

(Full Stack )

Kotlin

# Who's who

- Christian Gauch
- Céline Heldner
- Lukas Lädach



# Was kann man vom Kurs erwarten?

- Kurs ist nur ein Überblick
- Soll Interesse wecken und als Startpunkt fungieren
- Gewisse Themen werden nur angeschnitten
- Übungen sind timeboxed

# Agenda

- Arbeitszeiten, Pausen
  - 09:00 - 10:00
  - Pause (15min)
  - 10:15 - 12:00
  - Mittagspause (1h)
  - 13:00 - 15:00
  - Pause (15min)
  - 15:15 - 17:00
- Parkplatz für unbeantwortete Fragen

# Inhalt

- Kotlin
- Vorstellung Java App + Kotlin Conversion
- Kodein
- Room
- Testing

Day 1

- 
- Databinding
  - Functional Programming
  - RxKotlin
  - Couroutines
  - StandardLibs + Extensions

Day 2

# Kotlin - Java

Die Unterschiede

- Kotlin ist eine russische Halbinsel vor St. Petersburg
- ... und eine Programmiersprache
- Entwickelt von JetBrains
- Wurde 2011 erstmals der Öffentlichkeit vorgestellt ( Swift 2013)
  - Nix copycat...😊
- Februar 2016 wurde Version 1.0 veröffentlicht.

# Warum Kotlin ?

- Kotlin ist verständlicher (selbstsprechende Syntax, eindeutige Kommandos)
- Weniger Code -> weniger zu testen
- Default Arguments, Lambda Expression, Higher Order Functions,...
- Kein Semikolon
- Null Safety!



# Generelle Struktur

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

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

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



Visibility Modifier  
Ist redundant



Wenn nur ein Ausdruck -> One Liner mit  
Impliziten Rückgabewert

# Rückgabewert

```
fun printSum(a: Int, b: Int): Unit {  
    println(a + b)  
}
```

```
fun printSum(a: Int, b: Int){  
    println(a + b)  
}
```



In Kotlin hat jede Funktion einen Rückgabewert.  
**Unit** kann aber weggelassen werden

# Rückgabewert- Nothing

Nothing ist ein spezieller Typ in Kotlin, für den es kein Gegenstück in Java gibt. Der Rückgabewert Nothing kennzeichnet eine Methode welche eine Exception zurückgibt.

Kann hilfreich sein wenn man das Error Verhalten in Unit Tests verifizieren möchte.

```
fun throwException(): Nothing {  
    throw Exception("Exception message")  
}
```

# Mutable vs Immutable

```
var x : Int = 1      var x = 1  
val x : Int = 1      val x = 1  
lateinit var repository: Repository
```

mutable

immutable

# String templates

```
println("$a")
```

Einfache Ausdrücke, können mit dem „\$“ Zeichen im String referenziert werden

```
println("a + a = ${a + a}")
```

Komplexere Ausdrücke müssen mit „\$“ und **Curley Braces** umschlossen werden

# Null Safety

In Kotlin wird zwischen Referenzen unterschieden welche Null sein können, und jenen welche es nicht können.

```
var a = 1  
a = null
```

Dies führt zu einem Kompilierfehler, da **a** nicht null sein kann.

```
var a : Int? = 1  
a = null
```

Kein Fehler, diese Variable wurde als „nullable“ gekennzeichnet

**Int**

**Klasse**

**Int?**

**Typ**

# Null Safety – How to deal with it?

Variante 1: Explizit auf „NULL“ prüfen

```
val a : Int? = 2
```

```
val c = if(a != null) a else -1
```

In Kotlin ist **if** ein Ausdruck der einen Wert zurückgeben kann.  
Daher gibt es auch keinen **Conditional Ternary Operator** „a ? b : c“

```
val d = a ?: -1
```

Aber es gibt den **Elvis Operator** „?:“

# Null Safety – How to deal with it?

## Variante 2: Safe Call mittels Safe Call Operator (I)

```
val a : String? = "Hello"  
val b = a.length
```

Ergibt einen Kompilierfehler, da **a** null sein kann.

```
val a : String? = "Hello"  
val b = a?.length
```

„?.“ Ist der Safe Call Operator, die Operation „length“ wird nur dann ausgeführt wenn „a“ nicht null ist. (Anmerkung: Die Variable **b** ist somit auch nullable)

Der Safe Call Operator kann auch aneinander gereiht werden:

„House?.room?.door?.open“

# Null Safety – How to deal with it?

## Variante 2: Safe Call mittels Safe Call Operator (II)

Um nun Operationen nur auf Variablen auszuführen welche nicht null sind, kann der **Safe Call Operator** zusammen mit dem „**let**“ Operator verwendet werden.

```
val list = listOf("1","2","3","4")
for(number in list){
    number?.let {
        println(number)
    }
}
```

„println“ wird nur dann ausgeführt wenn „number“ nicht null ist.



