Vorlesung

Cloud- und Web-Technologien

(CS3140)

Fortgeschrittene Themen 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	

Domain-Specific Languages (DSLs)

- auf eine (beschränkte) Applikationsdomäne spezialisierte Computersprachen
  - Im Gegensatz zu General-Purpose Languages (GPL), die über verschiedene Domänen weit anwendbar sind
  - Beispiele für DSLs:
    - SQL
      - Datenbankanfragen
    - RegExp
      - Reguläre Ausdrücke zum Finden von Mustern in Text
    - Gradle
      - Beschreibung des Build-Prozesses von Computerprogrammen

Interne DSLs

- Spezifikation einer DSL in einer anderen General-Purpose Programming Language (z.B. in Kotlin)
  - mittels *Type-safe Groovy-style Builders* zum (semi-deklarativen) Beschreiben von hierarchischen Datenstrukturen

Bsp.: HTML

```
fun result() =
  html {
    head { title {+"XML"} }
    body {
      h1 {+"XML"}
      p  {+"Paragraph"}
      a(href =
       "http://kotlinlang.org")
       {+"Kotlin"}
    }
  }
```

Definition von User Interfaces (UI)
(Bsp.: TornadoFX)

```
form {
  fieldset(labelPosition =
           Orientation.VERTICAL) {
    field("Username") { textfield() }
    field("Password") { passwordfield() }
    button("Log in") {
      action {
        println("button press")
      }
    }
  }
}
```

Beispiel-DSL: persönliche Informationen

- DSL zur Angabe von persönlichen Informationen, Anwendung:

```
val person = person {
  name = "John"
  age = 25
  address {
    street = "Main Street"
    number = 42
    city = "London"
  }
}
```

- Ablage der Informationen in folgende Klassen:

```
data class Person(var name: String? = null,
                  var age: Int? = null,
                  var address: Address? = null)
data class Address(var street: String? = null,
                   var number: Int? = null,
                   var city: String? = null)
```

Sprachfeatures zur Entwicklung von DSLs 1/3

- Verwendung von Lambdas außerhalb von Methodenklammern
  - Nachteil: Verwendung von **it** notwendig

```
fun person(block: (Person)->Unit): Person {
  val p = Person()
  block(p)
  return p
}
```

Bsp. der Anwendung:

```
val person = person {
  it.name = "John"
  it.age = 25
}
```

Sprachfeatures zur Entwicklung von DSLs 2/3

- Lambdas mit Receivers
  - Angabe eines Scopes für übergebenes Lambda: Verwendung von Objektvariablen und -methoden (hier Scope `Person()`)

```kotlin
fun person(block: Person.() -> Unit): Person {
  val p = Person()
  p.block()
  return p
}
```

oder in kurz:

```kotlin
fun person(block: Person.() -> Unit): Person
  = Person().apply(block)
```

Bsp. der Anwendung:

```kotlin
val person = person {
  this.name = "John"
  this.age = 25
}
```

oder (besser):

```kotlin
val person = person {
  name = "John"
  age = 25
}
```

Sprachfeatures zur Entwicklung von DSLs 3/3

- Extension Functions
  - "Nachträgliches" Hinzufügen von Funktionen zu Klassen (außerhalb der Def. von Klassen)

```
fun Person.address(block: Address.() -> Unit) {
  address = Address().apply(block)
}
```

Bsp. der Anwendung:

```
val person = person {
    name = "John"
    age = 25
    address {
        street = "Main Street"
        number = 42
        city = "London"
    }
}
```

DSL: Beispiel 2: Gruppen in Veranstaltungen

```
sealed class Element(val name: String)
class Person(name:String): Element(name){
  override fun toString() = name
}
class Group(name:String): Element(name) {
  val children = arrayListOf<Element>()
  operator fun String.unaryPlus() {
    children.add(Person(this)) }
  fun group(name:String,
    init: Group.() -> Unit): Group {
    // init: fct. literal with receiver
    val g = Group(name); g.init();
    children.add(g); return g
  }
  override fun toString():String =
    name + children.toString()
}
fun lecture(name: String,
  init: Group.() -> Unit): Group {
  val g = Group(name); g.init();
  return g
}
```

```
fun main(args: Array<String>) {
  val wi = lecture(name="WebInfo"){
      +"Sven"
      group("Group1"){
          +"Patrick"
          group("BFF"){
           +"Peter"; +"Petra"
          }
      }
      group("Group2"){
        +"Heinz"; +"Herbert"
      }
  }
  println(wi)
}
```

