innFactory

Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025

innFactory

Fehlerbehandlung

Fehlerbehandlung
Fehler sind unvermeidbar (Anything that can go wrong will go wrong)

Programme stoßen oft auf Situationen, die fehlschlagen können

- Fehlerhafte Nutzereingabe
- Datenbankoperationen (Constraint Violation)
- Zugriff auf ein Element in einer Liste, das nicht existiert
- Asynchrone Operationen, die fehlschlagen oder abbrechen

Konsequenz

- Fehlerfälle sind keine Ausnahmen, sondern der Normalfall
- Gute Programme behandeln Fehler explizit und robust

Fehlerbehandlung
Probleme klassischer Fehlerbehandlung

- Übliche Praxis (Beispiel Java): **Exceptions werfen und fangen**
- Probleme mit Exceptions
  - Nicht direkt sichtbar, welche Methoden Exceptions werfen bzw. welche
  - Unübersichtliche Fehlerbehandlung mit try-catch auf verschiedenen Ebenen welche jeweils spezifische Exceptions fangen und andere weiter „fallen“ lassen
  - Fehler sind erst bei Laufzeit erkennbar

  => Unerwartete Programmabstürze
- Wir brauchen bessere Alternativen für robuste Software!

innFactory

Option[T]

Option[T]
Überblick

- **Option[T]** repräsentiert einen optionalen Wert vom Typ **T** (Alternative zu **null**)
  - **Some[T]:** Enthält einen Wert
  - **None:** „Kein Wert vorhanden“
- Typische Verwendung: Rückgabewerte von Funktionen, die keinen gültigen Wert garantieren können
  => Beispiel Listen Index Zugriff, verhindert **NullPointerException**

Option[T]
Wichtige Methoden

- **filter(p: T => Boolean):** Behält den Wert, falls Prädikat p erfüllt ist, sonst wird aus **Some** ein **None**. (None bleibt None)
- **getOrElse(default: T):** Liefert den Wert bei **Some**, sonst den angegebenen Default-Wert
- **orElse(alternative: Option[T]):** Liefert die Alternative, falls **None** (kann genutzt werden, um eine zweite Option zu versuchen)
- **isDefined / isEmpty:** Prüft, ob ein Wert vorhanden ist (true bei Some, false bei None bzw. umgekehrt)
- Hinweis: Die Methode **get** gibt den Wert zurück oder wirft eine Exception bei None – in idiomatischem Scala wird stattdessen getOrElse oder Pattern-Matching verwendet

innFactory

Option[T]
Pattern Matching Beispiel

```
val someList: List[Int] = List(1, 2)
val maybeElement: Option[Int] = someList.lift(5)

maybeElement match {
  case Some(value) ⇒ println(s"At index 5 we have $value")
  case None ⇒ println(s"Index 5 contains a whole lot of nothing")
}
```

Option[T]
map und flatMap

- **map(f: T => U):** Wandelt einen vorhandenen Wert **T** mit ***f*** in einen neuen Wert **U** um (Vergleichbar mit map bei Listen)
- **flatMap(f: T => Option[U])**: Ähnlich wie map, aber ***f*** liefert selbst ein Option. Gibt direkt dieses Ergebnis zurück (verhindert ein geschachteltes Option[Option[U]])

=> Beide Methoden ermöglichen das Verketten mehrerer Operationen auf dem Wert, ohne explizite If-Expression bzw. Pattern-Matching um None abzudecken

Option[T]
Problem bei mehreren map / flatMap nacheinander

Beispiel: Drei optionale Zahlen addieren

```
val sumOpt: Option[Int] = optA.flatMap(a ⇒
  optB.flatMap(b ⇒
    optC.map(c ⇒ a + b + c)
  )
)
```

Mehrere hintereinandergeschaltete flatMap-Aufrufe führen zu verschachteltem Code (Lambdas in Lambdas)

=> Unübersichtlich

Option[T]
Lösung: For-Comprehensions (for-yield)

```
val sumOpt: Option[Int] = for {
  a ← optA
  b ← optB
  c ← optC
} yield a + b + c
```

