Inhaltsverzeichnis

1 Einführung **2**
- 1.1 Einfache Programme . . . 2
- 1.2 Warum Skala . . . 3

2 Eigenschaften zur Sprache **5**
- 2.1 Paradigmen . . . 11
- 2.2 Operationen sind Objekte . . . 12
- 2.3 Varablendeklarationen . . . 13
- 2.4 if/else, while und do-while . . . 14
- 2.5 for-Schleife . . . 14
- 2.6 yield . . . 14
- 2.7 foreach . . . 15
- 2.8 try-catch, throw und finally . . . 15
- 2.9 Enumerations . . . 16
- 2.10 Linearisierung der Objekthierarchie . . . 16

3 Funktionen **18**
- 3.1 Parameter . . . 19
- 3.2 Typparameter . . . 19
- 3.3 Verschachtelte Funktionen und Parameter . . . 19
- 3.4 Wiederholende Parameter . . . 20
- 3.5 First-Class Funktionen . . . 20
- 3.6 Partielle Funktionen . . . 21
- 3.7 Funktionen in Container . . . 22
  - 3.7.1 Callback . . . 22
  - 3.7.2 Anonyme Funktionen . . . 22
- 3.8 Closures . . . 23
- 3.9 Generische Typen und Methoden . . . 23
- 3.10 Schwanzrekursion . . . 24

4 FP in Scala **26**
- 4.1 Was FP ist . . . 26
- 4.2 FP in Scala . . . 26
- 4.3 Tail Calls . . . 26
- 4.4 Funktionale Datenstrukturen . . . 27
- 4.5 Listen . . . 27
  - 4.5.1 List[T] . . . 28
  - 4.5.2 Transformation . . . 29
  - 4.5.3 Tupel . . . 31
  - 4.5.4 Map[K,V] . . . 32
- 4.6 Traversieren, Filtering, Folding und Reducing . . . 32
- 4.7 Currying . . . 33
- 4.8 Lazy Vals . . . 34

5 Klassen und Objekte **35**
- 5.1 Klassen . . . 37
- 5.2 Hierarchie . . . 37

5.3 Import . . . 38
5.4 Felder zu einer Klasse hinzufügen . . . 39
5.5 Self-Referenz . . . 39
5.6 Overriding . . . 39
5.7 Overloading . . . 40
5.8 Abstraktion . . . 40
5.9 Precondition . . . 41
5.10 Companion Objekte . . . 41
5.11 Komposition . . . 42
5.12 Abstrakte Klassen . . . 43
5.13 Dekomposition . . . 43
5.14 Case-Klassen . . . 44
5.15 Pattern Matching . . . 46
5.16 traits . . . 48

6 Varianz **50**
6.1 Varianzregeln . . . 50
6.2 Invariante Parametertypen . . . 50
6.3 Kovariante Parametertypen . . . 50

1 Einführung

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

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))
```

[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

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

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

val input = Source.fromInputStream(System.in)

val lines = input.getLines.collect

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

1.2 Warum Skala

- **Scala ist kompatibel**: Man kann alles von Java verwenden
- **Scala ist knapp und präzise**: weniger Text bedeutet schnelleres lesen, weniger lesen u. mehr Spaß beim proggen; Scalas *type inference* sorgt dafür, dass man manche Typinfos einfach weglassen kann
- **Scala ist *high-level***: Komplexität durch Abstraktion senken

[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

- **Scala ist statically typed**: *classifies variables and expressions according to the kinds of values they hold and compute*

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, d.h. es ist der void Rückgabetyp
- Scala hat eine Darstellung on Javas `void`, nämlich `Unit`[13]
- alle Kontrollstrukturen von Java sind auch in Scala[14] 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

[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] kann man explizit zurückgeben, wenn man einfach () beim Returnwert einer Funktion hinschreibt
[14] for-Schleifen wurde stark vereinfacht :)

```
class Succ(n: Nat){
  def isZero: boolean = false
  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 Methode[16]
- **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)

/*
```

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

[16] man kann aber auch explizit return angeben

```
    Delayed method
    Getting nano
    Count: 12
*/

notDelayed(nano)
/*
    Getting nano
    not delayed method
    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 hinzu[17]

```
implicit def intToRational(x: Int) = new Rational(x)
```

hier wird als `Int` nach `Rational` konvertiert

- 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

[17] einfach implicit vor Methodendef schreiben

- 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[18]:
  - 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* $\Rightarrow$ keine Seiteneffekte[19]; *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*) $\Rightarrow$ 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