Ausgabe: `WebInfo[Sven, Group1[Patrick, BFF[Peter, Petra]], Group2[Heinz, Herbert]]`

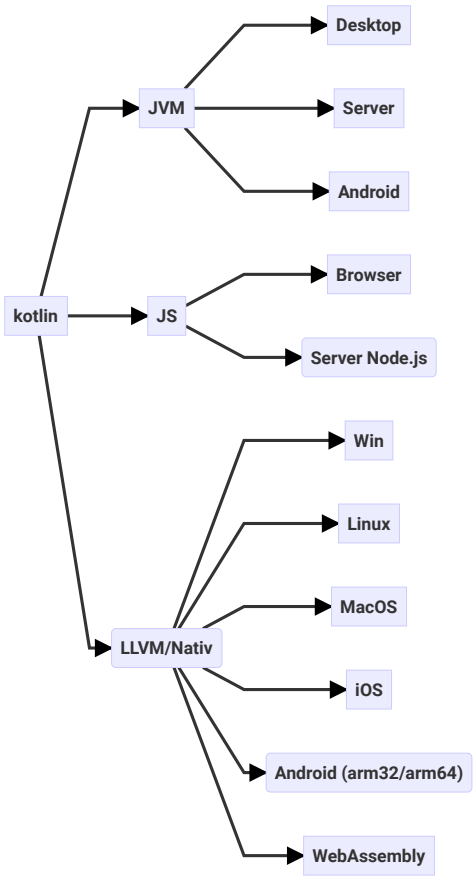
Schreiben einer DSL

- Entwerfe eine kleine DSL,
  die folgenden Beispielcode unterstützt:

```
val dependencies = DependencyHandler()
dependencies { //as good as dependencies.invoke(..)
    compile("Class1")
    testCompile("TestClass1")
}
```

Multi-Plattform Projekte

Targets:



- Aufteilung eines Projektes in plattform**un**abhängigen und plattformabhängigen Code
  - Plattformabhängiger Code kann teils auch in der Programmierprache der Zielplattform geschrieben sein (z.B. Java für JVM, JS für Web)
- Ermöglicht eine Codebasis für verschiedene Zielplattformen
  - Teilen von Code zwischen Server & (verschiedenen) Clients
- Vermeidung von Portierungsaufwand (in andere Programmiersprachen)

Multi-Plattform Projekte - Projektstruktur

- Common Module
  - Von bestimmten Plattformen unabhängiger Code mit Deklarationen für von Plattformen abhängigen Code ohne Implementation, Bsp.:

```
expect fun formatString(source: String, vararg args: Any): String
expect annotation class Test
```

- Platform Module
  - Implementation des im Common Module deklarierten plattformabhängigen Codes (und weiterer plattformabhängiger Code), Bsp.:

```
actual fun formatString(source: String, vararg args: Any) =
                                        String.format(source, args)
actual typealias Test = org.junit.Test
```

- Regular Module
  - hängt von einem Platform Module ab oder ein Platform Module hängt von diesem Modul ab

Alternativen der Multiplattform-Entwicklung

- Dieselbe Programmiersprache im Client & Server
  - Javascript im Server via `Node.js`
  - Server & Android-App mittels Java/JVM
- Web-Apps
  - Web-Anwendungen im Browser der verschiedenen Plattformen
- Hybride mobile Apps
  - Starten eine Art Browser im Fullscreen-Modus
  - für den Benutzer i.d.R nicht von nativer App unterscheidbar
  - Programmiert in HTML, CSS und JavaScript mit zusätzlichen APIs für native Smartphone-Apps
- Cross-Plattform-Apps
  - Bauen der Benutzeroberfläche meist mit den nativen APIs des jeweiligen Betriebssystems (keine Anzeige durch Webbrowser)
  - Teilen von bis zu 75% des Quellcodes* zwischen den verschiedenen Plattformen (ohne starke Einbußen bei der Performance)