# Null Safety – How to deal with it?

## Variante 3: Für NPE Liebhaber der !! Operator

Der !! Operator gibt entweder einen „Non-Null“ Wert zurück oder wirft eine NPE

```
var b : String? = null  
println("${b!!.length}")
```

Hier wird ein NPE geworfen

```
b?.let {  
    b!!.length  
}
```

Der sichere Weg, zuerst mit dem “?.” Operator auf non-null prüfen und anschließend den Wert entpacken. Für Fälle wo Kotlin auf den Typ der Variable schließen kann und es keine mutable\* Variable ist, kann der “!!” Operator innerhalb de “let-Blocks” weggelassen werden. (Smart Cast)

\*hängt davon ab ob zwischen dem ?.let und dem entpacken (!!) die Variable geändert werden kann    basic\_syntax\_03

# Null Safety – How to deal with it?

## Variante 4: Do it yourself

```
fun <T> T?.or(default: T): T = if (this == null) default else this
```

```
var variable: String? = "Hello"  
println(variable?.length.or(-1))
```

Eine kleine Helper Extension\* „or“ welche auf null prüft, oder einen default Wert zurück gibt.

\* = kommen übermorgen

# Type Checks: as & is

```
var obj : Int = 1
```

```
if(obj is String){  
    println(obj.length)  
}
```

```
if (x !is String || x.length == 0) return
```

**is = instanceOf**

Innerhalb des „if“ Branch wird „obj“ automatisch zu einem String gecasted.

**Smart Cast** funktioniert auch innerhalb von Ausdrücken

## Unsicherer Cast

```
val x: String = y as String
```

## Sicherer Cast

```
val x: String? = y as String?
```

## Sicherer Cast

```
val x: String? = y as? String
```

x kann hier null sein.

# Type Checks: as & is & Type Erasure

```
inline fun <reified A, reified B> Pair<*, *>.asPairOf(): Pair<A, B>? {  
    if (first !is A || second !is B) return null  
    return first as A to second as B  
}
```

Reified ermöglicht Type Checks ala **arg is T**

```
val somePair: Pair<Any?, Any?> = "items" to listOf(1, 2, 3)
```

```
val stringToSomething = somePair.asPairOf<String, Any>()
```

```
val stringToInt = somePair.asPairOf<String, Int>()
```

```
val stringToList = somePair.asPairOf<String, List<*>>()
```

```
val stringToStringList = somePair.asPairOf<String, List<String>>()
```

*// Hier wird Type Safety nicht funktionieren*

# When – Replacement of if/else& Switch

Der „**when{}**“ Block ist im Grunde eine erweiterte Form des „switch-case“ Ausdrucks von Java, aber viel mächtiger. Der größte Unterschied zwischen den beiden Ausdrücken besteht darin, dass der „**when**“ Block sowohl als **Statement** oder als **Expression** verwendet werden kann.

In Java kann das Switch Statement nur mit primitiven Datentypen, Enums oder Strings verwendet werden. In Kotlin kann der „**when**“ Block **mit jedem Typ** verwendet werden.

**Zudem** können auch **dynamische Fälle behandelt** werden, es müssen keine Konstanten sein.

## When – as a expression

```
fun getType(obj: Any?): String =  
    when (obj) {  
        1,2 -> {"is number one"}  
        "hello" -> "is hello"  
        is Long -> {"Is a long"}  
        is String -> {"Is a String"}  
        else -> throw IllegalArgumentException("Unkown value")  
    }
```

**Zwei Dinge** müssen bei Verwendung des „when“ Blocks als **Expression** berücksichtigt werden.

- Der Wert welcher zurückgegeben wird, ist der letzte Ausdruck des passenden Case Blocks.
- Es müssen sämtliche möglichen Fälle innerhalb eines „when“ Blocks abgedeckt sein.

