Vorlesung

Cloud- und Web-Technologien

(CS3140)

Einführung in Kotlin

Professor Dr. rer. nat. habil. Sven Groppe
https://www.ifis.uni-luebeck.de/index.php?id=groppe

Chronologische Übersicht über die Themen

Nr	Thema	
1	Einleitung	
2	Einführung in das Semantic Web, RDF und SPARQL	Datenmodell
3	Die Semantic Web-Ontologiesprachen RDFS und OWL	
4	**Multiplattform-Entwicklung mit Kotlin**	Multiplattform
5	Fortgeschrittene Themen mit Kotlin	
6	Einstieg in Cloud Computing, Hadoop	Backend
7	Operatoren der relationalen Algebra in Hadoop	
8	Datenverarbeitung mit Pig	
9	Einführung in Spark und Flink	
10	Stromverarbeitung mit Flink	
11	Knotenzentrische Algorithmen mit Flink	
12	HTML und CSS	Web
13	Browserprogrammierung mit JS/JQuery und Serverprogrammierung mit PHP Hypertext Preprocessor	
14	Zusammenfassung und Ausblick	

Kotlin - Geschichte & Motivation

- Geschichte
  - seit 2010 in der Entwicklung (JetBrains)
  - 2016 erste offizielle stabile Version (Kotlin v1.0)
  - 2017 verkündete Google "first-class support" für Kotlin auf Android
- Motivation für die Entwicklung von Kotlin
  - Kompabilität zu Java wegen JetBrains großer Java-Codebasis
  - Limitationen von Java aufheben
    - laut JetBrains Scala einzige brauchbare Alternative unter JVM-Sprachen, aber
      - Kompilierzeiten von Scala (zu) langsam
      - Aufruf von Scala-APIs aus Java meist unschön
      - Scalas Design (aus **akademischer Herkunft**):
        Was ist bei Programmiersprachen möglich?
        Aber Industrie: Was ist für den produktiven Einsatz wichtig?
      - Komplexität der Scala-Sprache häufig zu Lasten der Code-Lesbarkeit

Design-Ziele von Kotlin 1/2

- vollständige Kompabilität zu Java und zur JVM
- Unterstützung von verschiedenen Zielplattformen und Multiplattform-Entwicklung
  - Java/JVM (Desktop, Server & Android), JS und via LLVM nativ für MacOS, iOS, Android (arm), Linux, Win, WebAssembly
- Komfort für den Entwickler
  - Gute Integration in IntelliJ IDEA (IDE von JetBrains)
  - Prägnanter Quelltext
    - durch Übernahme von ausgewählten Sprachfeatures aus modernen Programmiersprachen
    - Sprachkonstrukte vermeiden *aufgeblähten* Code für wenige Funktionalitäten (wie häufig in Java)
  - Lesbarer Quelltext
    - durch Verzicht auf (zu) komplexe Sprachfeatures

Design-Ziele von Kotlin 2/2

- Parallele Unterstützung mehrerer Programmierparadigmen
  - objektorientiert
  - imperativ/prozedural
  - funktional
- statisch typisiert
  - weniger Laufzeitfehler und stabilere Programme als dynamisch typisiert
  - **Aber:** Automatische Inferenz von Datentypen durch den Compiler
    - Zumeist Angabe von Datentypen überflüssig
- elegante Standardbibliotheken mit wenig Overhead bei der Ausführung
  - ⇝ häufige Verwendung von *inlining*

Grundlagen der Syntax

Bezeichner

```
java_id = {letter|"$"|"_"}{letter|"$"|"_"|digit}*
identifier = java_id | "`" java_id "`"
```

- einheitliche Schreibweise für alle Arten von Bezeichnern
- Unterscheidung von Klein-/Großschreibung (Case Sensitive)
- Escaping für Schlüsselwörter (Java-Interop.) Bsp: `foo.`is`()`

Anweisungen

```
{ var k = 42; println(5*k) }
```

- **optionales** Semikolon am Zeilenende
- Semikolon erforderlich zwischen Anweisungen in einer Zeile
- Anweisungsfolge definiert Scope: Variablen können gleichnamige in beliebigen äußeren Scopes überdecken

Bedingte Anweisung

```
if(a<b) { min=a } else { min=b }   ≡   min = if(a<b) a else b
```

