innFactory

Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025

innFactory

Strukturen, Vererbung und Typsicherheit (in Scala)

Strukturen, Vererbung und Typsicherheit
class vs case class

```scala
class RegularPerson(val name: String, val age: Int)
val regularPerson = new RegularPerson("test", 1)
```

```scala
// Automatisches "val"
case class Person(name: String, age: Int)
// Kein new notwendig
val person = Person("Max Mensch", 20)
// "kostenlose" copy Methode
val olderPerson = person.copy(age = 21)
```

```scala
// Reference equality vs value equality
val rp2 = new RegularPerson("test", 1)
regularPerson == rp2 // false

val p2 = Person("Max Mensch", 20)
person == p2 // true
```

Strukturen, Vererbung und Typsicherheit
Enum

Scala 2

```
sealed abstract class AnimalType
object AnimalType {
  case object Mammal extends AnimalType
  case object Bird extends AnimalType
  case object Reptile extends AnimalType
  case object Amphibian extends AnimalType
  case object Fish extends AnimalType
  case object Invertebrates extends AnimalType
}
```

Scala 3

```
enum AnimalType {
  case Mammal, Bird, Reptile
  case Amphibian, Fish, Invertebrates
}
```

Strukturen, Vererbung und Typsicherheit
Abstract class

```
abstract class Animal(val type: AnimalType) {
    def eat(): Unit
}

class Dog extends Animal(AnimalType.Mammal) {
    override def eat(): Unit = ???
}
```

Strukturen, Vererbung und Typsicherheit
traits (Eigenschaften)

```scala
abstract class Animal(val type: AnimalType) {
    def eat(): Unit
}

class Dog extends Animal(AnimalType.Mammal) {
    override def eat(): Unit = ???
}
```

```scala
// Seit Scala 3 auch Konstruktorparameter wie bei abstrakten Klassen möglich
trait Carnivore(val dangerousToHumans: Boolean) {
    // Einschränkung des Traits auf Verwendung bei Subtypen der Klasse Animal
    self: Animal ⇒
    def eat(animal: Animal): Unit
}

class Wolf extends Dog with Carnivore(true) {
    override def eat(animal: Animal): Unit = ???
}
```

Strukturen, Vererbung und Typsicherheit
Unterschiede abstract class und trait

abstract class

- Klassen können nur von **einer** abstrakten Klasse mit **extends** erben
- Wird für gemeinsame Basisklassen mit Zustand verwendet

trait

- Eine Klasse kann mit **extends** und **with** von mehreren traits erben
- Wird für **Verhalten oder Eigenschaften** verwendet
- Seit Scala 3 auch Konstruktorparameter möglich

Strukturen, Vererbung und Typsicherheit
Modifiers sealed and final

sealed informiert den Compiler darüber welche Subklassen existieren

- Relevant für Pattern Matching
- Definition von Subklassen nur in derselben Datei möglich

```
sealed trait MultipleChoice

class OptionA extends MultipleChoice
class OptionB extends MultipleChoice
class OptionC extends MultipleChoice
```

final verhindert Vererbung von Klassen oder das Überschreiben von Methoden

```
final class DoNotExtendMe

// Error: Illegal inheritance from final class
class Extender extends DoNotExtendMe

class DoNotExtendMyMethod {
  final def test: String = "unchangable"
}

class MethodExtender extends DoNotExtendMyMethod {
  // error overriding method test in class DoNotExtendMyMethod
  override def test: String = "change"
}
```

Strukturen, Vererbung und Typsicherheit
Modifier private und protected

```
package de.innfactory.afps

class OtherModifiers {
  // Verfügbar innerhalb der Klasse
  private val t1: Int = 42
  // Verfügbar innerhalb der Klasse und lesbar/überschreibbar in Sub-Klassen
  protected val t2: Int = 42
  // Verfügbar innerhalb des package "t
  private[test] val t3: Int = 42
}
```

```
class AccessTest extends OtherModifiers {
  println(t1) // Not found: t1
  override val t1 = ??? // value t1 overrides nothing
  // Erlaubt
  override protected val t2: Int = 43
  println(t3)
}
```