## When – as a statement

```
val fileType = UnixFileType.HYPHEN_MINUS

when(fileType){
    UnixFileType.HYPHEN_MINUS -> {println("Regular File Type")}
    UnixFileType.D -> {"Directory"}
    UnixFileType.L -> {"Symbolic Link"}
}
```

Im Falle das der „when“ Block als Statement verwendet wird, ist es nicht nötig sämtliche Fälle abzudecken. In dieser Form ist der „when“ Block sehr ähnlich zu dem „switch-case“ von Java

# When – collections and ranges

```
private fun hasSymbolicLink(fileTypes: List<UnixFileType>) = when {  
    UnixFileType.in fileTypes -> true  
    else -> false  
}
```

```
private fun isInRange(number: Number) = when (number) {  
    in 1..10 -> true  
    else -> false  
}
```

Kotlin stellt den „in“ Operator zur Verfügung, welcher im Hintergrund zu „collection.contains(element)“ umgewandelt wird



# Ranges

Ranges werden durch die Funktion **rangeTo** geformt, welche durch den Operator **..** ausgedrückt wird und durch beiden Operatoren **in** and **!in** ergänzt wird. Der **rangeTo** Operator funktioniert für jeden Typ der **Comparable** Implementiert.

Im Falle von numerischen Ranges kann auch über die Ranges iteriert werden.

```
for (i in 1..4) print(i) // prints "1234"  
for (i in 4..1) print(i) // prints nothing  
for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "  
for (i in 4 downTo 1) print(i) // prints "4321"  
for (i in 4 downTo 1 step 2) print(i) // prints "42"  
if (str in "isle..island") println(str)
```

# Generierung von Collections

- **List:**
  - `listOf(T)` or `mutableListOf(T)`
- **Set**
  - `setOf(T)` or `mutableSetOf(T)`
- **Map**
  - `mapOf(key 'to' value, key 'to' value,...)` or `mutableMapOf(..)`

Zugriff auf Elemente einer Collections erfolgt über die **Property Syntax** und nicht wie bei Java über Getter/Setter

```
var mutableList = mutableListOf("1","2","3")  
println(mutableList[1])
```

# Verwendung von Collections

```
data class User(val name: String, val age: Int)
```

```
val users: List<User> = listOf(User("1",1),User("2",2),User("3",3),User("4",4),User("5",5))
```

```
users
```

```
    .filter { it.age < 3 }
```

```
    .sortedWith(compareBy<User>({it.age},{it.name}))
```

```
    .map{it.name}
```

```
    .forEach { println(it) }
```

*Wieviel Zeilen Code bräuchten wir wohl in Java ?*

# Konstrukturen

**class** Test{} oder **class** Test

Default Konstruktor, ohne Instanz Variablen

**class** Test (variable: Any)

Konstruktor mit einem Argument, aber ohne Instanz Variablen

```
class Test constructor(variable: Any){  
    init{  
        Log.i(TAG, "$variable")  
    }  
}
```

Konstruktor mit einem Argument und einer Initialisierung, ohne Instanz Variable

```
class Test constructor(variable: Any){  
    init{  
        val instanceVariable = variable  
    }  
}
```

Konstruktor mit einem Argument und einer Initialisierung einer Instanz Variable. Kürzer wäre folgender Ausdruck:

```
class Test(val instanceVariable: Any)
```

# Primäre und Sekundäre Konstruktoren

Wenn eine Klasse einen primären Konstruktor hat, so müssen alle sekundären Konstruktoren auf den primären Konstruktor verweisen.

```
class TestClass constructor(val variable1: Any, val variable2: Any){  
    constructor(variable1: Any) : this(variable1, "")  
    constructor() : this("", "")  
}
```

Primärer Konstruktor

Sekundäre Konstruktoren

# Extending Android Classes

```
class KotlinView : View {  
    constructor(context: Context) : this(context, null)  
    constructor(context: Context, attrs: AttributeSet?) : this(context, attrs, 0)  
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int) : super(context, attrs, defStyleAttr) {  
        ...  
    }  
}
```

Oder kürzer und lesbarer

```
class KotlinView @JvmOverloads constructor(  
    context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0) : View(context, attrs, defStyleAttr)
```

# No static Keyword, but Object

In Kotlin gibt es kein „static“ Keyword. Die Funktionalität von statischen Methoden wie wir es aus Java kennen kann mittels Higher Order Level Functions realisiert werden. Also Funktionen wecheln keiner Klasse angehören.