- Scala bietet mit For-Comprehensions eine Syntax, um mehrere Option-abhängige Schritte sequenziell abzuarbeiten
- Die Notation „for { ... } yield …“ ist syntactic sugar für mehrere flatMap-/map-Aufrufe
- **Vorteil**: Der Code liest sich linear und das Behandeln von **None** passiert automatisch
- Wenn eines der **Options** **None** ist, wird der gesamte Ausdruck zu **None** (vergleichbar mit frühzeitigem return)

For-Comprehensions

For-Comprehensions
Einführung & Syntax

- Eine for-Comprehension erlaubt das elegante Kombinieren mehrerer Berechnungen über *Monaden* (z.B. Option, Either, List, Future)
- Syntax: „for (x <- mx; y <- my; if Bedingung; z <- mz) yield { Ausdruck }“ – in geschweiften Klammern oder Zeilen für bessere Lesbarkeit
- **Wichtig:** Dies ist keine gewöhnliche Schleife, sondern konstruiert aus gegebenen monadischen Werten einen neuen Resultat-Wert.
- Jeder „<-“ Ausdruck entnimmt einen Wert aus dem Kontext (z.B. dem Option), führt den Codeblock rechts davon aus und gibt ihn in den yield-Ausdruck
- Durch **if** innerhalb der For-Comprehension kann man zusätzliche Bedingungen einbauen

innFactory

For-Comprehensions
Übersetzung in map/flatMap

- Der Compiler übersetzt eine For-Comprehension in Ketten von **map**, **flatMap** und evtl. **withFilter** Aufrufen

```
for { x ← optX; y ← optY } yield (x + y) // wird zu
optX.flatMap(x ⇒ optY.map(y ⇒ x + y))
```

- Mit einer Bedingung

```
for { x ← optX; if x > 0; y ← optY } yield f(x,y) // wird zu
optX.filter(x > 0).flatMap(x ⇒ optY.map(y ⇒ f(x,y)))
```

- Jeder Generator (Bindung mit <-) außer dem letzten wird zu einem flatMap; der letzte wird zu map (weil yield mit dem letzten kombiniert wird)

Alle eingebundenen Ausdrücke müssen von kompatiblem Monaden-Typ sein. Man kann z.B. nicht direkt ein **Option[T]** und eine **List[T]** in derselben For-Comprehension mischen

innFactory

Either[L, R]

Either
Einführung: Either für Entweder-Oder Ergebnisse

- **Either[L, R]** repräsentiert einen Wert, der zwei Ausprägungen haben kann: **Left[L]** oder **Right[R]**
- Konvention in Scala
  - **Left** wird oft für Fehler oder das Fehlschlag-Ergebnis genutzt
  - **Right** für den Erfolgsfall

  => z.B. Either[String, Int] mit Left = Fehlermeldung, Right = gültiges Ergebnis
- Vorteil gegenüber Option: Man kann im Fehlerfall zusätzliche Informationen (z.B. eine Fehlermeldung oder Error-Objekt) zurückgeben, statt nur None
- **Either** ermöglicht typsichere Fehlerbehandlung ohne **Exceptions**: der Aufrufer muss beide Fälle (**Left**/**Right**) behandeln oder weiterreichen

Either
Wichtige Methoden

- **Right-biased:** In Scala (ab Version 2.12) verhält sich Either wie eine Monade auf der Right-Seite. Das heißt, map und flatMap beziehen sich auf den Right-Wert (Erfolgsfall)
- **map(f: R => R2):** Wendet **f** auf den Recht-Wert an, gibt Right(f(value)) zurück. Bei einem Left passiert nichts
- **flatMap(f: R => Either[L2, R2]):** Kettet weitere Berechnungen an: bei Right(v) wird f(v) aufgerufen, das ein neues Either ergibt; ein vorhandener Left wird einfach durchgereicht
- **getOrElse(default: R)**: Gibt bei Right(v) den Wert v zurück, bei Left(e) stattdessen den Default-Wert
- **fold(fL, fR)**: Ermöglicht die Verarbeitung beider Varianten: wendet **fL** auf den **Left**-Inhalt an oder **fR** auf den **Right**-Inhalt und gibt das Ergebnis zurück
- Weitere nützliche Helfer: **isRight/isLeft** (Prüfen der Variante), **swap** (Left/Right tauschen), **toOption** (Right in **Some** wandeln, **Left** zu None)

