Android Programmierung mit Kotlin

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- http://mwilhelm.hs-harz.de
- Raum 2.202
- Tel. 03943 / 659 338

Gliederung

Überblick:

- **Sprache**
  - elementare Datentypen
  - Variablen und Kontrollstrukturen
  - Arrays, Funktionen, Rekursionen
  - Objekte, etc.

Links

- https://www.tutorialspoint.com/kotlin/index.htm
- https://kotlinlang.org/docs
- https://www.w3schools.com/kotlin/index.php

Eigenschaften:

- **Easy Language** − Kotlin supports object-oriented and functional constructs and very easy to learn. The syntax is pretty much similar to Java, hence for any Java programmer it is very easy to remember any Kotlin Syntax.

- **Very Concise** − Kotlin is based on Java Virtual Machine (JVM) and it is a functional language. Thus, it reduce lots of boiler plate code used in other programming languages.

- **Runtime and Performance** − Kotlin gives a better performance and small runtime for any application.

- **Interoperability** − Kotlin is mature enough to build an interoperable application in a less complex manner.

- **Brand New** − Kotlin is a brand new language that gives developers a fresh start. It is not a replacement of Java, though it is developed over JVM. Kotlin has been accepted as the first official language of Android Application Development. Kotlin can also be defined as - **Kotlin = Java + Extra updated new features**.

Kommentare:

- **Allgemeine Kommentare**
  - // nur eine Zeile
  - /* Bereich Anfang
  - */ Bereich Ende

Semikolon;

- **Es braucht kein Semikolon**

main:

Java

```
void main() {
  var string: String  = "Hello, World!"
  println("$string")
}
```

Kotlin:

```
fun main(args: Array<String>){
   println("Hello, world!")
}
```

Keywords

as	as?	break	class	continue	do
else	false	for	fun	if	in
!in	interface	is	!is	null	object
package	return	super	this	throw	true
try	typealias	typeof	**val**	**var**	when
while	by	catch	delegate	dynamic	field
file	finally	get	import	**init**	param
property	receiver	set	setparam	**value**	where

Java: Variablen: A-ZÖÄÜß#%

- `val alter = 12` Konstante
- `var alter = 12` Variable, Mutable
- `val MAX = 1000` Konstante
- `val name = "Anton"` Konstante
- `println("Name = $name")`
- `println("Alter = $alter")`
- `var s:String="abc"`
- `var s="abc"`
  automatisch ein String
- `val MAX:Int=100`
- `val MAX=100`
- var n=100
  // kann verändert werden

Kotlin: Datentypen

- Byte, 8 bit
- Short , 16 bit
- Int , 32 bit
- Long , 64 bit
- Float , 32 bit
- Double , 64 bit
- Boolean
- Char
- String
- Escape

Ein normaler Datentyp hat **KEINE** null-Werte!

- val boolNull: Boolean**?** = null
- var alterNull : Int? = 12
- var alterNull : Int? = null

String-Datentyp

- String: index von 0 bis n-1
- val s="abc"
- s.length integer 3
- s.count() "3"
- s.lastindex 2
- s.uppercase()
- s.lowercase()
- t="def"
- u=s.plus(t) abcdef
- u.drop(2) cdef
- u.dropLaste(2) abcd
- u.indexOf("cd") 2
  -1 = nicht vorhanden
- u.indexOf("cd", startindex, ignoreCase) 2
- u.compareTo(s) 0=equals 1=ungleich

String-Datentyp

- **Konvertierung**
  - toByte()
  - toShort()
  - toInt()
  - toLong()
  - toFloat()
  - toDouble()
  - toChar()

```
val x: Int = 100
val y: Long = x  // Not valid assignment
val y: Long = x.toLong()
```


Operatoren:

- +
- -
- *
- /
- %
- ++x
- x++
- --x
- x--
- !

- +=
- -=
- *=
- /=
- %=

Vergleiche:

- >
- <
- >=
- <=
- ==
- !=

Bool'sche Operatoren:

- &&
- ||
- !

if-Anweisungen: Immer mit den Klammern (), kein elif

```
int  a = 5
int  b = 7
if (a > b) {
        print("a ist größer als b")
}
else {
        print("a ist kleiner als b")
}

if (a > b)
        print("a ist größer als b")
else
        print("b ist kleiner als b")
```

val result = if (age>18) "Adult" else "Minor "

when-Anweisungen

```
val day = 2
when (day) {
  1 -> println("Monday")
  2 -> println("Tuesday")
  3 -> println("Wednesday")
  4 -> println("Thursday")
  5 -> println("Friday")
  6 -> println("Saturday")
  7 -> println("Sunday")
  else -> println("Invalid day.")
}
```

when-Anweisungen

```
val result = when (day) {
  1 -> "Monday"
  2 -> "Tuesday"
  3 -> "Wednesday"
  4 -> "Thursday"
  5 -> "Friday"
  6 -> "Saturday"
  7 -> "Sunday"
  else -> "Invalid day."
 }
```


when-Kombinierte Anweisungen

```
val day = 2
when (day) {
  1, 2, 3, 4, 5 -> {
    println("Weekday")
    println("Wann ist Wochenende?")
  }
  else -> println("Weekend")
}
```