Um trotzdem ein Singleton zu erzeugen oder eine anonyme Instanz einer Klasse zu erzeugen wird das Schlüsselwort „object“ verwendet.

```
private val usernameListener = object: TextWatcher{  
    override fun afterTextChanged(editable: Editable) {  
        if (editable.length > 0 && verifyIsEmail(editable.toString())) {  
            username!!.error = null  
        } else {  
            username!!.error = getString(R.string.login_screen_invalid_email)  
        }  
    }  
}
```

Zudem gibt es noch das Konzept von Companion objects, bei welchen es sich um ein Object handelt was an eine Klasse gebunden ist.

# Dekonstruktion

Manchmal ist es nötig ein Objekt in seine Bestandteile (Variablen) aufzusplitten. Dies wird **destructuring declaration** genannt. Häufig sieht man dies beim Umgang mit Lambda Expressions.

`val (name, age) = person` ➔ `val name = person.component1()`  
`val age = person.component2()` ➔ `operator fun Person.component1() = getName()`  
`operator fun Person.component2() = getAge()`

`val (_, status) = getResult()`

Die `componentN()` Operator Funktionen werden nicht für Komponenten aufgerufen welche so übersprungen werden.

## Dekonstruktion von Lambdas

<code>{ a -&gt; ... }</code>	<i>// Ein Parameter</i>
<code>{ a, b -&gt; ... }</code>	<i>// Zwei Parameter</i>
<code>{ (a, b) -&gt; ... }</code>	<i>// Ein dekonstruiertes Paar</i>
<code>{ (a, b), c -&gt; ... }</code>	<i>// Ein dekonstruiertes Paar und ein anderer Parameter</i>



## Vererbung I

Im Unterschied zu Java sind sämtliche Klassen standardmäßig **final**, sollte eine Klasse vererbbar sein muss sie mit Dem Schlüsselwort **open** versehen werden. Zudem muss **jede Methode** welche **überschrieben** werden kann ebenfalls mit **open** markiert werden.

```
open class Base {  
    open fun a(){}  
    fun b(){}  
}
```

Funktion b() kann nicht überschrieben werden

```
class Derived : Base(){  
    override fun a() {}  
}
```

Die Klasse Derived ist implizit wieder **final**

Die gleichen Regeln gelten auch für das Überschreiben von Properties. Es gilt jedoch zu beachten, das **val** Properties zwar mit **var** **überschrieben werden können**, jedoch **nicht umgekehrt**.

## Vererbung II

**Abgeleitete Klassen** sind von sich aus wieder **final**, um diese ebenfalls als vererbbar zu markieren, muss wieder Das Schlüsselwort **open** verwendet werden.

Sollte **bestimmte Methoden** von einer **Überschreibung geschützt** werden, so muss das Schlüsselwort **final** verwendet werden.

```
open class Base {  
    open fun a(){}  
    fun b(){}  
}
```

```
open class Derived : Base(){  
    final override fun a() {}  
}
```

```
class Derived2 : Derived(){  
    // Es gibt nichts zu überschreiben  
}
```

## Vererbung III

Sollte eine abgeleitete Klasse die gleiche Methoden Definition sowohl von einem Interface als auch einer Klasse bekommen, so muss die abgeleitete Klasse ihre eigene Implantation vorweisen.

```
interface B{  
    fun f(){  
        println("B")  
    }  
}  
  
open class A{  
    open fun f(){  
        println("A")  
    }  
}
```

```
class Derived: A(),B{  
  
    override fun f() {  
        super<A>.f()  
        super<B>.f()  
    }  
}
```

Derived muss eine eigene Implementation aufweisen.  
Mit **super<X>** kann auf die ver. **super Definitionen** verwiesen werden.

# Innere Klassen

Klassen können in Kotlin in andere Klassen verschachtelt werden. Damit die innere Klasse jedoch Zugriff auf die Members der äußeren Klasse hat, muss diese mit dem Schlüsselwort **inner** versehen werden.

```

open class Base(){
    open fun a(){
        println("Base.a()")
    }
    open fun b(){}
}

class Derived(val member: Any): Base(){
    override fun a() {
        super.a()
        print("Derived.a()")
    }
    inner class InnerDerivedClass{

        fun g(){
            super@Derived.a() ← Ruft die Methode a der Base Klasse Base auf
            println("I have access to $member")
        }
    }
}

```

# Data Classes

Häufig werden in Klassen nur erzeugt um bestimmte Daten zu halten. Um nicht jedes mal die ganze Kette von

- Getter / Setter
- hashCode / equals
- toString
- Component Functions

implementieren zu müssen, biete Kotlin die sogenannten **data classes** an.

```
data class User(val name: String,val age: Int)
```

Voraussetzung ist jedoch das mindestens 1 Member für die Klasse definiert wird, zudem können **data classes** **weder abstrakt, sealed, inner** noch **open** sein.