Strukturen, Vererbung und Typsicherheit
Generics

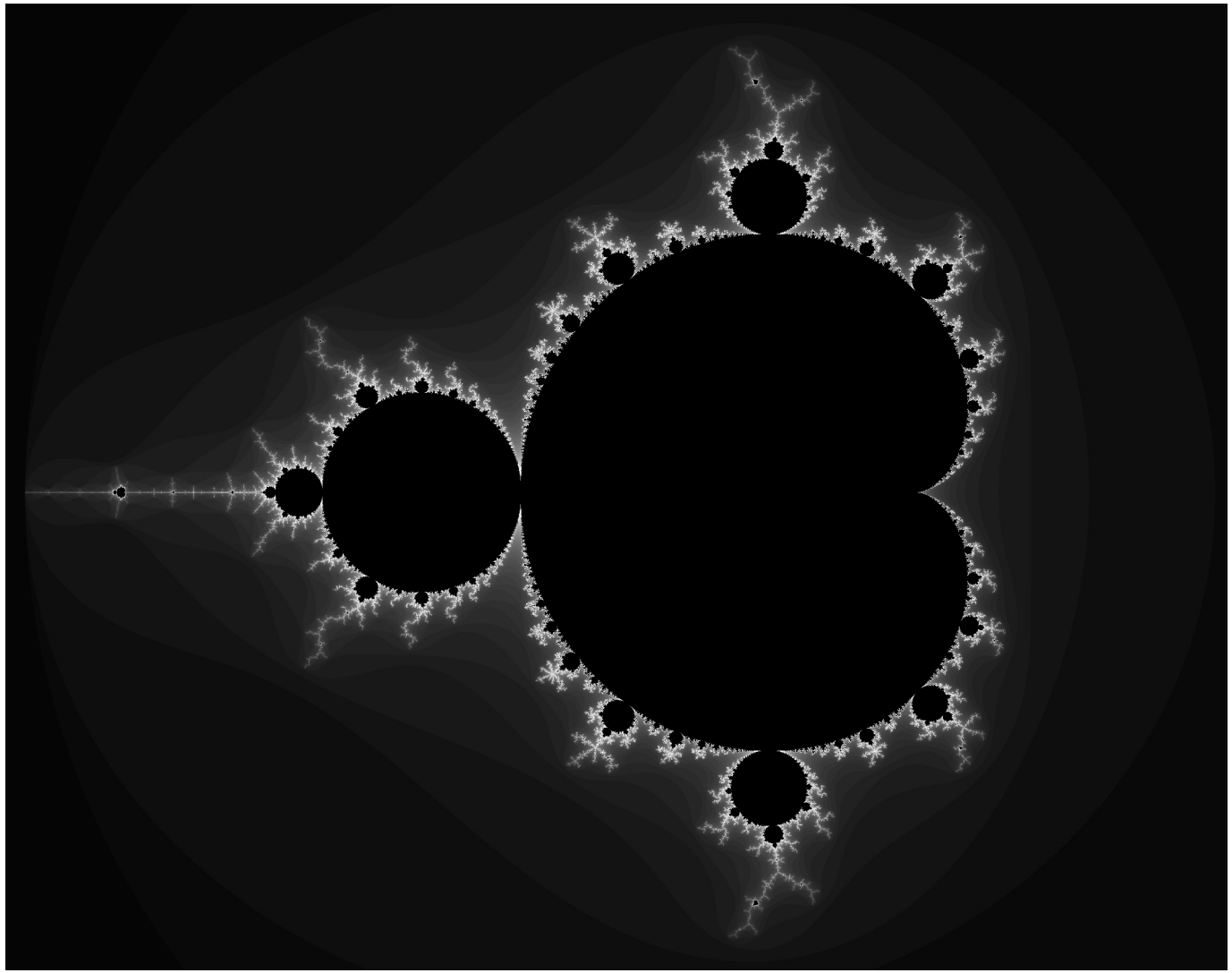
```
(abstract) class|trait Box[A](value: A) {
  def get: A = value
  def map[B](f: A → B): Box[B] = Box(f(value))
}
// Beispiel
val box = Box(42) // Typinferenz Box[Int]
val strBox = box.map(_.toString)
// → Box[String]("42")
```

innFactory

Funktionales Denken mit Scala: Rekursion, Higher-Order Functions & Pattern Matching

