Lernzettel

Programmieren II für Wirtschaftsinformatiker

Universität: Technische Universität Berlin
Kurs/Modul: Programmieren II für Wirtschaftsinformatiker
Erstellungsdatum: September 6, 2025

Lernzettel: Programmieren II für Wirtschaftsinformatiker

(1) Prinzipien der funktionalen Programmierung.
Die funktionale Programmierung (FP) betont Funktionen als zentrale Bausteine, unveränderliche Datenstrukturen und das Vermeiden von Nebenwirkungen. Zentrale Konzepte:

- Reine Funktionen (Pure Functions)
- Höherordentliche Funktionen (Funktionen als Werte)
- Funktionskomposition
- Rekursion statt Schleifen
- Referentielle Transparenz

Anwendung in der Wirtschaftsinformatik: bessere Modellierung von BI-Algorithmen, einfacheres Testen und bessere Voraussetzungen für Parallelität.

(2) Unterschiede zwischen objektorientierter und funktionaler Programmierung.
Objektorientierung (OO):

- Stateful Objects, Klassen und Vererbung
- Side Effects und mutable State
- Polymorphie durch Klassenhierarchien

Funktionale Programmierung (FP):

- Funktionen als Werte, keine oder wenige Seiteneffekte
- Unveränderliche Daten (Immutability)
- Referentielle Transparenz, leichteres Reasoning
- Häufigere Nutzung von Rekursion, höherordentlichen Funktionen

Beziehung: FP kann OO ergänzen; hybride Designs nutzen beides gezielt.

(3) Grundlagen von Scala.
Scala ist eine multi-paradigmische Sprache auf der JVM, die objektorientierte und funktionale Konzepte vereint.

- Statisch typisiert, Typinferenz
- Val (unveränderlich) vs. Var (veränderlich)
- Defs als Funktionen, Höherordentliche Funktionen
- Collections, Option, Either, Try
- Case Classes, Pattern Matching
- Interoperabilität mit Java

Beispiel (Verwendung von Listen und map)

```
val names = List("Alice", "Bob", "Carol")
def greet(n: String): String = s"Hallo, $n!"
val greetings = names.map(greet)
```

Beispiel für Typinferenz:

```
val nums = List(1, 2, 3, 4)
```

(4) Interoperabilität Java – Scala.
Scala läuft auf der JVM und kann Java-Bibliotheken direkt verwenden.

- Von Scala aus Java-Klassen nutzen (Import, Instanziierung, Methodenaufrufe)
- Java-Objekte als Scala-Objekte behandeln; Null-Safety beachten
- Gemeinsame Build-Tools (sbt, Maven/Gradle) unterstützen beide Sprachen

Vorteil: vorhandene Java-Ökosysteme nutzbar, schrittweise Migration zu FP-Paradigmen.

(5) Nebenläufigkeit und Parallelität in Java und Scala.
Java:

- Threads, Executors, Futures
- CompletableFuture, Streams (parallele Ausführung)

Scala:

- Futures und Promises, configurable ExecutionContext
- Akka (Actor-Modell) als primäres Muster für Message-Passing-Concurrency

Hinweis: Unveränderliche Datenstrukturen unterstützen sichere Parallelität.

(6) Fortgeschrittene Konzepte von Scala: Closures, Traits und Exceptions.
Closures:

- Funktionen, die Umgebungsvariablen erfassen
- Ermöglichen flexible API-Designs und Scoping

```
val multiplier = (x: Int) => x * 2
val apply = (f: Int => Int, z: Int) => f(z)
apply(multiplier, 10) // 20
```

Traits:

- Mischung von Verhalten (ähnlich wie Interfaces mit Implementierung)
- Mehrfachvererbung ohne Diamantenproblem
- Geeignet für wiederverwendbares Verhalten in Klassen

Beispiel sinnvoll in stilvollen Domänen-Interfaces.
Exceptions:

- Scala hat Exceptions wie Java, aber häufig werden Try, Option oder Either genutzt
- Beispiel mit Try:

```
import scala.util.{Try, Success, Failure}
val result: Try[Int] = Try(10 / 0)
result match {
  case Success(v) => println(v)
  case Failure(e) => println("Fehler: " + e.getMessage)
}
```

(7) Einführung in die Message-Passing-Concurrency mit Akka.
Ziel: robuste, verteilte Systeme durch Actors-Modell.

- Actors kommunizieren ausschließlich über Nachrichten
- ActorSystem als Wurzel, Actors als UNITS
- Supervisoren steuern Fehlerbehandlung und Wiederherstellung
- Skalierung durch verteilt arbeitende Actor-Systeme

Kurzer Einstieg:

```
import akka.actor.{Actor, ActorSystem, Props}

class PrintActor extends Actor {
  def receive: Receive = {
    case msg: String => println(msg)
  }
}
val system = ActorSystem("BIApp")
val printer = system.actorOf(Props[PrintActor], "printer")
printer ! "Hallo aus Akka"
```