Asynchroner Code - Motivation

- Bsp.: Android Applikation
  - Klick auf Button
  - Download von JSON Daten
  - Deserialisieren der Daten und Darstellung

```
fun fetchUserData(userId: String): String {
  // request user from network and return user data as JSON String
}
fun deserialize(userString: String): User {
  // deserialize and return deserialized user data
}
button.setOnClickListener { // sequential in UI thread
  val userString = fetchUserData("1")
  val user = deserialize(userString)
  showUserData(user)
}
```

Asynchroner Code - Motivation

- Bsp.: Android Applikation
  - Klick auf Button
  - Download von JSON Daten
  - Deserialisieren der Daten und Darstellung

```
fun fetchUserData(userId: String): String {
  // request user from network and return user data as JSON String
}
fun deserialize(userString: String): User {
  // deserialize and return deserialized user data
}
button.setOnClickListener { // sequential in UI thread
  val userString = fetchUserData("1")
  val user = deserialize(userString)
  showUserData(user)
}
```

- Problem: UI Thread lange blockiert
  => Bedienung der App stockt

Asynchroner Code - Lösungsansätze 1/2

- Lösungsansätze:
  - Callbacks
    - Refaktorieren von `fetchUserData` und `deserialize`: Ausführung im Background Thread und danach Aufruf von Callback-Funktionen

```
button.setOnClickListener { // welcome to the callback hell!
  fetchUserData("1") { userString ->
    deserialize(userString) { user ->
      showUserData(user)
    }
  }
}
```

  - Reactive Programming
    - Aneinandereihung anstatt Verschachtelung

```
button.setOnClickListener {
  fetchUserData("1")
  .flatMap { userString -> deserialize(userString) }
  .subscribe { user -> showUserData(user) }
}
```

Asynchroner Code - Lösungsansätze 2/2

- Lösungsansätze:
  - Coroutines
    - asynchrone Programmierung in einem sequentiellen Stil

```
button.setOnClickListener {
  launch(UI){
    val userString = fetchUserString("1").await()
    val user = deserializeUser(userString).await()
    showUserData(user)
  }
}
```

Coroutines

- Einige APIs initiieren langanhaltende (& den Thread des Aufrufers blockierende) Berechnungen, z.B.:
  - Transfer zum/vom Netzwerk/Externspeicher
  - CPU-intensive Arbeit
- Standardbibliotheken verwenden Coroutines zur Vereinfachung der asynchronen Programmierung
  - **Ziel:** Asynchrone Programmierung in einem sequentiellen Stil
    - Intern Verpackung des Benutzercodes in Callbacks
    - Anmeldung für die Abarbeitung relevanter Ereignisse
    - Programmausführung auf verschiedenen Threads möglich

Coroutines

- ausgefeilte Techniken, um Code nebenläufig zu strukturieren, der dann auch parallel ausgeführt werden kann
- kostengünstiger als Threads
  - Softwareseitiges Umschalten des Ausführens zwischen verschiedenen Coroutinen an vom Entwickler zu vermerkenden Stellen
  - I.d.R. ausgeführt durch Thread Pools, aber auch in einem Thread ausführbar
  - Tausende von Coroutinen (mit leichtgewichtigen Berechnungen) ohne Performanzprobleme ausführbar, aber nicht Tausende von nativen Threads

Coroutines - Interne Arbeitsweise

- komplett durch Kompiliertechnik mittels Codetransformation realisiert (ohne Unterstützung durch VM oder OS)
- Transformation jeder "suspending" Funktion zu einer Zustandsmachine *(Continuation-passing style)*
  - Zustände = Aufrufe der "suspending" Funktionen
  - Gleich nach dem Unterbrechen: nächster Zustand `z` wird in ein Feld einer Compiler-generierten Klasse zusammen mit relevanten lokalen Variablen abgelegt
  - Beim Wiederaufnehmen der Berechnungen: Lokale Variablen werden wiederhergestellt und die Zustandsmaschine wird mit nächsten Zustand `z` fortgesetzt
  - eine unterbrochene Coroutine kann als Objekt (mit gespeicherten Zustand und lokale Variablen) vom Typ `Continuation` abgespeichert und herumgereicht werden
  - "suspending" Funktionen haben intern einen Extraparamater vom Typ `Continuation`