Either
Anwendungsbeispiel Validierung

- Angenommen, wir wollen Benutzereingaben validieren (z.B. Name und Alter) und entweder ein Objekt erstellen oder Fehler melden
- Wir können für jede Prüfung ein Either[String, X] zurückgeben, mit einer Fehlermeldung als Left oder dem validierten Wert als Right

```
def validateName(name: String): Either[String, String] =
  if (name.isEmpty) Left("Name darf nicht leer sein") else Right(name)
def validateAge(age: Int): Either[String, Int] =
  if (age < 0) Left("Alter kann nicht negativ sein") else Right(age)

val result: Either[String, Person] = for {
  n ← validateName(inputName)
  a ← validateAge(inputAge)
} yield Person(n, a)
```

- Durch Verkettung in einer For-Comprehension stoppen wir bei dem ersten Fehler (fail-fast)

innFactory

Monaden

Monaden
Was ist eine Monade?

- Formal ist eine Monade ein Typkonstrukt M[?] mit zwei grundlegenden Operationen:
  - **pure** (Unit): einen Wert in den Kontext einbetten (z.B. Option(5) ergibt Some(5))
  - **flatMap** (Bind): eine Verarbeitung auf dem eingebetteten Wert durchführen und das Ergebnis flach im selben Kontext zurückgeben
- In Scala erkennt man Monaden daran, dass sie eine flatMap-Methode haben (und üblicherweise auch map). Beispiele: Option, Either, Future, aber auch List
- **Intuitiv**: Eine Monade ist ein Berechnungskontext. Man kann darin Werte verarbeiten, ohne den Kontext manuell aufzulösen. Die Monade kümmert sich um Dinge wie Fehlerfortpflanzung, Abwesenheit, Asynchronität etc.
- **Beispiel**: Bei Option führt flatMap dazu, dass nach einem None keine weiteren Berechnungen mehr durchgeführt werden – dieser Kontrollfluss steckt in der Monaden-Implementierung

Monaden
Monaden-Gesetze

- Damit Monaden zuverlässig funktionieren, müssen sie drei Gesetze erfüllen (Informell):
  1. **Links-Identität:** „pure(x).flatMap(f)“ ist dasselbe wie „f(x)“ – ein in die Monade eingebetteter Wert verhält sich beim Binden neutral
  2. **Rechts-Identität:** m.flatMap(pure) ist dasselbe wie m – eine Monade, die man wieder in den Kontext zurückführt, ändert nichts am Wert
  3. **Assoziativität:** „(m.flatMap(f)).flatMap(g)“ ist gleichbedeutend zu „m.flatMap(x => f(x).flatMap(g))“ – die Reihenfolge der Verkettung beeinflusst das Ergebnis nicht
- Im Alltag prüft man diese Gesetze nicht aktiv, aber die Bibliotheken (z.B. **Cats**) implementieren Monaden so, dass die Gesetze gelten. Dadurch können wir uns auf erwartbares Verhalten verlassen

Monaden
Warum sind Monaden nützlich?