when Range in when Conditions

```
val day = 2
when (day) {
  in 1..5 -> println("Weekday")
  else -> println("Weekend")
}
```


when Expression in when Conditions

```
val x = 20
val y = 10
val z = 10
when (x) {
  (y+z) -> print("y + z = x = $x")
  else -> print("Condition is not satisfied")
}
```

when Abfrage nach einem Type

```
var num: Any = "GeeksforGeeks"
when(num){
is Int -> println("It is an Integer")
is String -> println("It is a String")
is Double -> println("It is a Double")
}
```


when Funktionen

```
var num = 8
when{
isOdd(num) ->println("Odd")
isEven(num) -> println("Even")
else -> println("Neither even nor odd")
}

when  {
                                // ohne Bedingung
isOdd(num) ->println("Odd")
!isOdd(num) -> println("Even")
else -> println("Neither even nor odd")
}
```

While-Schleife

Aufbau:

1) Initialisierung

2) while (Bedingung/en) {

3) Schleifenrumpf / Anweisungen

4) }

Bedingung zeigt an, wie oft die Schleife durchlaufen wird

Do-While-Schleife

Aufbau:

1) Initialisierung

2) do {

3) Schleifenrumpf / Anweisungen

4) }

5) while (Bedingung/en);

Bedingung zeigt an, wie oft die Schleife durchlaufen wird

For-Schleife

```
for ( num in 1.rangeTo(4) ) {
  println(num)                    1,2,3,4
}
for ( num in 1..4 ) {
  println(num)                    1,2,3,4
}
for ( num in 1.. until 4 ) {                <
  println(num)                    1,2,3
}
for ( num in 4 downTo 1 ) {
  println(num)                    4,3,2,1
}
for ( num in 1..10 step 2 ) {
  println(num)                    1,3,5,7,9
}
```


Range-Utility Funktionen

```
val a = 1..10
println(a.min())                  1
println(a.max())                  10
println(a.sum())                  55
println(a.average())              5.5
println(a.count())                10
```

Arrays

- val fruits = arrayOf("Apple", "Mango", "Banana", "Orange")
- val fruits = arrayOf<String>("Apple", "Mango", "Banana", "Orange")
- val num = intArrayOf(1, 2, 3, 4)
- Parameter in einer Funktion:
- Feld:Array<Float>

- byteArrayOf()
- shortArrayOf()
- intArrayOf
- longArrayOf()
- charArrayOf()

Arrays: Zugriff

- s= fruits[3]
- fruits[3]="Kirschen"
- s= fruits.get(3)
- fruits.set(3, "Kirschen")

- Länge des Feldes: size

Arrays: Zugriff mit Schleifen

- val fruits = arrayOf<String>("Apple", "Mango", "Banana", "Orange")
- for(item in fruits){
- println(item)
- }
- if ("Mango" in fruits){
- println("Mango exists in fruits")
- }else{
- println("Mango does not exist in fruits")
- }

Arrays: Parameter

```
// Deklarieren
val templiste = arrayOf<String>("Celsius",
"Kelvin","Fahrenheit")

// Funktionsaufruf
functionString(templiste)

// Funktionsdeklaration
fun functionString(liste:Array<String>) {  }
```

Arrays: distinct

```
val fruits = arrayOf<String>("Apple", "Mango",
"Banana", "Orange", "Apple")
val distinct = fruits.distinct()
for( item in distinct ){
   println( item )
}
```

Arrays: empty

```
fruits.isEmpty()
```


Listen (Mutable und Immutable)

- ArrayList<T>()
- arrayListOf()
- mutableListOf()

- **Parameter:**
  - vorlesung:MutableList<Student>

- val numbers = mutableListOf("one", "two", "three", "four")
- numbers.add("five")

Listen: Methoden

- size()
- if (a in theList) {}
- if (theList.contain(a)) {}
- isEmpty()
- indexOf(a) // von 0 bis n-1
- get(index)
- list3 = list1+list2 // Operator-Overloading
- list3 = list1-list2 // Operator-Overloading
- list3 = list1.slice(2..4) // 2,3,4
- list3 = list1.filterNotNull() // kopieren
- list3 = list1.filter(a>10) // kopieren
- list1.drop(1) // ein Element gelöscht
- list1.drop(3) // die ersten DREI Elemente werden gelöscht

Listen: Methoden

- listArray = list1.groupBy(a %3) // listArray enthält DREI TeilListen
- listeKeyMap = list1.map(a/10) // list enthält die Keys für eine HashMap
- list3 = list1.chunked(3)
  - list3 enthält TeilListen mit jeweils DREI Elemente (beim Letzten?)
  - list3 = list1.windowed(size)
  - list3 enthält TeilListen mit jeweils DREI Elemente (beim Letzten?)
- val list1 = listOf(10, 12, 30, 31, 40, 9, -3, 0)
  - val list3 = list1.windowed(3)
  - [[10, 12, 30], [12, 30, 31], [30, 31, 40], [31, 40, 9], [40, 9, -3], [9, -3, 0]]