Coroutines - Bsp. der internen Arbeitsweise

```
var x = 0
while (x < 10) {
  x += nextX().await()
}
```

```
class Generated extends CoroutineImpl<...>
  implements Continuation<Object> {
    int state = 0
    int x // local variables of the coroutine
    void resume(Object data) {
        if (state == 0) goto L0
        if (state == 1) goto L1
        else throw IllegalStateException()
    L0: x = 0
    LOOP: if (x >= 10) goto END
          state = 1
          // 'this' is passed as a continuation
          data = nextX().await(this)
          if (data == COROUTINE_SUSPENDED) return
    L1: // external code has resumed this coroutine
        // passing the result of .await() as data
        x += ((Integer) data).intValue()
        goto LOOP
    END: state = -1 // No more steps are allowed
         return
    }
}
```

```
suspend fun <T>
 CompletableFuture<T>
 .await(): T
```

```
suspend fun <T>
 CompletableFuture<T>
 .await(continuation:
   Continuation<T>): Any?
```

gibt das Resultat oder `COROUTINE_SUSPENDED` zurück

Coroutine Builders

- `launch`
  - starte und vergesse (kann auch abgebrochen werden)
- `async`
  - gibt Promise (deutsch Versprechen) zurück
- `runBlocking`
  - blockiert aktuellen Thread, ist als Brücke zwischen regulär blockierenden Code und unterbrechenden (suspending) Funktionen entwickelt worden (anwendbar in main-Funktionen und Tests)
- ...

Coroutine Builders - Beispiel

```
suspend fun doSomething():Int {
  for(i in 0..100_000_000) print(".") // just simulate work
  return 42
}
fun main(args: Array<String>) =
  // use runBlocking as wrapper as join() may suspend...
  runBlocking {
    // GlobalScope is used to spawn a launch coroutine
    // limited to the application lifecycle itself
    val job=GlobalScope.launch{ // launch(CommonPool) for common pool of threads
      val result = doSomething() // call a suspending function
      print("the launched result is $result")
    }
    val a = async {
      println("I'm computing a piece of the answer")
      for(i in 0..100_000_000) print(".") // just simulate work
      6 // the result of this async-expression
    }
    println("The async-answer is ${a.await()}") // wait for the async-result
    job.join() // wait for the launched job
  }
```

Coroutines - Generatoren und `yield`

Sequentieller Stil mit Iterator-Generator

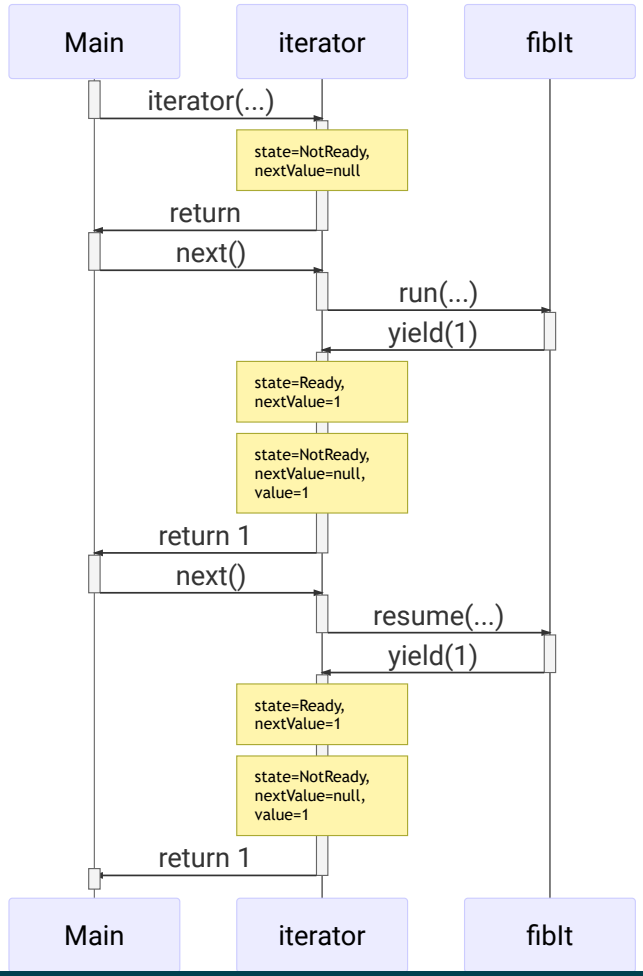
```
val fibIt = iterator {
  var a = 0
  var b = 1
  yield(1)
  while (true) {
    yield(a + b)
    val tmp = a + b
    a = b
    b = tmp
  }
}
// Print the first eight
// Fibonacci numbers
println(fibIt.asSequence().take(8)
        .toList())
```