Data Classes bieten zudem die **copy()** Funktion, welche ein Deep Copy des jeweiligen Objektes erzeugen, Werden beim Deep Copy keine Parameter angegeben, werden die Werte des zu kopierenden Objektes verwendet.

## Sealed Classes

Sealed classes werden dann verwendet um eine bestimmte Klassen-Hierarchie auszudrücken.  
Wenn ein Wert nur eine bestimmte Anzahl an Ausprägungen annehmen kann, aber keine andere.  
(Ähnlich wie Enums, jedoch mit dem Unterschied das es mehrere Instanzen einer Subklasse von einer Sealed Klasse geben kann)

Um eine Sealed Klasse zu erzeugen muss das sealed Schlüsselwort vor den Klassennamen gesetzt werden.  
Eine Sealed Klasse kann Subklassen haben, jedoch müssen alle Subklassen in der gleichen Datei deklariert werden.

```
sealed class Error (private constructor)
```

```
data class NetworkError(val statusCode: Int): Error()
```

```
data class DataBaseError(val dbException: Exception): Error()
```

```
object UnknownError: Error()
```

```
fun handleErrorState(error: Error) {  
    when (error) {  
        is NetworkError -> {}  
        is DataBaseError -> {}  
        is UnknownError -> {}  
    }  
}
```

# Field vs Property

```
public String name = "Chris ";    // Java Field
```

```
var name: String = "Chris "      // kotlin Property
```

Sehen beide ähnlich, sind aber 2 unterschiedliche Konzepte.

```
private String name = "Chris ";
```

```
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
}
```

Dieser Code Block wäre das Äquivalent zu der Kotlin Property

# Generelle Struktur

```
[val | var ] <Property Name> : <Property Typ> = <Property Initialisierung>  
    <getter>  
    <setter>
```

```
class PropertyClass{  
    private fun setDataFromString(data: String) {  
        println("called setDataFromString with value: $data")  
    }  
    var data: String  
        get() = this.toString()  
        set(value) {  
            setDataFromString(value)  
        }  
  
    var readOnlyProperty = "immutable var "  
        private set  
}
```



# No Backing Fields....but

Klassen in Kotlin haben *normalerweise* keine Felder. Aber manchmal ist es notwendig Felder zu haben wenn mit custom setter/getter gearbeitet wird. Daher bietet Kotlin innerhalb eines Getter/Setter die Möglichkeit ein „backing field“ zu generieren. Der **field** identifier ist jedoch nur innerhalb des Getters/Setters zugänglich

```
class PropertyClassWithField{  
  
    private fun setDataFromString(data: String) {  
        println("setDateFromString with value: $data called")  
    }  
  
    var data: String = ""  
        get() = field  
        set(value) {  
            if(value != null) field = value  
            setDataFromString(value)  
        }  
}
```

# Delegated Properties

Die Syntax von Delegated Properties sieht wie folgt aus: `[val | var] <Property Name> : <Property Typ> by <Expression>`

Der Ausdruck **<Expression>** nach **by** ist dabei das **Delegate**. Aufrufe von `get()` werden an das Delegate weitergeleitet. Daher muss das Delegate eine **get() Methode für val Typen** und eine **set() Methode für var Typen** bereitstellen.

```
class PropertyClass {  
    private var property: String by Delegator()  
}  
  
class Delegator{  
    operator fun getValue(ref: Any?, property: KProperty<*>): String{  
        return "$ref, thank you for delegating '${property.name}' to me"  
    }  
    operator fun setValue(ref: Any?, property: KProperty<*>, value: String){  
        println("$value has been assigned to '${property.name}' in $ref")  
    }  
}
```

Könnte das Interface `ReadOnlyProperty<in R, out T>` oder `ReadWriteProperty<in R,T>` implementieren

# Delegated Properties in action

```
interface UpdatableAdapter{
    fun <T> RecyclerView.Adapter<*>.autoNotify(old: List<T>,new: List<T>,compare: (T,T) -> Boolean){
        val diffUtil = DiffUtil.calculateDiff(object : DiffUtil.Callback(){
            override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int) =
                compare(old[oldItemPosition],new[newItemPosition])

            override fun getOldListSize() = old.size

            override fun getNewListSize() = new.size

            override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int) =
                old[oldItemPosition] == new[newItemPosition]

        })
        return diffUtil.dispatchUpdatesTo(this)
    }
}
```

```
private var items: List<T> by
Delegates.observable(emptyList(),{_,old,new->
autoNotify(old,new,compare)})
```