[18] statt Generics sagt man in Scala zu dynamischen Datentypen von Funktionen ***parameterized types***
[19] viele Datenstrukturen sind immutable, mit `val` sind immutable u. mit `var` sind mutable
[20] kann man festlegen, dass ein Wert eine Instanz von einer Liste von Klassen ist
[21] für Typen gibt es genau einen Wert
[22] sind Annotationen, die den Typ einer aktuellen Instanz einer Klasse festlegen

- **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")
```

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 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
}
```

- 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:

```
/*
  multiline
*/

// single line

/*
  this is outer comment
  /*
    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
- **Option[T]**
  - ist 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`
- alle `AnyVal` Instanzen sind immutable u. alle `Anyvalue` Typen sind *abstract final*
- Nerdwissen:
  - das sogenannte **Predef Objekt** lädt automatisch wichtige Sachen in ein Scala Programm
  - Objekte werden automatisch und *lazy* zur Laufzeit instanziiert
  - Typahierachieübersicht siehe Odersky-Paper
- **Typem Bounds**
  - `A <: AnyRef` means any type `A` that is a subtype of `AnyRef`
  - bla, wurde an anderer Stelle besser erklärt
- **Nothing und Null**
  - `Null` ist `final trait` u. kann nur eine Instanz haben

  - `Nothing` ist `final trait` u. hat keine Instanz

- ***Views***[23]? Was ist im selben Zusammehang der

- **implizit** Parameter[24]?

- 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

- 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 ausgewertet[25]

- 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

- neue Typen können insofern verwendet werden, indem diese Typen als *buil-ins* eingebunden werden

- die $++$ u. $--$ Ops aus Java gibts es in Scala nicht

- `max` Methode ermittelt von zwei Objekten das Maximum: `3.max(2)`

- jeder Basis-Typ hat einen sogenannten **Rich wrapper**, der zusätzliche Methoden bereitstellt:

  ```
  String hat scala.runtime.RichString
  ```

  $\Rightarrow$ genauso funzen auch die anderen Typen

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*

[23] *implicit conversions between types*; sie werden typischerweise angelegt, um neue Funktionalitäten für einen davorexistierenden Typ zu ergänzen
[24] Argumente hierfür können bei einem Methodenaufruf weggelassen werden
[25] sobald man `x` verwendet, ist der Ausdruck `e` bereits vorhanden

- 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[26]
- in puren FP kommunizieren Programme durch den Autausch von nebenläufigen autonomen Prozessen ⇒ Scala unterstützt dass durch seine `Actors Library`, aber es unterstützt auch veränderliche Elemente, wenn man das will
- Funktionen sind ***first class***[27] u. Scala bietet ***closures***[28]

③. Skalierbarkeit[29] 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) = {
  var i: int = 0
  while (i < xs.length && !p(xs(i))) i = i +1
  i < xs.length
}
```

[26] um Synchronisation muss man sich nicht kümmern
[27] d.h. sie können an Variablen, an andere Funktionen usw. ähnlich wie Werte übergeben werden
[28] bezeichnet man eine Programmfunktion, die beim Aufruf einen Teil ihres vorherigen Aufrufkontexts reproduziert, selbst wenn dieser Kontext außerhalb der Funktion schon nicht mehr existiert ⇒ sind ein mächtiges Werkzeug zur Abstraktion
[29] es wurde designt, um von kleinen, interpretierten Skripten zu großen, verteilten Anwendungen zu skalieren

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

// shorter and eleganter than forAll (works with anonymous function)
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)
  - 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 $!p(x)$

- 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 = ...
```

2.3 Varablendeklarationen

- 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 ⇒ *read only*
lazy val : wird erst dann zugewiesen, wenn die Variable verwendet wird

2.4 if/else, while und do-while

- if/else wird eher selten verwendet u. verhalten sich eher wie der Ternary-Operator; while-Schleifen sind ebenso effizient wie Rekursion[30]

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

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

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

// do-while
do {
  ...
} while (exp != 0)
```

- das Ergebnis von if u. while ist immer `Unit`, sofern man nichts anders angibt

2.5 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)
```

2.6 yield

- wird als Platzhalter für Berechnungen einer Liste genommen, in der die Ergebnisse in der `yield`-Variable abgelegt werden u. anschließend an die Liste zurückgegeben werden, die aufgerufen wurde