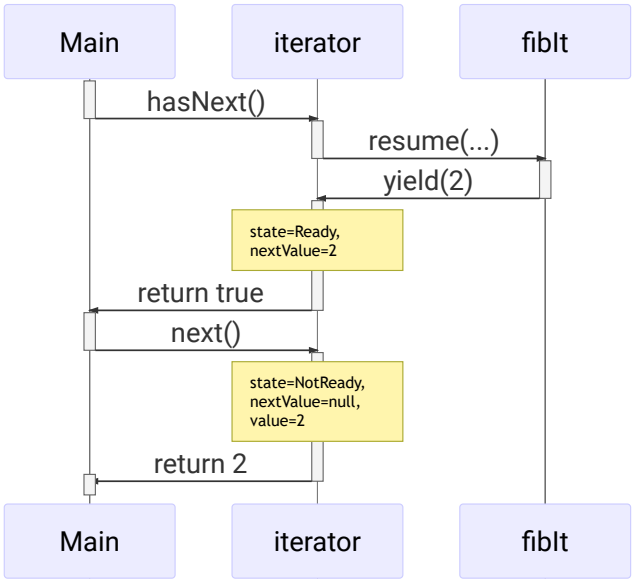
ergibt: `[1, 1, 2, 3, 5, 8, 13, 21]`

Zum Vergleich: Iterator-Klasse

```
class FibIt:Iterator<Int>{
  var a = 0
  var b = 1
  override operator
    fun hasNext(): Boolean = true
  override operator fun next():Int {
    val tmp = a + b
    a = b
    b = tmp
    return a
  }
}
// Print the first eight
// Fibonacci numbers
println(FibIt().asSequence()
        .take(8).toList())
```

Generatoren - Vereinfachter Ablauf





Wiederholung: 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
```

```
// x is automatically cast to String on the right-hand side of `||`
if (x !is String || x.length == 0) return
// x is automatically cast to String on the right-hand side of `&&`
if (x is String && x.length > 0) {
  print(x.length) // x is automatically cast to String
}
```

Contracts 1/2

- Kotlin 1.3: für Fkt. in Stdlib & experimentell für eigene Fkt.
- Mit Contracts beschreibt eine Fkt. explizit sein Verhalten (für den Compiler)
  - Verbesserung der Smart Casts-Analyse durch Beschreibung des Funktionsergebnisses in Abhängigkeit mit den übergebenen Argumentwerten

```
fun require(condition: Boolean) {
  // This is a syntax form, which tells compiler:
  // "if this function returns successfully, then passed 'condition' is true"
  contract { returns() implies condition }
  if (!condition) throw IllegalArgumentException(...)
}
fun foo(s: String?) {
  // s is smartcasted to 'String' here,
  // because otherwise 'require' would have thrown an exception
  require(s is String)
}
```

Contracts 2/2

- Verbesserung der Variableninitialisierungsanalyse in Gegenwart von Funktion höherer Ordnung

```kotlin
fun synchronize(lock: Any?, block: () -> Unit) {
  // It tells compiler:
  // "This function will invoke 'block' here and now, and exactly one time"
  contract { callsInPlace(block, EXACTLY_ONCE) }
}
fun foo() {
  val x: Int
  synchronize(lock) {
    x = 42 // Compiler knows that lambda passed to 'synchronize' is called
           // exactly once, so no reassignment is reported
  }
  // Compiler knows that lambda will be definitely called,
  // performing initialization, so 'x' is considered to be initialized here
  println(x)
}
```

Zusammenfassung

- Kotlin als moderne Programmiersprache
  - Domain-Specific Languages (DSL)
  - Performance (auch Programmierkomfort)
    - Coroutines
      - leichtgewichtiger als Threads
      - asynchrone Programmierung im sequentiellen Stil
      - Iterator-Generatoren
  - Multiplattform-Entwicklung
  - Contracts