- Bei einzelnen Anweisungen sind die {}-Klammern optional
- (fast) alle Konstrukte sind Anweisungen wie auch Ausdrücke

while-Schleife

```
var i = 0; while(i < n) { println("+"); i++; }
```

for-Schleife

```
for (i in 0 .. n-1) { println("+"); }
```

return-Anweisung

```
return n*42;     return "+";
```

Parameterübergabe

- standardmäßig mittels **call-by-value**

Kommentare

- Einzeilig: `// Kommentar`
- Mehrzeilig (verschachtelbar): `/* Kommentar … */`

(Un-)Veränderbare Var. und Collections

	Veränderbar	Unveränderbar

Deklaration Variable/ Konstante	`var i = 0 // Typ Int inferiert!`
	`var j:Int // Typangabe notw.`
	`j=42; i++; j-=2`
		`val i = 0 // Typ Int inferiert!`
		`val j:Int // Typangabe notw.`
		`j=42`

Listen	`// Typ: MutableList<Int>`
	`val n = mutableListOf(1, 2, 3)`
	• Liste `n` ist veränderbar, aber eine andere Liste kann nicht zugewiesen werden
	⇒ In dem Fall Verwendung von `var`
		`val readOnly: List<Int> = n`
		`val immutable = listOf(1, 2, 3)`
		• Durch `readOnly` ist die Liste `n` nicht veränderbar, aber durch `n` die zugrundeliegende Liste von `readOnly`. Liste `immutable` ist vollständig unveränderbar!

Sets	`val ms = hashSetOf(1,2)` `ms += 3 // ms.add(3)`	`val s = setOf(1,2)` `if(1 in s) println("in")`

Maps	`val map = mutableMapOf(2 to "y")`
	`map[1] = "x" // map.put(1,"x")`
	`println(map) // {1=x, 2=y}`
		`val readMap = mapOf("foo" to 1,`
		`"bar" to 2)`
		`println(readMap["foo"]) // "1"`

Primitive Datentypen

Int, Long, Short, Byte, Float, Double

- Dezimale, hexadezimale oder binäre Notation von Ganzzahlen (optional mit Underscores ⇝ Lesbarkeit)
- abgebildet auf primitive Datentypen der Zielplattform (z.B. JVM) (falls die Variable `null` sein kann ⟶ Objekte)
- keine implizite Konvertierung, sondern explizit z.B. mittels `.toFloat()` (*deutlicher*: ungleiche Werte)

String

- Zeichenkettenliterale mit
  - doppelten Anführungszeichen inklusive Parsen des Strings und Ersetzen von vorkommenden Ausdrücken und Escape-Folgen

```
val s = "abc"
println("$s.length is ${s.length}") // prints "abc.length is 3"
```

  - `"""` … `"""` für *rohe* Strings (über mehrere Zeilen)

Boolean

- Literale: `true` und `false` (Kleinschreibung beachten!)
- Operatoren: Konjunktion `&&`, Disjunktion `||`, Negation `!`

Datentypen: Arrays

Einträge	Paare (k, v) mit k ganzzahl. Schlüssel und zugeordneter Wert v
	Erzeugung von Arrays
Leeres Array	`val emptyArray = arrayOfNulls<String>(3)`
Liste von Werten	`// indiziert von 0 an:` `val monthName = arrayOf("", "Jan", "Feb", "Mar")`
Initialisierungs-funktion	`// Creates an Array<String> with values ["0", "1", "4"]` `val squares = Array(3, { i -> (i * i).toString() })`
Assoziatives Array	≈ Map, schon besprochen
	Zugriff
Explizite Zuweisung	`emptyArray[0] = "hello" // emptyArray.set(0,"hello")`
Lesezugriff	`println(squares[2]) // squares.get(2)`

Aufzählung aller Elemente	`val a = arrayOf(1,2,3)`
	`for(v in a) println(v)`
	`a.forEachIndexed { k, v -> println("$k => $v")}`

Smart Casts

- Ermittlung impliziter Typen durch statische Kontrollflussanalyse ⇝ **automatische Typumwandlungen**
  - unter Berücksichtigung von Datentypüberprüfungen mittels `is` und Typumwandlungen mittels `as` im Kontrollfluss

```
if (x is String)
  print(x.length) // x is automatically cast to String
```

```
if (x !is String) return
print(x.length) // x is automatically cast to String
```

```
x as String // unsafe cast: throws an exception if the cast is not possible
print(x.length) // x is automatically cast to String
```

```
val x: String? = y as? String // safe cast: x = null if the cast is not possible
```

Null Safety

- **Ziel:** Vermeidung von `NullPointerExceptions`*
- Variablen, die `null`-Werte halten können, müssen extra mit `?` nach der Datentypangabe gekennzeichnet werden:

```
var a: String = "abc" // no null allowed!
```

```
var b: String? = null
```

- Prüfen auf `null`:

```
val l = if (b != null) b.length else null
```

≡

```
val l = b?.length
```

  - Methodenaufruf bei `null`-able Typen kompilieren sonst nicht!

