Wozu Scala wenn's doch Kotlin gibt?

Michael Sperber

Created: 2024-06-30 Sun 17:31

Wozu Scala wenn's doch Kotlin gibt?

Michael Sperber

Active Group GmbH

@sperbsen@discuss.systems

Funktionale Softwarearchitektur

- Tübingen
- Softwareprojektem, Beratung, Reviews
- Scala, Clojure, F#, Haskell, OCaml, Erlang, Elixir, Swift
- Schulungen (iSABQ Foundation, FUNAR, FLEX, DSL)

Blog (German): https://funktionale-programmierung.de

Scala

- Martin Odersky, EPFL
- "OOP + FP"
- Scala 1.0: 2004
- Scala 2.0: 2006 (noch aktiv, Lightbend)
- Scala 3.0: 2021 (neuer Compiler)

Kotlin

- JetBrains
- "better language than Java"
- schneller compilieren als Scala
- einfacher als Scala
- Kotlin 1.0: 2016
- Kotlin 2.0: 2024

Kotlin ist besser als Java

- `null` explizit im Typsystem
- Companion Objects statt statischer Methoden
- primitive Typen und Referenztypen vereint
- Generics: use-site variance (keine Wildcards)
- `data`-Klassen (aber siehe `record` in Java)
- Funktionstypen
- Annotation für endrekursive Funktionen
- unveränderliche Collections
- Extension Functions
- benamste Parameter, Default-Argumente
- Coroutinen

Scala ist besser als Java

- ~~`null` explizit im Typsystem~~
- Companion Objects statt statischer Methoden
- primitive Typen und Referenztypen vereint
- Generics: use-site variance (keine Wildcards)
- `enum`-Klassen (aber siehe `record` in Java)
- Funktionstypen
- Annotation für endrekursive Funktionen
- ~~unveränderliche Collections~~ funktionale Datenstrukturen
- Extension Functions
- named parameters, default arguments
- benamste Parameter, Default-Argumente
- ~~Coroutinen~~ Monaden-Syntax
- **Higher-Kinded Types**
- **Implicits und Typklassen**

Kotlin/Scala-Features, die nicht von FP kommen

- "use-site variance"
- "extension functions"
- benamste Parameter, Default-Argumente

OOP vs. FP

OOP	FP
Klassen, Methoden, Objekte	Funktionen, Daten
Zustand durch Mutation	durchgängige Unveränderlichkeit
abstrakte Klassen	algebraische Datentypen
nominale Typsysteme	strukturelle Typsysteme
Turm von Konzepten	Higher-Order-ABstraktionen

FP-Vorteile

- einfache Datenmodelle
- Alles ist Daten
- leichtere Abstraktion
- weniger Fehler
- weniger Kopplung
- weniger Komplexität

Scala FTW

```
type PH = Double

enum Hairtype {
  case Straight
  case Wavy
  case Curly
  case Coily
}

enum ShowerProduct {
  case Soap(pH: PH)
  case Shampoo(hairtype: Hairtype)
  case Mixture(product1: ShowerProduct, product2: ShowerProduct)
}
```

Kotlin FTW

```
typealias PH = Double

enum class Hairtype {
    STRAIGHT, WAVY, CURLY, COILY
}

sealed interface ShowerProduct { ... }
data class Soap(val pH: PH): ShowerProduct
data class Shampoo(val hairtype: Hairtype): ShowerProduct
data class Mixture(val product1: ShowerProduct,
                   val product2: ShowerProduct)
    : ShowerProduct
```

Methoden vs. Funktionen in Scala

```
enum ShowerProduct {
  ...
  def soapProportion: Double =
    this match {
      case Soap(pH) => 1.0
      case Shampoo(hairtype) => 0.0
      case Mixture(product1, product2) =>
        (product1.soapProportion + product2.soapProportion) / 2
    }
}

def soapProportion(product: ShowerProduct): Double =
  product match {
    case Soap(pH) => 1.0
    case Shampoo(hairtype) => 0.0
    case Mixture(product1, product2) =>
      (soapProportion(product1) + soapProportion(product2)) / 2
  }
```

Methoden vs. Funktionen in Kotlin

```
sealed interface ShowerProduct {
    fun soapProportion(): Double =
        when (this) {
            is Soap -> 1.0
            is Shampoo -> 0.0
            is Mixture ->
                (this.product1.soapProportion() +
                    this.product2.soapProportion()) / 2
        }
}

fun soapProportion(product: ShowerProduct): Double =
    when (product) {
        is Soap -> 1.0
        is Shampoo -> 0.0
        is Mixture ->
            (product.product1.soapProportion() +
                product.product2.soapProportion()) / 2
    }
```

