Vorlesung

# Webbasierte Informationssysteme

(CS4130)

## Multiplattform-Entwicklung mit Kotlin

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

# Chronologische Übersicht über die Themen

# 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
  - $\rightsquigarrow$ 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!`<br>`var j:Int // Typangabe notw.`<br>`j=42; i++; j-=2` | `val i = 0 // Typ Int inferiert!`<br>`val j:Int // Typangabe notw.`<br>`j=42` |
| **Listen** | `// Typ: MutableList<Int>`<br>`val n = mutableListOf(1, 2, 3)`<br>• Liste `n` ist veränderbar, aber eine andere Liste kann nicht zugewiesen werden<br>$\Rightarrow$ In dem Fall Verwendung von **var** | `val readOnly: List<Int> = n`<br>`val immutable = listOf(1, 2, 3)`<br>• 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)`<br>`ms += 3 // ms.add(3)` | `val s = setOf(1,2)`<br>`if(1 in s) println("in")` |
| **Maps** | `val map = mutableMapOf(2 to "y")`<br>`map[1] = "x" // map.put(1,"x")`<br>`println(map) // {1=x, 2=y}` | `val readMap = mapOf("foo" to 1,`<br>`"bar" to 2)`<br>`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:`<br>`val monthName = arrayOf("", "Jan", "Feb", "Mar")` |
| **Initialisierungs-funktion** | `// Creates an Array<String> with values ["0", "1", "4"]`<br>`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)`<br>`for(v in a) println(v)`<br>`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!
```

# 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

```kotlin
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:

```kotlin
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)
```

# Domain-Specific Languages (DSLs)

```
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
}
```

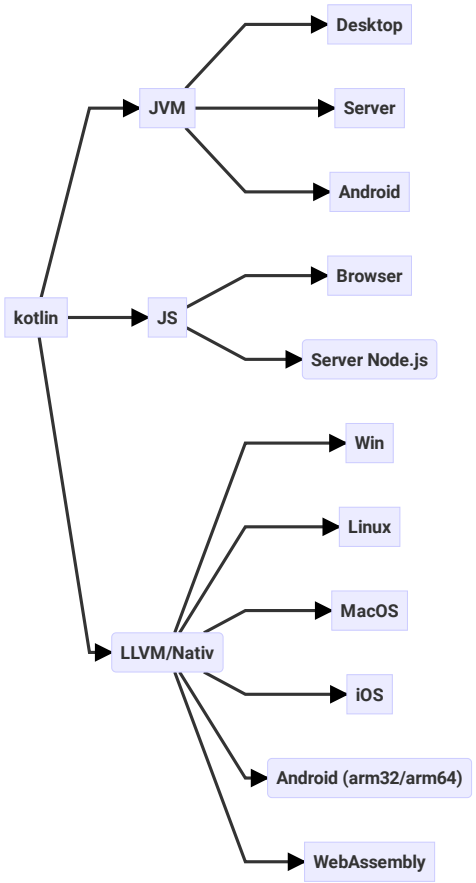
mittels *Type-safe Groovy-style Builders* zum (semi-deklarativen) Beschreiben von hierarchischen Datenstrukturen

```
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]]`

# 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)

# 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
- kostengünstiger als 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

# Coroutines - Generatoren und `yield`

## Sequentieller Stil mit Iterator-Generator

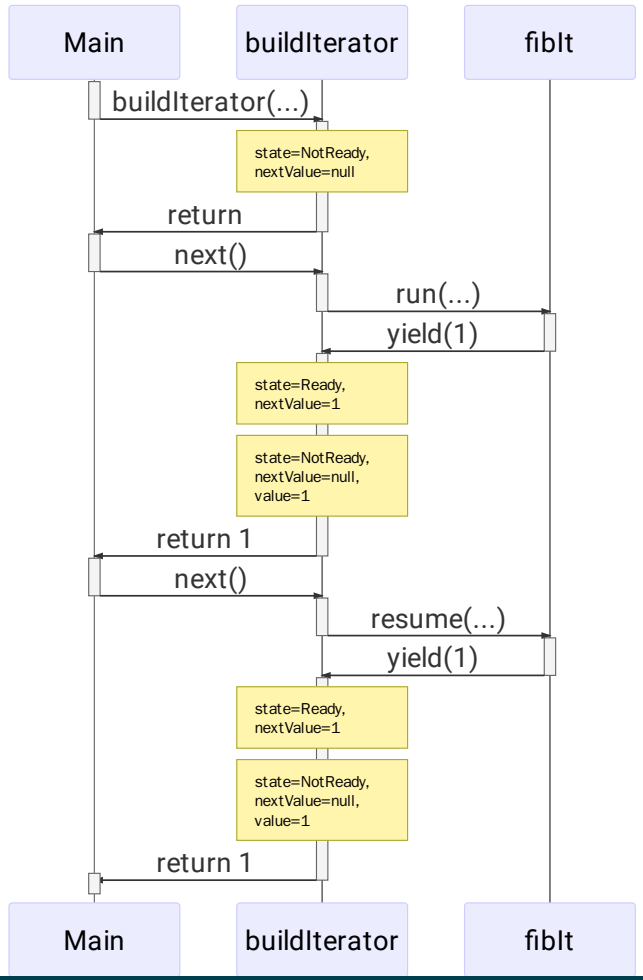
```
val fibIt = buildIterator {
  var a = 0
  var b = 1
  yield(1)
  while (true) {
    yield(a + b)
    val tmp = a + b
    a = b
    b = tmp
  }
}
// Print the first eight Fibonacci
// numbers
println(fibIt.asSequence().take(8)
        .toList())
```

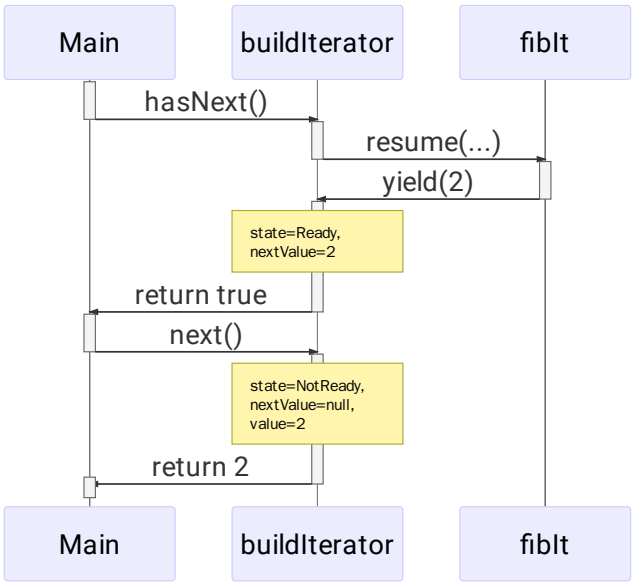
ergibt: `[1, 1, 2, 3, 5, 8, 13, 21]`

## 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



# 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)
    - Coroutines
      - leichtgewichtiger als Threads
      - asynchrone Programmierung im sequentiellen Stil
  - Multiplattform-Entwicklung