- Elvis Operator `?:` (Angabe eines Wertes bei `null`)

```
val l: Int = if (b != null) b.length else -1
```

≡

```
val l = b?.length ?: -1
```

- Nicht-Null-Annahme-Operator (möglichst nicht einsetzen!)

```
val l:String = b!!.length // throws NullPointerException if b is null!
```

* vgl. mit Tony Hoare, Billion-Dollar-Mistake als Folge der Einführung von null in ALGOL W im Jahre 1965

Definition von Funktionen

- ohne Rückgabewert (Bem.: `Unit` ist Singleton $\neq$ Java-Typ `void`)

```
fun pSum(a:Int, b:Int): Unit {
  println("$a + $b = ${a + b}")
  return Unit }
```

$\equiv$

```
fun pSum(a:Int, b:Int) {
  println("$a + $b = ${a + b}")
}
```

- mit Rückgabewert und Kurznotation für Funktionen mit **einem** Ausdruck (& inferierten Rückgabetyp)

```
fun sum(a:Int, b:Int): Int {
  return a + b
}
```

$\equiv$

```
fun sum(a:Int, b:Int) = a + b
```

- Benannte Argumente im Zusammenspiel mit Default-Werten

```
fun reformat(str:String, lowercase:Boolean = false,
             splitLongLines:Boolean = false,
             wordSeparator:String = " "):String {…}
val res = reformat("Hello world!", wordSeparator = '_')
```

Endrekursive Funktionen

- rekursiver Funktionsaufruf ist die letzte Aktion der Funktion
- automatische Transformation zu iterativer Funktion
- Bsp: Berechnung des Fixpunktes des Kosinus, für den $x = cos(x)$ gilt
  - mathematische Konstante ≈ 0.739
  - Laut Leonhard Euler: Fixpunktiteration $x_{n+1} = cos(x_n)$ konvergiert für jeden Startwert x_0 gegen die Lösung

```
tailrec fun findFixPoint(x:Double=1.0): Double =
                                if(x == cos(x)) x else findFixPoint(cos(x))
```

Transformation einer endrekursiven Funktion zu einer iterativen

```
tailrec fun findFixPoint(x:Double=1.0): Double =
                                    if(x == cos(x)) x else findFixPoint(cos(x))
```

Funktionen höherer Ordnung

- verwenden Funktionen als Parameter oder Rückgabe, Bsp.:

```
fun repeat(times:Int = 2, block:() -> Unit) {
    for(i in 1 .. times)
        block()
}
fun printHello(){
    println("hello")
}
repeat(3, ::printHello) // prints 3 times "hello"
fun repeatUntil(block:(Int) -> Boolean) {
    var i = 0
    do {
        val flag = block(i)
        i++
    } while(!flag)
}
fun printWorld(i:Int):Boolean {
    println("world")
    return i>=2
}
repeatUntil(::printWorld) // prints 3 times "world"
```

Lambdas

- in der Form `{` Parameter (mit optionalen Typ-Annotationen) `->` Rumpf `}`
- Letzter Ausdruck im Lambda-Rumpf ist Rückgabewert (falls vorhanden/nicht `Unit`)

```
val h = { -> println("h") }
```

```
val w = { i -> println("w"); ( i >= 2 ) }
```

- Lambdas mit einem Parameter: Paramter-Deklaration kann entfallen und Parameter erhält Default-Namen **`it`**

```
val w = { println("w"); (it >= 2) }
```

- Lambdas als Parameter einer Funktion verwendbar

```
repeatUntil( { println("w"); (it >= 2) } )
```

- Falls der letzte Parameter eines Funktionsaufrufes eines Lambda-Ausdruck erwartet, kann dieser aus der Klammer `( . . . )` herausgezogen werden, evtl. leere Klammer kann entfallen:

```
repeat(3){println("h")}
```

```
repeat{…}
```

```
repeatUntil{ println("w"); (it>=2) }
```

Klassen und Vererbung

```
open class Mammal {
  val legs:Int;
  constructor(plegs:Int){
    this.legs=plegs
  }
  open fun description() =
      legs+" legs-mammal!"
}
```

