Function Chaining and Composition

Was ist ein Functor in Scala?

= ein Container mit Elementen mit generischem Typ, besitzt `map`-Methode mit der man eine Funktion auf die Elemente anwenden kann ohne die äußere Struktur zu verändern

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

Welche Functoren gibt es in Scala?

- List
- Option
- Either

Welche Gesetztze gelten für Functoren?

- Identitätsgesetz: `fa.map(x => x) == fa` -> `map` mit Identitätsfunktion gibt das gleiche zurück
- Kompositionsgesetz: `fa.map(f).map(g) == fa.map(x => g(f(x)))` -> Reihenfolge der Funktionen ist egal

Was ist eine Monade in Scala?

= ein Functor mit zusätzlichen Methoden `flatMap` und `unit`

- `unit` - erzeugt eine Monade aus einem Wert
- `flatMap` - wendet eine Funktion auf den Inhalt der Monade an und gibt eine neue Monade zurück

Welche Monaden gibt es in Scala?

- List

```
val numbers = List(1, 2, 3)
val squared = numbers.flatMap(x => List(x, x * x))

println(squared) // List(1, 1, 2, 4, 3, 9)
```

- Option

```
def divide(a: Int, b: Int): Option[Int] =
  if (b == 0) None else Some(a / b)
```

```scala
val result = Some(10).flatMap(x => divide(x, 2)) // Some(5)
val fail = Some(10).flatMap(x => divide(x, 0))   // None

println(result) // Some(5)
println(fail)   // None
```

- Try
- Future
- Stream
- Reactive Streams
- Reader
- Writer
- State
- Parser
- Gen

Welche Gesetze gelten für Monaden?

- Identitätsgesetz
- Assoziativgesetz: `(m flatMap f) flatMap g == m flatMap (x => f(x) flatMap g)`

Was is eine For-Comprehension in Scala?

= syntaktischer Zucker für `flatMap` und `map`

```scala
for (x <- collection) yield expression
```

`yield` gibt eine Sammlung zurück, die aus den transformierten Elementen besteht

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

val squaredNumbers = for (n <- numbers) yield n * n

println(squaredNumbers) // List(1, 4, 9, 16)
```

äquivalent zu

```scala
val squaredNumbers = numbers.map(n => n * n)
```

Was ist eine Function Composition in Scala?

= Verkettung von Funktionen, wobei das Ergebnis der ersten Funktion als Argument für die zweite Funktion dient

Vorwärtsverkettung -> mit `andThen` führt zuerst die erste Funktion aus und dann die zweite

```
val add5: Int => Int = _ + 5
val multiply2: Int => Int = _ * 2

val combined = add5 andThen multiply2

println(combined(10)) // (10 + 5) * 2 = 30
```

Rückwärtsverkettung -> mit `compose` führt zuerst die zweite Funktion aus und dann die erste

```
val combined2 = add5 compose multiply2

println(combined2(10)) // (10 * 2) + 5 = 25
```

mit mehreren Funktionen

```
val subtract3: Int => Int = _ - 3
val divide2: Int => Int = _ / 2

val complexFunction = add5 andThen multiply2 andThen subtract3 andThen divide2

println(complexFunction(10)) // (((10 + 5) * 2) - 3) / 2 = 11
```

Was sind Extension Methods in Scala?

= Methoden, die an bestehende Klassen angehängt werden können, ohne die Klasse selbst zu ändern

```
extension (x: String)
  def hello: String = s"Hello, $x!"
  def wordCount: Int = x.split(" ").length
```

```
println("Scala".hello)     // "Hello, Scala!"
println("I love Scala".wordCount) // 3
```

Wie funktioniert das Ordering-Trait in Scala und welche Methoden erbt es?

= Trait, das die Ordnung von Objekten definiert, erbt die Methoden `compare`, `lt`, `lteq`, `gt`, `gteq`, `equiv`

Sortieren einer Personenklasse:

```
case class Person(firstName: String, lastName: String, age: Int)

// Sortiere zuerst nach Nachname, dann Vorname, dann Alter
val orderByLastName: Ordering[Person] = Ordering.by(_.lastName)
val orderByFirstName: Ordering[Person] = Ordering.by(_.firstName)
val orderByAge: Ordering[Person] = Ordering.by(_.age)

val personOrdering = orderByLastName
  .orElse(orderByFirstName)
  .orElse(orderByAge)

val people = List(
  Person("Anna", "Müller", 30),
  Person("Peter", "Schmidt", 25),
  Person("Anna", "Müller", 28)
)

// Sortieren der Liste nach dem definierten Ordering
val sortedPeople = people.sorted(personOrdering)

println(sortedPeople)
```