```
val noEastCost =
```

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

```
  for (state <- states if state.location != 'east)
    yield state
```

2.7 foreach

- ist die bessere u. eher funktionale Variante

```
var my_list = List("Test1", "Test2", "Test3", 1)
my_list.foreach(elem => println(elem))
// my_list.foreach((elem: Int) => println(elem)) funzt nicht
my_list.foreach((elem: Any) => println(elem)) // funzt

List(1, 2, 3, 4, 5) foreach { i => println("Int: " + i) }


val stateCapitals = Map(
  "Lustria" -> "Montgomery",
  "Great Mountains" -> "Karaz-A-Karak",
  "Imperium" -> "Altdorf")
stateCapitals foreach { kv => println(kv._1 + ": " + kv._2) }
```

2.8 try-catch, throw und finally

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

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

try {
  throw new Exception("Working...")
} finally {
  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

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

- `finally` clause wird immer ausgeführt, unabhängig davon, wie das Programm beendet wird:

```
import java.io.Filereader

val file = new FileReader("bla.txt")
try {
  // use the file
} finally {
  file.close()
}
```

2.9 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 moulder = Value("Master Moulders")
  val pestilens = Value("Biological Weapon")
  val eshin = Value("Assassins")
  val skyre = Value("Gatling Gun and Warpstone")
  val khazad = Value("Great race in Warhammer")
}

for (breed <- Breed) println(breed.id + "\t" + breed)

// print just the molders :)
println("\nJust Moulders:")
Breed.filter(_.toString.endsWith("Moulders")).foreach(println)
```

2.10 Linearisierung der Objekthierarchie

betrachte folgendes Bsp.:

```
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 Linearisierungsalgo, 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.

3 Funktionen

- FKt. 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>
```

- **Funktionen als Parameter übergeben**:

  definieren eine Methode w42, die eine Funktion als Parameter übernimmt:

```
def w42(f: Int => String) = f(42) // f stammt von obigen Bsp.

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
```

- Funktionen können in Scala Symbole wie $+, -, *, and, ?$ enthalten $\Rightarrow$ sind sogenannte Varianzparameter
- Funktionen können die folgende Form haben: `Function[A, B]`, wobei `A` der Parametertyp u. `B` der Rückgabewert ist

  $\Rightarrow$ andere Schreibweise für `Function[A, B]`:

  `A => B`

- Funktionen, 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 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 $=>$:

```
def loop: Int = loop

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

constOne(1, loop)  // 1
constOne(loop, 1)  // infty loop
```

3.2 Typparameter

- Typparameter definieren den Typ von Parametern od. den Rückgabewert der Fkt.:

```
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
```

3.3 Verschachtelte Funktionen und Parameter

- wird auch als *enclosing function bezeichnet*, d.h. durch Schachtelung von Funktionen können Helper-Funktionen auf die Parameter ihrer Elternfunktionen zurückgreifen

-
```
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)
}
```

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

3.4 Wiederholende Parameter

- in die Argumentenliste einer Funktion packt man einfach ein *asterisk*

```
def juhu(args: String*)
  for (arg <- args) println(arg)
```

`String*` ist shortcut für `Array[String]`

3.5 First-Class Funktionen

- d.h. man kann Fkt. in Funktionsliteralen darstellen

```
// ganz normal als einzige Fkt.
(x: Int) => x + 1

// hier als Funktionsliteral
val tmp: Int = (x: Int) => x + 1
```

- man kann durch sogenanntes *taget typing* die Schreibweise von oben verkürzen:

```
someNumbers.filter((x) => x > 0)
```

der Compiler weiss, dass der Filter auf eine Liste von Integern angewendet wird u. deswegen werden die entsprechenden Typen weggelassen u. der Code wird übersichtlicher

- **Placeholder Syntax**: man kann Underscores als Platzhalter für die Variablen verwenden ⇒ dadurch wird die Notation noch kürzer

```
someNumbers.filter(_ > 0)
```

man kann sich den Underscore als Blank vorstellen, der ausgefüllt werden muss

3.6 Partielle Funktionen

- **in Scala ist alles bis auf Methoden eine Instanz**[31]
- Methoden werden an Instanzen angehangen u. angewendet
- in Scala können wir partiell angehauchte Funktionen von Methoden her ableiten:

```
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)
```

durch diese Notation kann man Codeblöcke als Parameter übergeben:

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

- 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

// keep care of the underscore
val c = concatUpper _
println(c("short", "pants"))
```

[31] Methoden sind keine Funktionen in Scala

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

lässt man hinter den Funktionsaufruf ein Leerzeichen u. setzt einen Underscore, so ist dieser der Platzhalter für alle Parameter in der Funktionsliste

⇒ beim Aufruf kann man sich dann ausuchen, wie viele Parameter man der Funktion dann übergeben will

3.7 Funktionen in Container

- Funktionen sind Instanzen u. deswegen kann man alles, was man mit Instanzen machen kann auch mit Funktionen machen
- Array von Funktionen[32]:

```
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>, ...
```

3.7.1 Callback

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

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

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

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

3.7.2 Anonyme Funktionen

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

[32] erinnert frappierend an Wörterbuch der Algorithmen

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

3.8 Closures

- der Name stammt daher, dass die Funktion durch Bindung der freien Variablen an seinen Rumpf entsteht:

```
val more = 1
val addmore = (x: Int) => x + more
```

- eine Funktion ohne freie Variablen wird *closed term* genannt
- was passiert nun, wenn die freien Variablen geändert werden, wenn das Closure schon angelegt werden? Scala erkennt den Kontext u. passt sich an die Änderung der freien Variable an

3.9 Generische Typen und Methoden

- haben einen Stack für Integer

```
abstract class IntStack {
  def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}

class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}

class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

- diesen Stack durch **Typen Parameter** ganz leicht generisch machen

```
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}

class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}

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)
}
```

3.10 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)
}
```

4 FP in Scala

4.1 Was FP ist

- haben keine Seiteneffekte, d.h. Analyse, Testen u. debuggen werden leichter
- Variablen sind in FP immutable
- *referential transparency*[33]

4.2 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!!!!
```

- anonyme Fkt. 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) // List(3, 6, 9, 12, 15)
println(l2) // List(5, 10, 15, 20, 25)
```

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

- **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

4.3 Tail Calls

Tail Call, d.h. wenn eine Funktion sich erst bei ihren finalen Durchlauf selbst aufruft ⇒ Schleifen verhindern die Gefahr eines Stackoverflows:

[33] 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

```
// 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)
}
```

4.4 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 )
```

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

4.5 Listen

- Listen in Scala unterscheiden sich folgendermaßen von anderen Sprachen:
  1. sind unver-Δ u. dessen Elemente müssen immer vom selben Typ sein (was bei Arrays nicht der Fall sein muss)
  2. haben rek. Struktur
  3. haben viel mehr Ops. als Arrays

```
// list of strings
val dwarfs = List("White Dwarf", "Snorri")
val nums = List (1,2,3,4,5,6,7,8,9,10)

// nested list
val div = List(List(1,2), List("String"), 1)
```

- Vorteile Unver-Δ:

- weniger globale Zustände ⇒ weniger Dinge können sich Δ
- Fkt. 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[34] durchzuführen

4.5.1 List[T]

- List[T] ist eine verkettete Liste vom Typ T[35] ⇒ ist sequentielle Liste, welche Javas primitve Datentypen beinhaltet (`Int, Float, Double`), da sich um **boxing**[36] kümmert
- **Listen-Konstruktoren**: `Nil` ist Repräsentant für eine leere Liste, `::` (cons genannt) z.B. `x :: xs` ⇒ Liste mit ersten Element `x` und zweiten Element/Liste `xs`

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

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

⇒ mit `Nil` drücken wir die leere Liste aus

- **Listen sind homogen**[37]:

```
val darkelves:List[Int] = List("Malekith", "Morathi")    // eeek
val darkelves:List[String] = List("Malekith", "Morathi") // jo
```

- **Item an Liste 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 alte Liste x drangehangen[38]

- **Listen mergen**: via `:::`

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

- **typische Listen-Ops**:

[34] `ScalaCheck`
[35] ist also verkettet
[36] also Umwandlung von primitiven Datentypen in Objekten
[37] d.h. innerhalb einer Liste müssen alle Elemente vom selben Typ sein
[38] läuft in O(1)

```
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)
```

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

4.5.2 Transformation

- `map` transformiert alle Elemente einer Collection basierend auf eine Funktion:

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

// kuerzer: List("A", "Cat").map(_.toLowerCase)

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

mit Listen kann man komplexe DB-Queries machen u. Elemente in Liste ausgeben:

```
trait Person {def first : String }

val d = new Person {def first = "German Lord" }
val e = new Person {def first = "Hobbit"}
val a = new Person {def first = "John Grisham"}

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

- `List` hat `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)
```