Companion Object in Scala

```
object ShowerProduct {
  def soapProportion(product: ShowerProduct): Double =
    product match {
      case Soap(pH) => 1.0
      case Shampoo(hairtype) => 0.0
      case Mixture(product1, product2) =>
        (soapProportion(product1) + soapProportion(product2)) / 2
    }
}
```

Companion Object in Kotlin

```
sealed interface ShowerProduct {
    companion object {
        fun soapProportion(product: ShowerProduct): Double =
            when (product) {
                is Soap -> 1.0
                is Shampoo -> 0.0
                is Mixture ->
                    (product.product1.soapProportion() +
                        product.product2.soapProportion()) / 2
            }
    }
}
```

Extension Methods in Scala

```scala
extension (product: ShowerProduct)
  def soapProportion: Double =
    product match {
      case Soap(pH) => 1.0
      case Shampoo(hairtype) => 0.0
      case Mixture(product1, product2) =>
        (soapProportion(product1) + soapProportion(product2)) / 2
    }
```

Extension Methods in Kotlin

```
fun ShowerProduct.soapProportion(): Double =
    when (this) {
        is Soap -> 1.0
        is Shampoo -> 0.0
        is Mixture ->
            (this.product1.soapProportion() + this.product2.soapProportion()) / 2
    }
```

Funktionstypen in Kotlin

```kotlin
sealed interface Optional<out A>

data object None: Optional<Nothing>
data class Some<out A>(val value: A): Optional<A>

fun <A, B> map(f: (A) -> B, optional: Optional<A>): Optional<B> =
    when (optional) {
        is None -> None
        is Some -> Some(f(optional.value))
    }
```

Funktionstypen in Scala

```scala
enum Optional[+A] {
  case None
  case Some(value: A)
}

def map[A, B](f: A => B, optional: Optional[A]): Optional[B] =
  optional match {
    case None => None
    case Some(value) => Some(f(value))
  }
```

Generics in Java

```
interface Stream<T> {
  <R> Stream<R> map(Function<? super T,? extends R> mapper)
}
```

Bessere Generics in Kotlin

```
public interface List<out E> : Collection<E> {
    ...
}

public fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    ...
}
```

Bessere Generics in Scala

```
trait Seq[+A] {
  def map[B](f: A => B): Seq[B]
}
```

Funktionale Listen in Kotlin

```
sealed interface List<out A>
object Empty : List<Nothing>
data class Cons<A>(val first: A, val rest: List<A>) : List<A>
```

Liste umdrehen in Kotlin

```
fun <A> List<A>.reverse2(): List<A> {
    var list = this
    var acc: List<A> = Empty
    while (list is Cons) {
        acc = Cons(list.first, acc)
        list = list.rest
    }
    return acc
}
```

Endrekursion in Kotlin

```
fun <A> List<A>.reverse(): List<A> {

    fun <A> reverseHelper(list: List<A>, acc: List<A>): List<A> =
        when (list) {
            is Empty -> acc
            is Cons ->
                reverseHelper(list.rest, Cons(list.first, acc))
        }
    return reverseHelper(this, Empty)
}
```

Endrekursion in Kotlin

```
fun <A> List<A>.reverse(): List<A> {
    tailrec
    fun <A> reverseHelper(list: List<A>, acc: List<A>): List<A> =
        when (list) {
            is Empty -> acc
            is Cons ->
                reverseHelper(list.rest, Cons(list.first, acc))
        }
    return reverseHelper(this, Empty)
}
```

