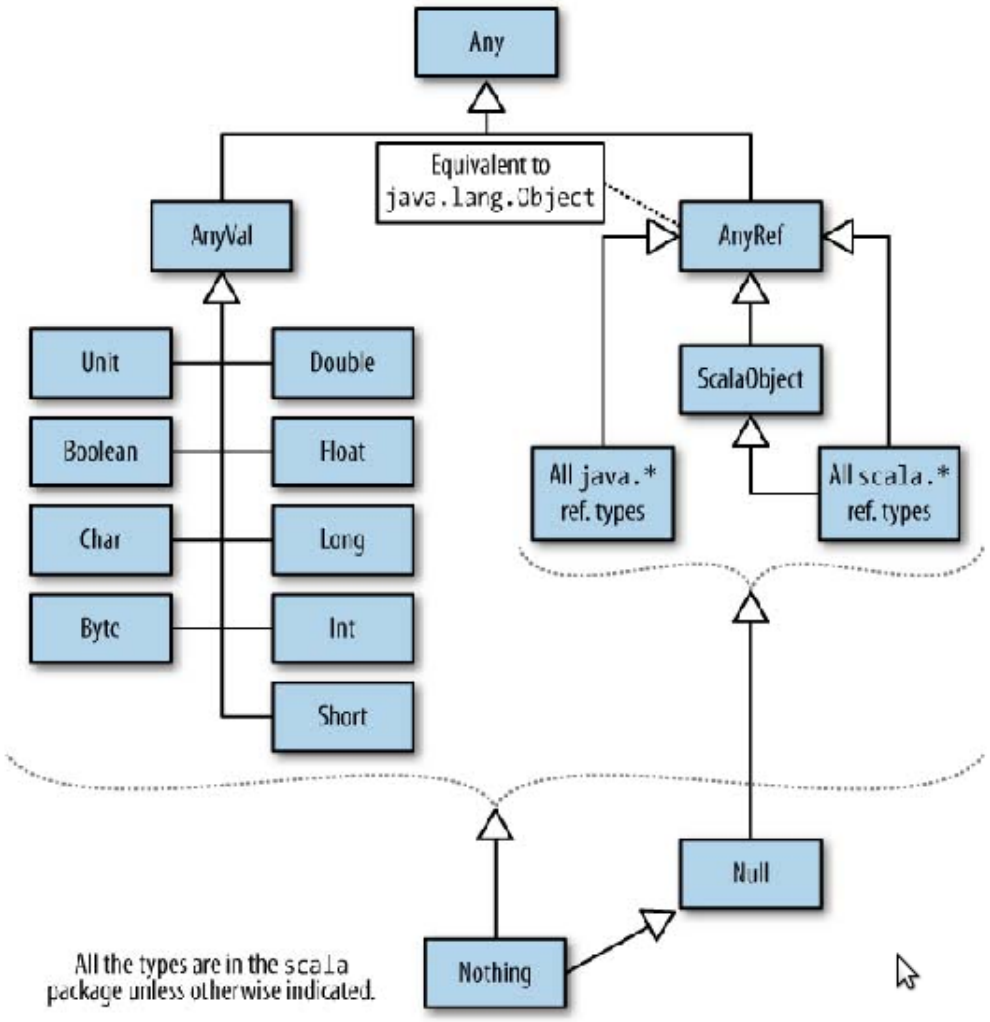
- **kovariant** heißt, dass der return-Wert von Funktionen in abgeleiteten Klassen geändert werden kann:

```
  nabstract 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

11.4 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



12 Scalas Objekt System

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

12.1 Typhierarchie

siehe folgendes Bild

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

12.2 Linearisierung der Objekthierarchie

betrachte folgendes Beispiel:

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

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

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

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

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

val c2 = new C2
println(c2.m)

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


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

// 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.

13 FP in Scala

13.1 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

13.2 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.

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

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

println(l1)
println(l2)

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

$\Rightarrow$ Faktor ist kein formaler Parameter sondern eine Referenz zu einer Variable in einen bstimmten scope, d.h. der Compiler kreiert ein *closure*

13.3 Rekursion

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

13.4 Tail Calls und Tail-Call Optimierung

- Tail Call, d.h. wenn eine Funktion sich erst bei ihren finalen Durchlauf selbst aufruft ⇒ 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 {
  case _ if i == 1 => i
  case _ => i * factorial(i - 1)
}

// 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
    case _ => fact(i - 1, i * accumulator)
  }
  fact(i, 1)
}
```

13.5 Funktionale Datenstrukturen

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

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

// 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)

13.6 Traversieren, Mapping, Filtering, Folding und Reducing

- traveriseren geht mit der `foreach`:

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

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

// The signature of foreach is the following:
trait Iterable[+A] {
  ...
  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

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

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

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

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:

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

13.7 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

13.8 Partielle Funktionen

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

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

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

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

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

13.9 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 multiplier(i: Int)(factor: Int) = i * factor
val byFive = multiplier(5) _
val byTen = multiplier(10) _

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

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

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

13.10 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 `implizit` verwenden

```
class FancyString(val str: String)

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

import FancyString2RichString._
```

```
9
val fs = new FancyString("scala")
println(fs.capitalize.reverse)
```

13.11 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

14 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

14.1 Parametriesierte Typen

- bla

14.2 Typen Bounds

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

14.3 Nothing und Null

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

15 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. *asign-many* haben
   - *assign-once*: mit `val` (ähnlich final in Java)
   - *assign-many*: mit `var`
     $\Rightarrow$ 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 Zusammehang der **implizit** Parameter[33]?
7. Was versteht man unter kontravariante Parametertypen?

[31] zusammengehöriges, 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 davorexistierenden Typ zu ergänzen

[33] Argumente hierfür können bei einem Methodenaufruf weggelassen werden

15.1 Temproräres für DA

15.1.1 Statische versus Dynamsich 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 typung*; 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

15.1.2 Typinferenz

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