# Lazy Properties

```
fun <T>lazy(initializer: () -> T): Lazy<T> (source)
```

```
fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> (source)
```

Lazy() ist eine Funktion welche einen Lambda Ausdruck entgegen nimmt und eine Instanz von Lazy<T> zurück gibt. Der erste Aufruf von get() führt den Lambda Ausdruck aus und merkt sich das Resultat. Alle weiteren Aufrufe an get() geben das gespeicherte Resultat zurück.

```
class LazyProperty{
```

```
    val lazyValue: String by lazy {  
        println("Doing something heavy")  
        "iam finished"  
    }  
}
```

Standardmäßig ist der Auswertung der Property synchronized. Der Wert wird nur in einem Thread berechnet und alle Threads sehen den gleichen Wert.

LazyThreadSafetyMode.**PUBLICATION**

LazyThreadSafetyMode.**NONE**

# Observable Properties

Delegates Observable nimmt 2 Argumente entgegen. Den initialen Wert und den Handler für die Modifikation. Der Handler wird jedesmal aufgerufen wenn wir einen Wert zuweisen ( `set()` ). Der Handler an sich hat 3 Parameter

- Die Property an sich
- Alter Wert
- Neuer Wert

```
var property: String by Delegates.observable("<no value>") {  
    prop, old, new ->  
    println("$old -> $new")  
}
```

Der Wert ist bereits gesetzt

```
var property: String by Delegates.vetoable("<no value>"){  
    desc, old, new ->  
    !new.startsWith("s")  
}
```

Hier kann die Zuweisung verhindert werden

# Map as Delegate

Properties einer Klasse können zudem auch in Maps gespeichert werden, dies ist insbesondere dann hilfreich wenn mit Json (dyn. Properties) gearbeitet wird.

```
data class ImmuatbleUser(val map: Map<String,Any?>){  
    val name: String by map  
    val age: Int by map  
}
```

```
val immutableUser = User(mapOf(  
    "name" to "Chris",  
    "age" to 36  
))
```

```
data class MutableUser (val map: Map<String,Any?>){  
    var name: String by map  
    var age: Int by map  
}
```

```
val mutableUser = User(mutableMapOf(  
    "name" to "Chris",  
    "age" to 36  
))
```

# Variance & Generics

Um Generics in Kotlin wirklich zuverstehen müssen 3 Konzepte verstanden sein:

- Class vs Type
- SubClass vs SubType
- Variance: Covariance, contravariance und invariance

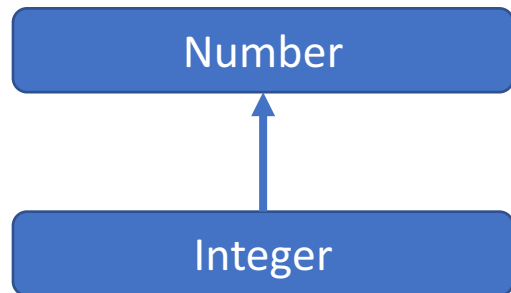
## Class vs Type

	Class	Type
String	y	y
String?	n	y
List	y	y
List<String>	n	y

# Variance & Generics

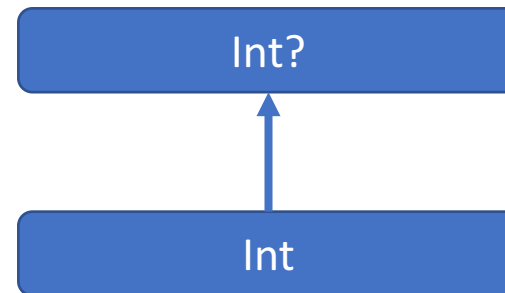
## SubClass vs SubType

Subclass



```
Integer integer = new Integer(1);  
Number number = integer;
```

Subtype

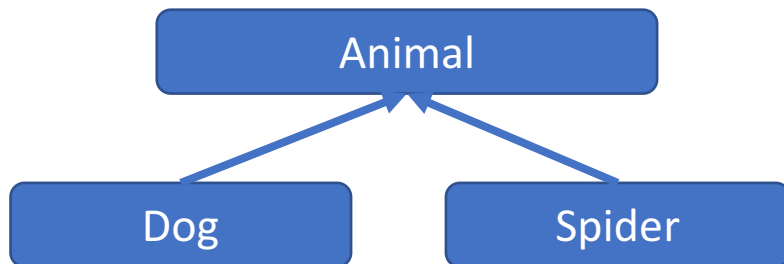


```
val integer: Int = 1  
val nullableInt: Int? = integer
```