Endrekursion in Scala

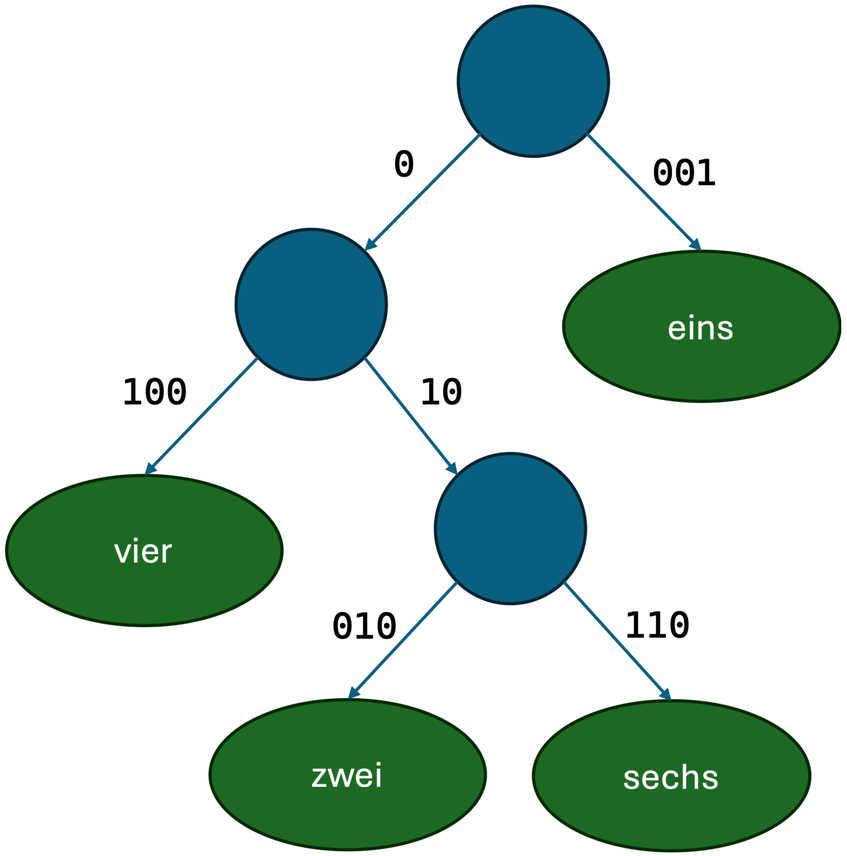
```scala
def reverse[A](list: List[A]): Unit = {
  @tailrec
  def revHelper(list: List[A], acc: List[A]): List[A] =
    list match {
      case Nil => acc
      case first::rest =>
        revHelper(rest, first::acc)
    }
  revHelper(list, Nil)
}
```

Unveränderliche Listen in Kotlin

```kotlin
val list1 = listOf(1,2,3)
val list2 = listOf(4,5,6)
val list3 = list1 + list2
```

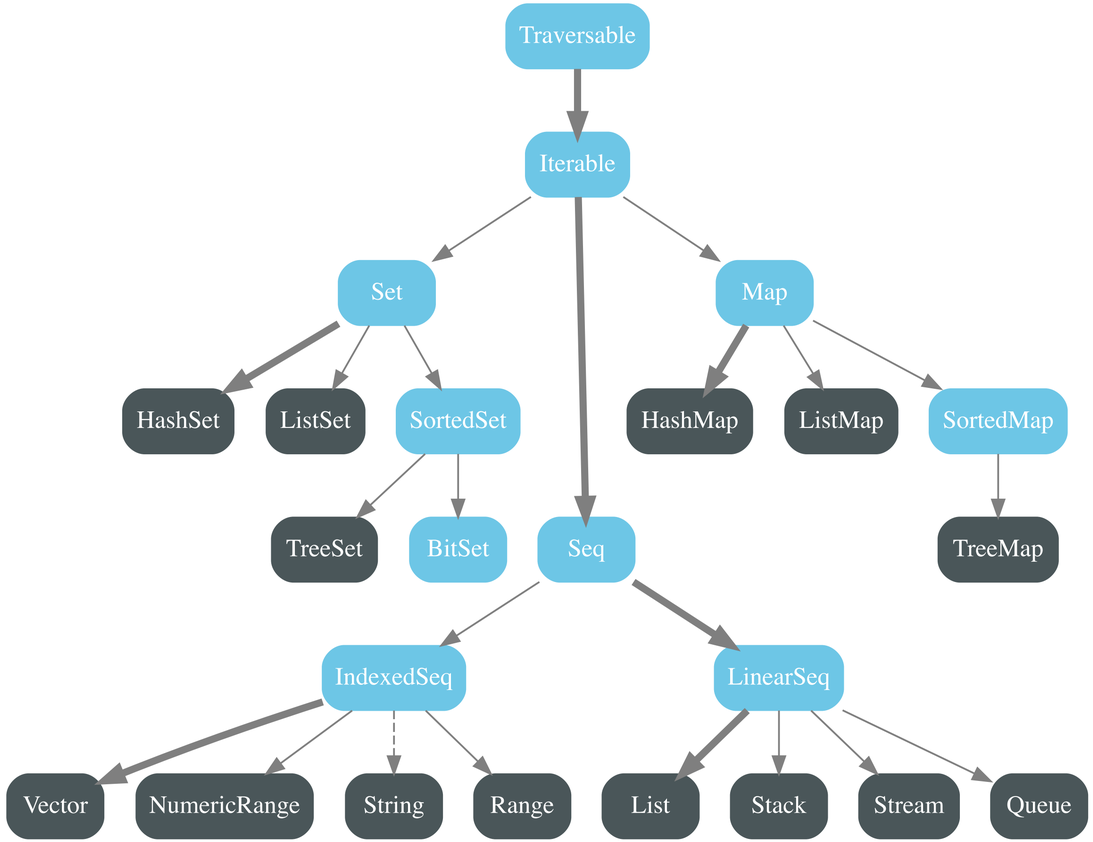
```kotlin
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.size > 0) elements.asList() else emptyList()

...

public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }
```