Listen: Methoden

- list3 = list1.windowed(size,step)
  - list3 enthält TeilListen mit jeweils DREI Elemente (beim Letzten?)
  - val list1 = listOf(10, 12, 30, 31, 40, 9, -3, 0)
  - val list3 = list1.windowed(3,3) // size, step=1
  - [[10, 12, 30], [31, 40, 9]]

- **Methoden:**
  - add(obj)
  - add(index,obj)
  - remove(obj)

Java: Enumeration:

- Enumeration sind besser als mehrere Konstante:
  - enum class Direction {
  - NORTH, SOUTH, WEST, EAST
  - }

- **Enumeration**

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

  - Enum sinnvoll auch in einer switch-Anweisung

Funktionen

```
fun myfunc() {
}
fun myfunc():Unit {
}
fun printSum(a:Int, b:Int){
  println(a + b)
}
fun add(a:Int, b:Int):Int{
  sum = a + b
  return sum
}
```


Funktionen

```
fun factorial(x:Int):Int{
  val result:Int
  if( x <= 1){
    result = x
  }else{
    result = x*factorial(x-1)
  }
  return result
}
```

Klassen:

```
class ClassName { // Class Header
  // private, protected, public, internal (innerhalb des Moduls)
   var a:Int;
  // Variables or data members
  // Member functions or Methods
  constructor (a:Int) {
   this.a=a
    }
  constructor ():this(42)
    // !!
}
```


Klassen: Constructor wird per Parameter erzeugt!, ohne Zuweisung

```
class Person constructor(val firstName: String, val age: Int=20) {
  init {
    this.firstName = firstName
    this.age = _age
  }
}

class Student ( var lastname:String="???", var mnr:Int=123)
val std = Student("Annemarie",42)
val s:String="std: lastname: "+std2a.lastname+ "  mnr: "+std2a.mnr

// durch data nun mit equals, hash und toString
data class Student ( var lastname:String="???", var mnr:Int=123)
val std = Student()
val s:String="std: "+std3.toString()    // Student(lastname="???", mnr=123)
val std2 = std.copy()
val s:String="std=std2 "+std.equals(std2)                    // true
```

Klassen, Keywords

- private, protected, public, internal
- open class
  - damit ableitbar, aber ohne toString()
- open variable
- data class
  - data class können nicht abgeleitet werden
  - können aber eine Oberklasse haben
  - können erweitert werden durch Interface
  - es werden automatisch toString, equals, hashValue erzeugt
- sealed class
- override
- inner class
  - Zugriff auf die Variablen, Methoden der Hauptklasse à la Java
- Nested class, innerClass
  - Ist static (default)
  - Kein Zugriff auf die Variablen, Methoden der Hauptklasse à la Java

Klassen, Keywords

- **Properties à la C#**
  - class Person () {
    - var lastname: String=""
    - get() {
      - **return field**
    - }
    - set(value) {
      - if (!value.isEmpty()) {
        field = value
      - }
    - }
  - **constructor**(lastname:String):this() {
    - this.lastname = lastname
    - }
- }

Klassen, Keywords

- : Superclass
  - : statt extends
- **Ohne new**
- Default Parameter
- super à Java
- interface kann echte Funktionen haben
- **„Multivererbung“**

Klassen, Beispiele

```
class OuterClass{
  // Members of Outer Class
  class NestedClass{
    // Members of Nested Class
  }
}

val obj = Outer.Nested()

  print(obj.foo())
```

Klassen, Beispiele

```
class OuterClass{
  // Members of Outer Class
  class inner InnerClass{
    // Members of Inner Class
  }
}
```

```
val outerObj = OuterClass()
val innerObj = outerObj.InnerClass()
```

Oder

Klassen, Beispiele

```
open class Person1 (var lastname:String, var firstname:String="-")

class StudentPerson1 (lastname:String, firstname:String="-", var
mnr:Int):Person1(lastname,firstname)

fun bn9_Click(view: View) {
   val p1 = Person1("Bates","Kim")
   var s:String = "p1: lastname: "+p1.lastname

val std1 = StudentPerson1("Bates","Kim",123)
   s=s+"\n"+ "std1: lastname: "+std1.lastname+ " firstname:
"+std1.firstname+"  mnr: "+std1.mnr
   show("std1: "+s)

}
```

Klassen, Beispiele

```
open class Person2 () {
   var lastname: String=""
   get() {
      return field
   }
   set(value) {
      if (!value.isEmpty()) {
         field=value
      }
   }
   var firstname: String=""
      get() {
         return field
      }
      set(value) {
         if (!value.isEmpty()) {
            field=value
         }
      }
```


Klassen, Beispiele

```
   constructor(lastname:String, firstname:String):this() {
      this.lastname = lastname
      this.firstname = firstname
   }
}
```

Klassen, Beispiele

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // public by default

    protected class Nested {
        public val e: Int = 5
    }
}
class Subclass : Outer() {
    // a is not visible,  b, c and d are visible,  Nested and e are visible
    override val b = 5   // 'b' is protected
    override val c = 7   // 'c' is internal
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```