≈

```
// class with a primary constructor,
// which can be extended by a subclass
// and its function description()
// can be overriden!
open class Mammal(val legs:Int){
  open fun description() =
      legs+" legs-mammal!"
}
```

```
// without open class Dog cannot be extended any more by a subclass:
class Dog(val name:String): Mammal(4) {
  override fun description() = name+" has "+legs+" legs!"
}
fun main(args: Array<String>) {
  val waldi = Dog("Waldi")      // no new-keyword!
  println(waldi.description()) // prints "Waldi has 4 legs!"
}
```

- OOP-Sprachkonstrukte von Java haben zumeist Entsprechungen in Kotlin, z.B. abstrakte Klassen, Sichtbarkeit (z.B. `private`), `enum`, Schnittstellen, …

Hier Fokus auf einzelne Besonderheiten!

data-Klassen

```
data class User(val name: String, val age: Int)
val john = User(name = "John", age = 42)
```

- Kotlin-Compiler generiert für alle im Primärkonstruktor deklarierten Eigenschaften automatisch
  - `equals()` / `hashCode()`
  - `toString()` erzeugt Zeichenkette der Form: `"User(name=John, age=42)"`
  - `componentN()` Funktionen in der Reihenfolge der Deklarationen, z.B. für Destrukturierung: `val (name, age) = john`
  - `copy()` Funktion, hier:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name,age)
val copiedJohn = john.copy() // new object with properties copied
// new object with the same name, but different age:
val olderJohn = john.copy(age = 50)
```

Unterstützung des Singleton-Musters

- Verwendung, wenn
  - nur ein Objekt zu einer Klasse existieren darf und
  - ein einfacher Zugriff auf dieses Objekt benötigt wird
- Anwendungsbeispiele
  - global verwendetes Konfigurationsobjekt
  - Protokoll-Objekt für die Entgegennahme von Log-Einträgen und Ausgabe bzgl. registrierter Ausgabemöglichkeiten
    - in eine Datei
    - auf den Bildschirm

```
object Log {
    fun registerLogOutput(logOutput: LogOutput) { … }
    fun log(message: String) { … }
}
```

```
Log.registerLogOutput(…)
Log.log(…)
```

Sealed Classes

- definieren eine eingeschränkte Klassenhierarchie, wenn der Datentyp eines Wertes nur aus einer endlichen Menge gewählt sein darf

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr() // singleton!
```

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Extension Functions

- Erweiterung bestehender Klassen um zusätzliche Methoden (durch Definition außerhalb der Klassen)
  - wird in der JVM auf eine statische Methode abgebildet

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // calling swap like a normal method of the class MutableList!
```

Überladen von Operatoren & Infix-Notation

- Vordefinierte Menge an Operatoren (mit fester symbolischer Repräsentation (z.B. `*`) und fester entsprechender Präzedenz) kann überladen werden
  - Klassenmethode mit festen Namen muss mit **operator** gekennzeichnet werden
  - Häufige Verwendung in Standardbibliotheken für Programmierkomfort (z.B. Zugriff auf `Map`-Einträge via `[…]`)

```
data class Point(val x: Int, val y: Int)
operator fun Point.unaryMinus() = Point(-x, -y) // name 'unaryMinus' fixed!
val point = Point(10, 20)
println(-point)  // equivalent to println(point.unaryMinus())
```

- Infix-Notation: Durch **`infix`**-gekennzeichnete einparametrige Methoden (ohne Default-Wert)

```
infix fun Int.shl(x: Int): Int {…}
val result = 1 shl 2 // equivalent to 1.shl(2)
```

Tutorials und Spezifikation

- Offizielle Kotlin-Webseiten
  - Einstiegsseite: https://kotlinlang.org/
  - Doc: https://kotlinlang.org/docs/home.html
  - Blog: https://blog.jetbrains.com/kotlin/
  - Playground: https://play.kotlinlang.org/
  - Hands-On: https://play.kotlinlang.org/hands-on/overview
  - Learn-By-Example: https://play.kotlinlang.org/byExample/overview
  - Koans: https://play.kotlinlang.org/koans/overview
- Tutorials
  - https://www.guru99.com/kotlin-tutorial.html
  - Viele weitere Webseiten...

Zusammenfassung

- Kotlin als moderne Programmiersprache
  - Prägnanter Code ohne Einbußen der Lesbarkeit
  - Programmierkomfort & direkte Unterstützung gängiger Programmiermuster
    - Singleton
    - Domain Specific Language (DSL)
    - Container-Klassen (`data`-class)
    - Smart Casts durch statische Programmanalysen
  - Robustere Programme
    - Null-Safety
  - Performance (auch Programmierkomfort)