`Vector` in Scala



(Tatsächlich: Verzweigung 32)

Collections In Scala



https://docs.scala-lang.org/overviews/collections/overview.html

Geschönfinkelte Funktionen in Scala

```scala
def map2[A, B](optional: Optional[A])(f: A => B): Optional[B] =
  optional match {
    case None => None
    case Some(value) => Some(f(value))
  }

def o1: Optional[Int] = Some(5)
def o2 = map2(o1) { _ + 1 }
```

Syntaktischer Zucker für Lambdas in Kotlin

```
fun <A, B> map(optional: Optional<A>, f: (A) -> B): Optional<B> =
    when (optional) {
        is None -> None
        is Some -> Some(f(optional.value))
    }

val o1: Optional<Int> = Some(5)
val o2 = map(o1) { it + 1}
```

Funktoren in Scala

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

Funktoren in Kotlin

```
interface Functor<F> {
    fun <A, B> map(f: (A) -> B, thing: F<A>): F<B>
}
```

"Type arguments are not allowed for type parameters."

Monaden in Scala

```scala
enum Optional[+A] {
  case None
  case Some(value: A)

  def map[B](f: A => B): Optional[B] =
    this match {
      case None => None
      case Some(value) => Some(f(value))
    }

  def flatMap[B](f: A => Optional[B]): Optional[B] =
    this match {
      case None => None
      case Some(value) => f(value)
    }
}
```

Monaden in Scala

```scala
def optionalAdd(o1: Optional[Int], o2: Optional[Int]): Optional[Int] =
  for {
    i1 <- o1
    i2 <- o2
  } yield i1 + i2
```

Couroutinen in Kotlin

```
fun optionalAdd(o1: Option<Int>, o2: Option<Int>): Option<Int> =
    optionally {
        val i1 = o1.susp()
        val i2 = o2.susp()
        pure(i1 + i2)
    }
```

https://github.com/active-group/kotlin-free-monad

Implicits in Scala

```scala
given functorForOption: Functor[Option] = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa match {
    case None    => None
    case Some(a) => Some(f(a))
  }
}

extension [F[_], A] (thing: F[A]) {
  def map[B](f: A => B)(using functor: Functor[F]): F[B] =
    functor.map(thing)(f)
}
```

Funktionen mit "Receiver" in Kotlin

```
object OptionDSL {
    suspend fun  <A> pure(result: A): A = MonadDSL.pure(result) { Some(it) }
}
...
fun <A> optionally(block: suspend OptionDSL.() -> A): Option<A> =
    MonadDSL.effect(OptionDSL, block)
...

fun <DSL, FA, A> effect(dsl: DSL,
                        block: suspend DSL.() -> A,
                        coroutineContext: CoroutineContext
                            = EmptyCoroutineContext): FA {
    ...
}
```

Funktionale Architektur

- unveränderliche Daten
- funktionale Datenstrukturen
- Kombinatormodelle
- Abstraktion
- höherstehende Abstraktionen aus der Algebra

Funktionale Architektur in Scala

- unveränderliche Daten ✅
- funktionale Datenstrukturen ✅
- Kombinatormodelle ✅
- Abstraktion ✅
- höherstehende Abstraktionen aus der Algebra ✅

Funktionale Architektur in Kotlin

- unveränderliche Daten ✅
- funktionale Datenstrukturen ❌
- Kombinatormodelle ✅
- Abstraktion ✅
- höherstehende Abstraktionen aus der Algebra ❌

IDE-Unterstützung

- Fehlermeldungen
- Compilier-Geschwindigkeit
- REPL

Zusammenfassung

- Java OK
- Kotlin gut
- Scala besser