# Variance & Generics

## Variance



```
val dog: Dog = Dog()
val spider: Spider = Spider()
var animal: Animal = dog
animal = spider
```

# Variance & Generics

## Covariance

```
val dogList: List<Dog> = listOf(Dog(10), Dog(20))
```

```
val animalList: List<Animal> = dogList
```

Variance beschreibt die Beziehung zwischen `List<Dog>` und `List<Animal>` wo `Dog` ein Subtyp von `Animal` ist

In Kotlin kann `dogList` zu einer `AnimalList` zugewiesen werden. Das heisst die Typ-Beziehung wird erhalten und es gilt `List<Dog>` ist ein Subtyp von `List<Animal>`. Dieses Verhalten wird **covariance** genannt.



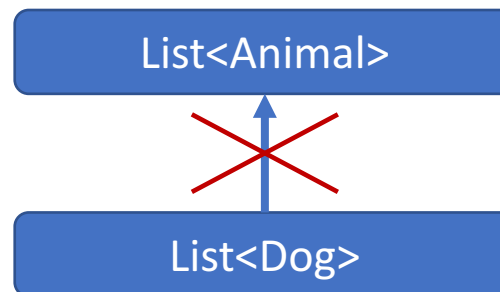
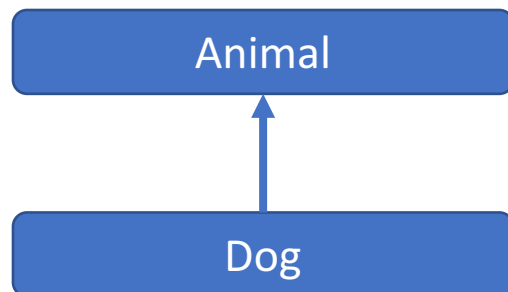
# Variance & Generics

## Invariance

```
List<Dog> dogList= new ArrayList<>();  
List<Animal> animalList = dogList; // Compiler error
```

Obschon Dog ein Subtyp von Animal ist, kann diese Zuweisung nicht in Java erfolgen.

Dies liegt daran das Generics die Beziehung zwischen Typ und Subtypen ignorieren. In Fällen wo List<Dog> nicht zu List<Animal> zugewiesen werden können, spricht man von **Invarianz**. **Es gibt keine Beziehung Subtypen und dessen SuperTypen.**



Invariance

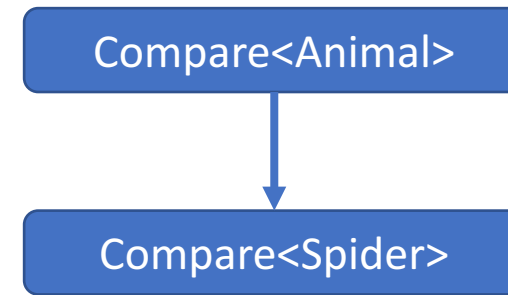
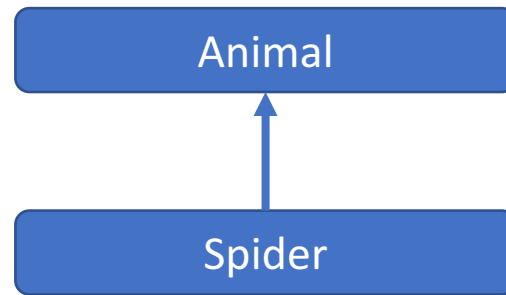
# Variance & Generics

## Contravariance

```
val dogCompare: Compare<Dog> = object: Compare<Dog> {  
    override fun compare(first: Dog, second: Dog): Int {  
        return first.cuteness — second.cuteness  
    }  
}  
  
val animalCompare: Compare<Animal> = dogCompare    // Kompilier Fehler  
  
val animalCompare: Compare<Animal> = object: Compare<Animal> {  
    override fun compare(first: Animal, second: Animal): Int {  
        return first.size — second.size  
    }  
}  
  
val spiderCompare: Compare<Spider> = animalCompare. // Dies geht
```

# Variance & Generics

## Contravariance



# Generics in Java

Ein grosser Unterschied zu den Java Generics ist das Handling von Wildcard Typen, welche es in Kotlin nicht gibt. Stattdessen verwendet Kotlin „declaration side variance“ und “type projection“.... aeh Was?