- Monaden bieten ein einheitliches Schema, um mit Kontexten/Effekten umzugehen: optionalen Werten, Fehlern, zeitversetzten Berechnungen, Zuständen usw.
- Dank Monaden können wir komplexe Abläufe in kleinere Schritte unterteilen und trotzdem elegant zusammensetzen (via flatMap/For-Comprehension), ohne in jeder Zwischenschritt den Kontext neu zu behandeln
- Der Code wird dadurch ausdrucksstärker und weniger fehleranfällig: z.B. keine expliziten Nullprüfungen oder try-catch an jeder Stelle – das übernimmt die Monade
- Viele Sprachfeatures und Bibliotheken in Scala nutzen Monaden: z.B. **Futures**, **Streams**, **Options** – sobald man Monaden verstanden hat, erkennt man die Muster wieder und kann sie generalisieren
- Insgesamt erhöhen Monaden die Wiederverwendbarkeit: Der gleiche For-Comprehension-Stil funktioniert für verschiedenste Monaden, was zu allgemeiner einsetzbarem Code führt

innFactory

Futures

Futures
Einführung: nebenläufige Berechnungen mit Future

- **Future[T]** repräsentiert eine Berechnung, die asynchron abläuft und irgendwann in einem von zwei Fällen endet
  - **Sucess[T]** enthält einen Wert vom Typ **T**
  - **Failure(exception)**
- **future.isCompleted** => Boolean, abgeschlossen? - oder mit Callback-Funktionen (**onComplete**, **onSuccess**, etc.) reagieren, ohne zu blockieren
- Statt imperativ zu warten, nutzt man bevorzugt Combinators wie **map** und **flatMap**, um Folgeberechnungen anzustoßen
- Future ist ebenfalls eine Monade: erlaubt also mit **For-Comprehensions** oder Ketten von **map/flatMap** geschrieben zu werden, was asynchrone Abläufe viel lesbarer macht

Futures
Syntax Beispiel

- Futures benötigen einen **ExecutionContext**
  => Hier wird der globale Scala **ExecutionContext** als **Implicit** importiert

```
import scala.concurrent.ExecutionContext.Implicits.global
val f: Future[Int] = Future {
  // auf einem Hintergrund-Thread ausgeführt
  longComputation()
}
```

Futures Exkurs
Implicits (In Scala 3: „givens“)

- Definition eines impliziten Werts mit **given**
- Defintion von impliziten Parametern in letztem Parameter-Block mit **using**
- **Scala Compiler** sucht im Kontext der Funktion nach Wert des Typs im **using**-Block und übergibt diese implizit

```
given ExecutionContext = ???

def functionRequiringEC(param: String)(using ExecutionContext): Future[String] = ???
```

Futures
Fehlerbehandlung

- Wenn innerhalb des Future-Blocks eine **Exception** geworfen wird, markiert das den Future als fehlgeschlagen (**Failure(exception)** anstelle von einem **Success**-Wert)
- Anstatt **try-catch** zu verwenden, bieten Futures eingebaute Mechanismen, um mit Fehlern funktional umzugehen:
  - **future.recover { case ex: Throwable => Ersatzwert }:** Wandelt einen fehlgeschlagenen Future in einen erfolgreichen um, indem ein Ersatzwert geliefert wird (für bestimmte Exceptions oder allgemein)
  - **future.recoverWith { case ex => andererFuture }:** Ähnlich wie recover, aber erlaubt statt eines Wertes einen neuen Future zu liefern, z.B. um eine alternative asynchrone Aktion zu starten
  - **future.fallbackTo(alternativeFuture):** Gibt einen Future zurück, der den Wert von *future* nimmt, falls erfolgreich, andernfalls den Wert des *alternativeFuture*

Futures
For-Comprehensions mit Future

```
val resultF: Future[Double] = for {
  raw ← fetchFromServer()                // Future[Double]
  processed ← Future { process(raw) }    // Future[Double]
  saved ← saveToDatabase(processed)      // Future[Boolean]
} yield if(saved) processed else 0.0
```

Jeder Schritt wartet, bis der vorherige Future erfolgreich ist

=> Sequentielle verarbeitung aber **nicht Blockierend**

Futures
Problem: Verschachtelte Monaden

- Oft trifft man auf Funktionen, die zusammengesetzte Ergebnis-Typen zurückgeben, z.B. eine Datenbankabfrage als **Future[Option[User]]** – sowohl asynchron als auch optional
- Das Handhaben solcher Typen kann umständlich werden: Möchte man damit weiterrechnen, muss man zwei Ebenen von Kontext berücksichtigen (**Future** und **Option**)

```
futureOptUser.flatMap {
  case Some(u) ⇒ getProfile(u).map {
    case Some(profile) ⇒ Some(doSomething(profile))
    case None ⇒ None
  }
  case None ⇒ Future.successful(None)
}
```