- `reduceLeft`: Operation auf adjazenten Elemente einer Collection rekursiv durchführen bis alles abgegrast ist:

```
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` ist:

```
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)
```

- geschachtelten Collection erstellen:

```
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 `flatMap`-Methode verwenden

- **for-Comprehension**[39]
  - angenommen haben Liste von Personen mit `namen` u. `age` 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 `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:

[39] for-Comprehension ist kein Schleifenkonstrukt sondern syntaktische Vereinfachung

```
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)
```

4.5.3 Tupel

- sind wie Listen unver-Δ und können aber im Vergleich zu Listen verschiedene Elemente haben
- wollen Fkt. schreiben, die 3 return-Werte hat:

```
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))

// obiges pattern-Matching lesbarer

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
```

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

4.5.4 Map[K,V]

- `Map` Sammlung von key/value Paaren
- `Map` Klasse ist unver-Δ
- jeder beliebige value kann von einen eindeutigen Schlüssel beschrieben werden
- 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):

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

// an eine Map noch ein Element dranhaengen
p + 2 -> "test"

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

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

- mit `-= key` kann man Elemente aus einer Map entfernen
- mit `.contains` kann man testen, ob ein Schlüssel in der Map enthalten ist

4.6 Traversieren, Filtering, Folding und Reducing

- traveriseren via `foreach`
- `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)(_ * _)
```

4.7 Currying

- betrachten `sum`, bei der 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 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

- stammt vom Mathematiker **Haskell Curry** u. transformiert eine Fkt., 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
```

- Verwendung: spezialisierte Fkt. hat, die nur für bestimmte Daten geignet ist

4.8 Lazy Vals

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

```
trait AbstractT2 {
  println("In AbstractT2:")
  val value: Int
  lazy val inverse = { println(" initializing inverse:"); 1.0/value }
}

val c2d = new AbstractT2 {
  println("In c2d:")
  val value = 10
}

println("Using c2d:")
println("c2d.value = "+c2d.value+", inverse = "+c2d.inverse)
```

5 Klassen und Objekte

- Klasse für rationale Zahlen:

```
class Rational(n: Int, d: Int) {
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)
    else if (y < 0) gcd(x, −y)
    else gcd(y%x, x)
  }

  // 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

  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**[40]:

```
var test1 = new RationalNumber(3, 9)
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
}
```

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

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

[40] erfolgt nach *lazy evaluation*

```
  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
}
```

- **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)
}
```

- jede Deklaration ohne ein Sichtbarkeits/Scopewort ist per default *public*[41]
- mit `equal` vergleicht man, ob Objekte den gleichen Werte besitzen
- mit == u. != vergleicht man Wertgleichheit

[41] dafür gibts kein Schlüsselwort

5.1 Klassen

- in Klassen werden *fields* u. *Methoden* angelegt, die als `members` bezeichnet werden
- alle Methodenparameter sind vom Typ `val` ⇒ können nicht ver-Δ werden
- werden in Packeten definiert u. spielen eine ähnliche Rolle wie in Java
- jedes Java-Packet ist auch eine Scala-Packet (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
    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
}
```

- `private`: deklariert man solche Variablen so, so kann man auf diese Variablen nur mit einer Methode der Klasse angesprochen werden, in der diese definiert wurde ⇒ normale Punktnotation geht nicht

```
class CheckSomething{
  private var sum = 0
  def add(b: Byte) {sum += b}
  def checksum {sum}
}
```

5.2 Hierarchie

- 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
- `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

  $\Rightarrow$ einzige Instanz ist die **null** Referenz
- `scala.Nothing` ist *subtypen* von jeden anderen Typen

  $\Rightarrow$ 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`

5.3 Import

- Syntax:

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

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

```
import scala.collection.mutable{Map, Set, Buffer}
```

- können alias für Imports definieren:

```
import scala.collection.mutable{Map => MutableMap}
```

- man kann auch bestimmte Klassen vom import "excluden"

```
import scala.collectiaon.mutable{Map => _, Set => _}
```

5.4 Felder zu einer Klasse hinzufügen

- den Konstruktor-Parameter muss man in einer Klasse noch explizit Werte zuweisen, damit man via der Punktnotation auf diese zugreifen kann