Generics in Java sind von sich aus invariant, dies bedeutet dass folgender Aufruf nicht kompiliert.

~~List<Derived> derivedList = new ArrayList<>();~~  
~~List<Base> baseList = derivedList;~~

Kompilierfehler in Java

Daher verwendet Java die Wildcard Typen um Generics variant zu machen.

Das Prinzip der Varianz Definition am Ort ihres Gebrauchs wird „**use-site-variance**“ genannt.

**Joshua Bloch:** „*Producer-extends, Consumer-super (PECS)*“ (*Effective Java*)

extends = covariant = read  
super = contravariant = write

List<? extends Animal> animals = dogs;  
Compare<? **super** Spider> spiderCompare = animalCompare;

# Generics in Kotlin

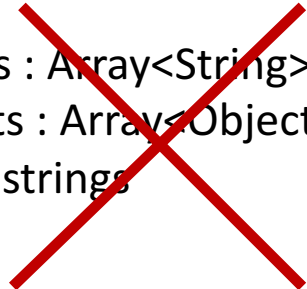
Wie auch C# verwendet Kotlin die Schlüsselwörter **in** and **out**. Ein grosser Unterschied von Java zu Kotlin sind Arrays. Während in **Java Arrays covariant** sind, sind **Arrays in Kotlin invariant** .

## Covariant (Java)

```
String[] strings = new String[0];  
Object[] objects = new Object[0];  
objects = strings;
```

## Invariant (Kotlin)

```
val strings : Array<String> = arrayOf()  
val objects : Array<Objects> = arrayOf()  
objects = strings
```



# In & Out

## Out

```
interface List<out E> {  
    fun get(index: Int): E  
}
```

Das „**out**“ Schlüsselwort bedeutet das Methoden in der Klasse List nur Typen vom Typ E zurückgeben können und das sie keine Typen vom Typ E entgegen nehmen können. Diese Limitierung erlaubt es die Klasse **covariant** zumachen.

## In

```
interface Compare<in T> {  
    fun compare(first: T, second: T): Int  
}
```

Das „**in**“ Schlüsselwort bedeutet das Methoden in der Klasse Compare nur Objekte vom Typ T entgegen nehmen können, aber nicht zurückgeben. Dies macht die Klasse **kontravariant**

In Kotlin muss der Entwickler der die Klassen Deklaration schreibt, die Varianz berücksichtigen und nicht der Entwickler der die Klasse verwendet: **declaration side variance**

„Consumer in, Producer out (CIPO)“



# Generics Constraints

Wenn der Typ eines generischen Parameters auf gewisse SubTypen eingeschränkt werden soll, so muss das Schlüsselwort **where** eingesetzt werden.

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T> where T : Comparable<T>, T: Copyable<T>{  
    return list.filter { it > threshold }.map { it.copy() }  
}
```

```
interface Copyable<out T>: Cloneable{
```

```
    fun copy(): T {  
        return this.clone() as T  
    }  
}
```

# Functions & Default Values

Im Gegensatz zu Java ist es in Kotlin möglich Default Werte für Funktion / Konstruktor Parameter zu definieren

```
class User( val name: String = "", val age : Int = 0)
```

Aber, wenn ein Parameter einem Parameter mit Default Wert folgt, so darf die entsprechende Methode nur mit dem Namen des Parameters aufgerufen werden.

```
fun a (x: Int = 10, y : Int) {}          a(y = 1)
```

Zudem gilt beim Überschreiben von Funktionen mit Default Werte das die überschreibende Klasse keine Default Werte angeben darf. (Die überschreibende Klasse verwendet implizit die Default Werte)

## Var. Args.

Eine Variable Menge von Parametern kann durch das Schlüsselwort **vararg** angegeben werden. Innerhalb der Funktion wird der entsprechende Parameter als Array interpretiert.

```
fun foo(vararg strings: String){  
    strings.forEach {  
        println(it)  
    }  
}
```

Diese Methode kann auf 2 Arten aufgerufen werden:

```
foo("1","2","3")
```

```
val a= arrayOf("1","2","3")
```

```
foo(*a)
```

\* = Spread Operator