Futures
Lösung: Monad Transformer (OptionT, EitherT)

- Monad Transformer sind Konstrukte, die es erlauben, zwei (oder mehr) Monaden miteinander zu kombinieren, so dass man sie wie eine behandeln kann
- Ein **OptionT[F, A]** umhüllt z.B. einen Wert vom Typ **F[Option[A]]** (wobei **F** selbst eine Monade ist, z.B. **Future**)
  => Dadurch wird ein neuer Monaden-Typ geschaffen, der beide Ebenen kapselt
- Statt mühsam Future und Option separat zu handhaben, kann man mit OptionT direkt flatMap über die kombinierte Struktur verwenden
- Diese Transformer sind in Scala nicht Teil der Standardbibliothek, aber z.B. in **Cats** verfügbar (cats.data.OptionT, cats.data.EitherT)

Futures
Beispiel: Verwendung von OptionT

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.data.OptionT
import cats.instances.future.*  // Monad[Future]
```

```scala
val optA: Future[Option[Int]] = Future.successful(Some(5))
val optB: Future[Option[Int]] = Future.successful(Some(7))

val sumT: OptionT[Future, Int] = for {
  a ← OptionT(optA)
  b ← OptionT(optB)
} yield a + b
val sumOptF2: Future[Option[Int]] = sumT.value
```

=> Hier kümmert sich OptionT darum, dass wir im for-Ausdruck direkt mit den Int-Werten rechnen können.

Das Ergebnis **sumT** ist ein **Option** im **Future**-Kontext.

innFactory

Übung

Übung

- **Option und Either:** Implementiere folgende Methoden
  - safeDivide(a: Int, b: Int): Option[Int] − Division durch 0 abfangen
  - validateUser(email: String, password: String): Either[String, Unit] − Gültige Email, Passwort min 6 Zeichen
- **For-Comprehensions**
  - Liste auf gerade Elemente filter und diese quadrieren
  - Implementiere folgende Funktion:

```scala
def getProductPrice(productId: Int): Either[String, Double] = Right(100.0)
def getUserDiscount(userId: Int): Either[String, Double] = Left("Discount nicht gefunden")
def getShippingCost(addressId: Int): Either[String, Double] = Right(5.0)

def calculateFinalPriceEither(userId: Int, productId: Int, addressId: Int): Either[String, Double]
```

- **Futures OptionT und EitherT**

```scala
def login(username: String, pass: String): EitherT[Future, String, String]
```

  - Den Nutzer aus der Datenbank holen
  - Prüfen, ob der Nutzer existiert (sonst Fehler "User not found" zurückgeben)
  - Das Passwort prüfen (falls falsch, Fehler "Invalid password" zurückgeben)
  - Prüfen, ob die E-Mail verifiziert ist (sonst Fehler "Email not verified" zurückgeben)
  - Schließlich ein Auth-Token generieren (bei Fehler "Token generation failed" weitergeben)

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

case class User(username: String, passwordHash: String, emailVerified: Boolean)
// Nutzer aus Datenbank holen (simuliert)
def fetchUserFromDB(username: String): Future[Option[User]] = Future.successful(
  Some(User("alice", "hashed_password", true)) // oder None, falls nicht gefunden
)
// Passwortvalidierung (simuliert)
def validatePassword(inputPassword: String, hashedPassword: String): Future[Boolean] =
  Future.successful(inputPassword == "secret" && hashedPassword == "hashed_password")
// Auth-Token generieren (simuliert, könnte fehlschlagen)
def generateAuthToken(user: User): Future[Either[String, String]] =
  Future.successful(Right("token123")) // Left("Token-Erzeugung fehlgeschlagen")
```