```
class SoUndSo (a: Int, b: Int){
  val aa: Int = a
  val bb: Int = b

  def add(test: SoUndSo): SoUndSo ={
    new SoUndSo(
      aa + that.aa + that.bb,
      bb * that.aa + that.bb
    )
  }
}
```

- mit `that` greift man gerade auf das Objekt `SoUndSo` zu der aufrufenden Methode zu

5.5 Self-Referenz

- das Schlüsselwort `self` zeigt auf eine Objektinstanz der gerade ausgeführten Methode od. wenn es in einem Konstruktor verwendet wird, dann auf die Instanz, die in dem Konstruktor angelegt wird:

```
class SoUndSo (a: Int, b: Int){
  val aa: Int = a
  val bb: Int = b

  def greaterThen(that: SoUndSo) = {
    if (this.aa * this.bb > that.aa * that.bb)
      this.aa
    else
      that.aa
    }
}
```

5.6 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 $\Rightarrow$ potentielle Fehler werden dadurch abgefangen
- Sachen, die als final deklariert sind, kann man nicht ***overriden***

```
class RationalClass(n: Int, d: Int){
```

```
  // ohne dass kommt RationClass@54dh32 heraus
  override def toString = n + "/" + d
}
```

5.7 Overloading

- gleiche Fkt.namen mit verschiedenen Signaturen lösen verschiedene Nachrichten aus

```
class SoUndSo (a: Int, b: Int){
  val aa: Int = a
  val bb: Int = b

  def +(that: SoUndSo) = {that.aa + that.bb}
  def +(string: String) = {string}
}
```

5.8 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
  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

- **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

5.9 Precondition

- kann man bei Konstruktoren nehmen, damit bestimmte Werte von vornherein ausgeschlossen werden:

```
class RationalClass(n: Int, d: Int){
  require(d != 0)
}
```

5.10 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

```
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")
```

- **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()+")"
}
```

```
11
object Button {
  def unapply(button: Button) = Some(button.label)
}
```

5.11 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 Interatoren

```
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 ⇒ 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
  }
```

```
}
```

5.12 Abstrakte Klassen

- von solchen Klassen können keine Instanzen erzeugt werden:

```
abstract class Element {
  def contents: Array[String]
}
```

- Methoden einer abstrakten Klasse brauchen keinen Extramodifier, der das anzeigt, da die Klasse diese Aufgabe bereits übernimmt

5.13 Dekomposition

von Objekten via Pattern-Matching

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

```
abstract class Term {
  def eval: int
}

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

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 ⇒ 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":

```
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

5.14 Case-Klassen

- es wird einfach das Schlüsselwort **case** vor Klassen bzw. Objekten geschrieben
- **case-Klassen** generieren automatisch *factory method* mit denselben Argumenten als Primärkonstruktor
- 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)

# 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:

```
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))
}
```

5.15 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"
}
```

- **Fibonaccizahlen** verschiedenen Varianten:

```
// normale Fibos
def fibonacci(input: Int): Int = input 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 "Dwarf" | "Dawi" => Some("stubborn")
  case "Elves" => Some("intriguer")
  case "Skaven" | "Skavenblight" => Some("back−stabbing")
  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 zu dieser Klasse:

```
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
}
```

- **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
```

```
}
```

- **Pattern-Matching 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)
```

5.16 traits

- ist besondere Variante eines ***mixins***[42]
- trait besitzt eine gemeinsame Basisklasse mit der Klasse, auf die das Trait angewendet wird
- in Scala können traits in Objekten erst bei Instanziierung einbinden
- traits unterstützen nicht beliebige Konstruktoren u. nehmen auch keine Argumente für ihre Konstruktoren auf

  $\Rightarrow$ können keine Argumente an ihre Elternklassen weitergeben

```
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 )
```

[42] zusammengehöriges, mehrfach verwendbares Bündel von Funktionalität bezeichnet, das zu einer Klasse hinzugefügt werden kann

```
}

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
After Creating C12
*/
```

⇒ Reihenfolge der Abarbeitung der trais ist von links nach rechts

- traits können auch als abstrakt deklariert haben
- **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

6 Varianz

Regeln festlegen, nach denen parametrisierte Typen als Parameter übergeben werden können

6.1 Varianzregeln

- Δ Container sollten invariant sein
- unver-Δ Container sollten kovariant sein
- die Inputs von Transformationen sollten kontravariant sein u. die Outputs von Transformationen sollten kovariant sein

6.2 Invariante Parametertypen

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

6.3 Kovariante Parametertypen

- Kennzeichen: + vorm Typparameter, z.B. `List[+T]`
- Anwendung: wenn wir mit *read-only* container umgehen

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