Übungsaufgabe

Prinzipien der funktionalen Programmierung: Reinheit, Referentielle Transparenz und unveränderliche Daten.

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

Aufgabe 1: Reinheit, Referentielle Transparenz und unveränderliche Daten

In dieser Aufgabe geht es um die drei zentralen Prinzipien der funktionalen Programmierung: Reinheit, referentielle Transparenz und der Umgang mit unveränderlichen Daten. Veranschaulichen Sie die Konzepte anhand von Scala-Beispielen und kurzen Erläuterungen.

a) Definieren Sie Reinheit und referentielle Transparenz in eigenen Worten. Geben Sie jeweils ein kurzes praktisches Beispiel in Scala an, das eine rein funktionale (reinheitliche) Implementierung illustriert bzw. ein Beispiel, das von Seitenwirkungen (Nebenwirkungen) geprägt ist.

```
def pure(x: Int): Int = x + 1
var counter = 0
def impure(x: Int): Int = { counter += 1; x + counter }
```

b) Sind die folgenden Funktionen rein oder nicht rein? Begründen Sie Ihre Entscheidung kurz.

```
object PT {
  var count = 0
  def f1(x: Int): Int = x + 1
  def f2(x: Int): Int = { count += 1; x + count }
}
```

c) Diskutieren Sie, wie sich fehlende referentielle Transparenz auf Optimierungen auswirken kann. Wählen Sie zwei Aussagen und beurteilen Sie, ob sie unter rein funktionalen Annahmen gelten:
- Durch referentielle Transparenz lassen sich Ausdrucksbausteine durch deren Werte ersetzen, ohne das Programmverhalten zu ändern.
- Nicht deterministische Ausdrücke (z. B. durch I/O oder zufällige Zahlen) zerstören diese Eigenschaft.

d) Ergänzen Sie zwei weitere Beispiele, die Nicht-Referentialität zeigen (jeweils kurz erklären, warum die Transparenz verletzt ist).

```
def readLine(): String = scala.io.StdIn.readLine() // Seiteneffekt
def now(): Long = System.currentTimeMillis()     // Nicht-deterministisch
```

Aufgabe 2: Unveränderliche Daten und setzende Konzepte

In dieser Aufgabe soll die Rolle unveränderlicher Daten (Persistenz) sowie deren Nutzen für Parallelität/Verlässlichkeit in der Programmierung herausgearbeitet werden.

a) Erklären Sie, warum Listen in Scala standardmäßig unveränderlich sind und welche Vorteile dies für Nebenläufigkeit und Parallelität bietet. Geben Sie dazu ein kurzes, klares Beispiel in Scala an (ohne Mutation).

```
val xs = List(1, 2, 3)
val ys = 0 :: xs          // xs bleibt unverändert
```

b) Gegeben sei folgende Fallklasse. Implementieren Sie eine Funktion, die eine unveränderliche Kopie mit geändertem Feldwert erzeugt, statt das Original zu verändern.

```
case class Konto(name: String, saldo: Double)
def ueberweisen(k: Konto, betrag: Double): Konto =
  k.copy(saldo = k.saldo + betrag)
```

c) Diskutieren Sie, wie immutable Daten Strukturen die Nachvollziehbarkeit von Programmen verbessern und welche Auswirkungen dies auf Tests und Debugging hat. Geben Sie drei Stichworte an.

Aufgabe 3: Praktische Anwendungen der Reinheit und Unveränderlichkeit in Scala

In diesem Abschnitt soll praktisch demonstriert werden, wie rein funktionale Muster in typischen Aufgabenstellungen eingesetzt werden können.

a) Schreibe eine rein funktionale Funktion, die aus einer Liste von Zahlen deren Quadrate bildet, und gib die neue Liste zurück (keine Mutation der Eingabe).

```
defQuadrate(items: List[Int]): List[Int] = items.map(n => n * n)
```

b) Verwenden Sie unveränderliche Datenstrukturen, um eine einfache Transformation durchzuführen, z. B. das Zusammenführen zweier Listen zu einer neuen Liste.

```
val a = List(1, 2, 3)
val b = List(4, 5)
val zusammen = a ++ b
```

c) Beschreiben Sie, wie sich das Muster map/foldLeft in Scala dazu eignet, Funktionen ohne Seiteneffekte zu kapseln. Skizzieren Sie zwei kurze Aufgaben, die Sie damit lösen würden (ohne Lösung).

```
def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
def quadriertenSumme(xs: List[Int]): Int = xs.map(x => x * x).sum
```

Lösungen

Aufgabe 1: Reinheit, Referentielle Transparenz und unveränderliche Daten

Lösung:

a) Reinheit und referentielle Transparenz

- Reinheit: Eine Funktion ist rein, wenn sie keinerlei Nebeneffekte hat und bei gleichen Eingaben immer das gleiche Ergebnis liefert.
- Referentielle Transparenz: Ausdrücke können durch ihren Wert ersetzt werden, ohne Beobachtbares am Programmverhalten zu ändern.

```
def pure(x: Int): Int = x + 1
var counter = 0
def impure(x: Int): Int = { counter += 1; x + counter }
```