Rekursion & Endrekursion
Einführung

Rekursion liegt dann vor, wenn eine Funktion, ein Algorithmus, eine Datenstruktur, ein Begriff, etc. durch sich selbst definiert wird



https://mandelbrot.silversky.dev/

Rekursion & Endrekursion
Überblick

- **Rekursion:** Eine Funktion ruft sich selbst auf, um Teilprobleme zu lösen. Wird genutzt, um Schleifen zu ersetzen
- **Basisfall & Rekursiver Fall:** Rekursive Funktionen brauchen einen Abbruchfall (Basis), sonst laufen sie unendlich. Der rekursive Fall bricht das Problem auf und ruft die Funktion mit kleineren Parametern erneut auf
- **Tail Recursion (Endrekursion):** Spezialfall, bei dem der rekursive Aufruf die *letzte Aktion* der Funktion ist.

Rekursion & Endrekursion
Rekursionstiefe und Laufzeit-Stack

- Zur Laufzeit wird bei jedem Funktionsaufruf ein **Call-Frame** bestehend aus
  - Parameter,
  - Rücksprungadresse und
  - lokale Variablen

  in den Laufzeit-Stack abgelegt
- Zu große Rekursionstiefe => Überlauf des Laufzeit-Stack (Stack Overflow Exception)
- **Lösung in Scala: Endrekursion**
  - Compiler optimiert endrekursive Funktionen zu einer Schleife
  - @tailrec Annotation prüft auf Endrekursion

Rekursion & Endrekursion
Rekursion vs Iteration

- Rekursion kann Schleifen (**Iteration**) ersetzen. Jede **for**- oder **while**-Schleife lässt sich durch eine rekursive Funktion ausdrücken
- **Imperativ (Iteration):** Zustand wird in jeder Schleifeniteration geändert (z.B. Loop-Index, Akkumulator-Variable)
- **Funktional (Rekursion):** Zustand wird nicht mutiert, stattdessen wird pro Rekursionsaufruf ein neuer „Zustand“ als Parameter übergeben

Rekursion & Endrekursion
Beispiel Rekursive Funktion: Fakultät

- Mathematik
  - **Iterativ** $n! \coloneqq \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdots n$
  - **Rekursiv** $n! = \begin{cases} 1, & n=0 \\ n \cdot (n-1), & n>0 \end{cases}$

innFactory

Code Beispiel Iterativ vs Rekursiv: factorial

(+ stringConcatenation tailrec)

https://github.com/innFactory-Classrooms/afps/blob/main/vl02/proj-vorlesung/src/main/scala/de/innfactory/afps/IterativeVsRecursive.scala

Rekursion & Endrekursion
Probleme rekursiv lösen

Problemstellung

- Gesucht ist eine rekursive Funktion zur Lösung eines Problems P der Größe n $(n \geq 0)$
- Beispiele: $\mathrm{fak(n)}$, Suche x in n Zahlen oder Sortiere n Zahlen

Vorgehensweise

- **Rekursionsfall**: Reduziere Problem der Größe n auf ein Problem der Größe k mit $0 \leq k < n$ (oder evtl. mehrere Probleme).
  Beispiel: bei der Fakultätsfunktion wird $\mathrm{fak(n)}$ zurückgeführt auf $\mathrm{n} * \mathrm{fak(n-1)}$
- **Basisfall** (bzw. Basisfälle): Löse P für alle Werte n direkt, die sich im Rekursionsfall nicht weiter reduzieren lassen.
  Beispiel: bei der Fakultätsfunktion ist der Basisfall $\mathrm{fak(0)}$

