Lambda Expressions

Was sind Lambda Expressions?

= kompakte Schreibweise um Funktionen als Literal zu definieren, ohne Namen zu geben (anonyme Funktionen)
-> können als Argumente übergeben, gespreichert und zurückgegeben werden

```
(x: Int) => x < 0
```

Was sind High Order Functions?

= Funktionen, die andere Funktionen als Argumente entgegennehmen oder zurückgeben

```
def apply(f: Int => Int, x: Int) = f(x)
```

Was ist eine Closure?

= eine Funktion, die auf eine oder mehrere Variablen aus ihrem äußeren Gültigkeitsbereich zugreift. Diese Variablen bleiben erhalten, auch wenn die Closure an einem anderen Ort im Programm verwendet wird.

```
var faktor = 2  // Eine Variable außerhalb der Funktion

val multiplizieren = (x: Int) => x * faktor  // Zugriff auf "faktor"

println(multiplizieren(5))  // 10

faktor = 3  // Ändern der äußeren Variable
println(multiplizieren(5))  // 15
```

Welche zwei Arten von Closures gibt es?

Reine Closures:

- greifen nur auf unveränderliche Variablen zu
- keine Seiteneffekte
- deterministisch

```
val faktor = 2  // Unveränderliche Variable
val multiplizieren = (x: Int) => x * faktor  // Greift auf "faktor" zu
```

```
println(multiplizieren(5))  // Immer 10
```

Impure Closures:

- greifen auf veränderliche Variablen zu
- Seiteneffekte
- nicht deterministisch

```
var faktor = 2  // Veränderliche Variable
val multiplizieren = (x: Int) => x * faktor  // Greift auf "faktor" zu

println(multiplizieren(5))  // 10

faktor = 3  // Ändert die äußere Variable
println(multiplizieren(5))  // Jetzt 15!
```

Welche Vorteile bieten Higher-Order Functions in der funktionalen Programmierung?

- allgemein Funktion für verschiedene Eingaben und Operationen wiederverwendetbar

```
def applyToList[A, B](list: List[A], fn: A => B): List[B] = list.map(fn)

val numbers = List(1, 2, 3)
val squared = applyToList(numbers, x => x * x) // List(1, 4, 9)
val doubled = applyToList(numbers, x => x * 2) // List(2, 4, 6)
```

- weniger Boilerplate Code

```
val evenNumbers = List(1, 2, 3, 4, 5, 6).filter(_ % 2 == 0)
// Statt einer expliziten Schleife mit `if`-Bedingungen.
```

- kombinieren von Funktionen
  -- Lazy Evaluation

```
val numbers = List(1, 2, 3, 4, 5)
val lazyFiltered = numbers.view.filter(_ % 2 == 0).map(_ * 10)
println(lazyFiltered.toList) // List(20, 40) – erst jetzt wird die Berechnung
ausgeführt
```

- Parallelisierung

```scala
val numbers = List(1, 2, 3, 4, 5)
val squared = numbers.par.map(x => x * x) // Parallel berechnet
```