Beispiel-Erläuterung: `pure` ist rein; `impure` ist nicht rein, da sie eine Veränderung (Nebeneffekt) an einer globalen Variable erzeugt und das Ergebnis abhängig von der Ausführung ist.

b) Sind die folgenden Funktionen rein oder nicht rein?

```
object PT {
  var count = 0
  def f1(x: Int): Int = x + 1
  def f2(x: Int): Int = { count += 1; x + count }
}
```

Lösung:

- `f1`: rein.
- `f2`: nicht rein, da `count` moduliert wird und das Ergebnis von externem Zustand abhängt.

c) Diskutieren Sie, wie sich fehlende referentielle Transparenz auf Optimierungen auswirken kann. Wählen Sie zwei Aussagen und beurteilen Sie, ob sie unter rein funktionalen Annahmen gelten:

- Durch referentielle Transparenz lassen sich Ausdrucksbausteine durch deren Werte ersetzen, ohne das Programmverhalten zu ändern. **Gilt unter rein funktionalen Annahmen.**
- Nicht deterministische Ausdrücke (z. B. durch I/O oder zufällige Zahlen) zerstören diese Eigenschaft. **Gilt unter rein funktionalen Annahmen; unproblematisch, wenn Nicht-RT vermieden wird.**

d) Ergänzen Sie zwei weitere Beispiele, die Nicht-Referentialität zeigen (jeweils kurz erklären, warum die Transparenz verletzt ist).

```
def randomInt(): Int = scala.util.Random.nextInt()
def logAndReturn(x: Int): Int = { println(x); x }
```

Lösungs-Erklärungen:

- `randomInt`: Nicht-Determinismus/Nicht-Determinismus – jedes Aufrufen kann unterschiedlich ausfallen; ersetzt man den Ausdruck durch seinen Wert, ändert sich das Programmverhalten.
- `logAndReturn`: Seiteneffekt (Ausgabe auf Konsole) – das Verhalten hängt von der Ausführung ab und lässt sich nicht allein durch den Wert ersetzen.

Aufgabe 2: Unveränderliche Daten und setzende Konzepte

Lösung:

a) Erklären Sie, warum Listen in Scala standardmäßig unveränderlich sind und welche Vorteile dies für Nebenläufigkeit und Parallelität bietet. Geben Sie dazu ein kurzes, klares Beispiel in Scala an (ohne Mutation).

```
val xs = List(1, 2, 3)
val ys = 0 :: xs          // xs bleibt unverändert
```

Lösungserläuterung: Listen in Scala sind standardmäßig unveränderlich (immutable). Dadurch entstehen keine Race Conditions durch gleichzeitige Modifikationen, und parallele oder nebenläufige Zugriffe bleiben deterministisch. Neue Listen entstehen durch Konstruktionen wie `::` oder `List(...)`, wobei das Original unverändert bleibt.

b) Gegeben sei folgende Fallklasse. Implementieren Sie eine Funktion, die eine unveränderliche Kopie mit geändertem Feldwert erzeugt, statt das Original zu verändern.

```
case class Konto(name: String, saldo: Double)
def ueberweisen(k: Konto, betrag: Double): Konto =
  k.copy(saldo = k.saldo + betrag)
```

c) Diskutieren Sie, wie immutable Daten Strukturen die Nachvollziehbarkeit von Programmen verbessern und welche Auswirkungen dies auf Tests und Debugging hat. Geben Sie drei Stichworte an.

- Determinismus
- Reproduzierbarkeit
- Nachvollziehbarkeit / Einfacheres Debugging

Aufgabe 3: Praktische Anwendungen der Reinheit und Unveränderlichkeit in Scala

Lösung:

a) Schreibe eine rein funktionale Funktion, die aus einer Liste von Zahlen deren Quadrate bildet, und gib die neue Liste zurück (keine Mutation der Eingabe).

```
def quadrate(items: List[Int]): List[Int] = items.map(n => n * n)
```

b) Verwenden Sie unveränderliche Datenstrukturen, um eine einfache Transformation durchzuführen, z. B. das Zusammenführen zweier Listen zu einer neuen Liste.

```
val a = List(1, 2, 3)
val b = List(4, 5)
val zusammen = a ++ b
```

c) Beschreiben Sie, wie sich das Muster map/foldLeft in Scala dazu eignet, Funktionen ohne Seiteneffekte zu kapseln. Skizzieren Sie zwei kurze Aufgaben, die Sie damit lösen würden (ohne Lösung).

- Aufgabe 1: Berechne die Summe einer Liste von Integers (ohne Mutation) durch `foldLeft`.
- Aufgabe 2: Transformiere eine Liste von Strings, bilde jeweils deren Länge (mit `map`) und fasse die Längen zusammen (ohne Seiteneffekte).

```
def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
def quadriertenSumme(xs: List[Int]): Int = xs.map(x => x * x).sum
```