Rekursion & Endrekursion
Teile und Herrsche (Divide-and-Conquer)

Prinzip

- **Zerlege** ein Gesamtproblem in kleinere Teilprobleme
- **Löse** diese Teilprobleme rekursiv
- **Kombiniere** die Teilergebnisse zur Lösung des Gesamtproblems

Beispiele

- Mergesort, Quicksort
- Binary Search
- Matrix- oder Bildverarbeitung

Rekursion & Endrekursion
Beispiel Binary Search (Divide-and-Conquer)

- Suche in sortierter Liste (Existiert Element in einer Liste)
  - Vergleiche mittleres Element => **Divide**
  - Suche weiter in linker oder rechter Hälfte => **Conquer**

```
def binarySearch(lst: VectorlInt], target: Int, low: Int, high: Int): Boolean = {
  if (low > high) false
  else {
    val mid = (low + high) / 2
    if (lst (mid) = target) true
    else if (lst(mid) › target)
      binarySearch(lst, target, low, mid - 1)
    else
      binarySearch(lst, target, mid + 1, high)
  }
}
```

Rekursion & Endrekursion
Vorteile von Divide-and-Conquer

- **Klare Struktur:** Aufteilung => Lösung => Kombination
- **Elegant mit Rekursion:** Jedes Teilproblem wird unabhängig berechnet
- **Testbarkeit:** Teilfunktionen lassen sich separat testen
- **Einfach zu parallelisieren:** Teillösungen können unabhängig voneinander berechnet werden

innFactory

Higher-Order Functions

Higher-Order Functions
Konzept

- **Funktionen als Werte:** In Scala können Funktion in Variablen gespeichert, als Parameter mitgegeben oder als Rückgabewert zurückgegeben werden
- Eine **Funktion höherer Ordnung** ist eine Funktion, die mindestens eine Funktion als Argument nimmt **oder** eine Funktion zurückgibt
- **Vorteil:** Wiederkehrende Abläufe lassen sich in generische Funktionen auslagern
  => Weniger duplizierter Code, klarere Struktur

Higher-Order Functions
Lambdas in Scala

- Lambdas sind **anonyme Funktionen**, d.h. ohne Namen
- Werden oft als **Argument** an Higher-Order Functions übergeben
- Schreibweisen

```
// Normal
val f = (x: Int) ⇒ x * 2
// Abgekürzt
List(1).map(x ⇒ x * 2) // (Nur möglich wenn der Typ bekannt ist)
// Platzhalter
List(1).map(_ * 2)
// Pattern Matching mit case
((Int, Int)) ⇒ Int = { case (a, b) ⇒ a + b }
```

Higher-Order Functions
Beispiele in Collections

```scala
val zahlen = List(1, 2, 3, 4)

// map: Transformation
zahlen.map(_ * 2) // List(2, 4, 6, 8)

// filter: Selektion
zahlen.filter(_ % 2 == 0) // List(2, 4)

// foldLeft: Aggregation
zahlen.foldLeft(0)(_ + _) // 15

// flatMap: Transformation + Flattening
zahlen.flatMap(n ⇒ List(n * 2, n * 3) // List(2, 3, 4, 6, 6, 9, 8, 12)
```

Weitere Funktionen:
https://docs.scala-lang.org/scala3/book/collections-methods.html
und
https://superruzafa.github.io/visual-scala-reference/

Higher-Order Functions
Currying

- **Currying** wandelt eine Funktion mit mehreren Parametern um in eine **Kette von Funktionen,** welche jeweils nur ein Argument haben
  => Funktionen werden **teilanwendbar**

```
def pow(base: Int)(exp: Int): Int = Math.pow(base, exp).toInt

// Spezialisiert
val square: Int ⇒ Int = pow(_)(2)
val cube: Int ⇒ Int = pow(_)(3)

// Verwendung
val nine = square(3)    // 9
val twentySeven = cube(3) // 27
```

Higher-Order Functions
Vorteile

- **Weniger Boilerplate:** Wiederkehrende Muster (z.B. über eine Liste iterieren) sind in Bibliotheksfunktionen gekapselt. Der eigene Code bleibt kurz und fokussiert auf die eigentliche Logik
- **Weniger Code-Duplizierung:** Anstatt ähnliche Schleifen immer wieder zu schreiben, nutzt man allgemeine Funktionen (map, filter etc.) und gibt nur das spezifische Verhalten als Parameter
- **Bessere Lesbarkeit:** Der Code sagt was getan wird, nicht wie. Z.B. liste.filter(isValid).map(toDto) ist wie eine Satzbeschreibung – leichter zu verstehen als verschachtelte Schleifen
- **Wartbarkeit:** Weniger Fehlerquellen (keine Indexfehler, keine vergessenen break/continue wie in Schleifen). Außerdem können Bibliotheksfunktionen intern optimiert sein.

innFactory

Pattern Matching

Pattern Matching
Überblick

- Pattern Matching ermöglicht elegante Fallunterscheidungen basierend auf dem *Inhalt* von Werten
- Ähnelt auf den ersten Blick einem switch-case, ist aber deutlich mächtiger und sicherer
- Idee: Ein Wert wird gegen verschiedene **Muster** geprüft. Beim ersten passenden Muster wird der zugehörige Code ausgeführt (bzw. Wert zurückgegeben)

Pattern Matching
Idee

- **Vergleich mit switch:** Statt vieler „if-else“ oder eines eingeschränkten „switch“ bietet Scala mit „match { case ... }“ eine ausdrucksstarke Alternative
- **Muster statt Werte:** Man kann nicht nur auf Gleichheit prüfen (Wert X?), sondern auch Muster wie Zahlenbereiche, Typen oder Strukturen (z.B. Liste leer/nicht leer) angeben
- **Immer ein Ergebnis:** Pattern Matching ist ein *Ausdruck* – es liefert einen Wert zurück. (Kein break nötig wie in switch, keine fall-through-Probleme.)
- **Wildcard:** Ein Muster _ dient als Auffang für "alles andere" (ähnlich default im switch). Damit kann man sicherstellen, dass alle Fälle abgedeckt sind. (Scala warnt, wenn nicht alle Möglichkeiten abgedeckt sind – erhöhte Sicherheit bei sealed classes.)

Code Beispiele Pattern Matching

https://github.com/innFactory-Classrooms/afps/blob/main/vl02/proj-vorlesung/src/main/scala/de/innfactory/afps/PatternMatching.scala

Pattern Matching
Vergleich mit Java

- **Kürzer & sicherer:** Pattern Matching ersetzt oft lange if-else-Ketten oder umständliche instanceof-Prüfungen in Java. Weniger Code, klarere Struktur.
- **Keine Fallthrough-Probleme:** Jeder case-Zweig ist in Scala automatisch abgeschlossen. Kein break notwendig, und es gibt kein ungewolltes “weiterfallen” in den nächsten Fall.
- **Exhaustiveness:** Scala kann (bei geschlossenen Typen wie sealed Klassen) prüfen, ob alle Fälle abgedeckt sind – erhöht die Sicherheit.
- **Java holt auf:** Neuere Java-Versionen (Java 16+/17) haben rudimentäres Pattern Matching eingeführt (z.B. instanceof mit Pattern-Binding, switch auf einzelne Typen), aber es ist längst nicht so umfassend wie in Scala.

innFactory

Übung

Übung
Eigene Liste in Scala implementieren

Zu verwendende Konzepte

- sealed abstract class
- @tailrec
- Pattern Matching

Tips

- Generic für Liste [+T]
- FLEmpty als object mit Generic [Nothing]
- Beispiel FLNonEmpty(1, FLNonEmpty(2, FLNonEmpty(3, FlEmpty)))
  => 1, 2, 3
- Durch funktionales erstellen der Liste ist der Inhalt “rückwärts”
  => Hilfsmethode *reverse* sinnvoll