Object Orientation

https://kotlinlang.org/

Klassen erstellen

```
class Empty {
    // ...
}

fun main(args: Array<String>) {
    val empty = Empty()
}
```

die geschwungenen Klammern sind bei einer leeren Klasse nicht notwendig

beim instanzieren einer Klasse gibt es kein „new“-keyword

Properties

```
class City {

    var name: String = ""
    var population: Int = 0

}
```

```
fun main(args: Array<String>) {
    val berlin = City()
    berlin.name = "Berlin"
    berlin.population = 3_500_000

    println(berlin.name)
    println(berlin.population)
}
```

Diese Zuweisungen erfolgen über Properties und sind daher implizit Zuweisungen über setter

detto, nur getter

Überschreiben von get/set

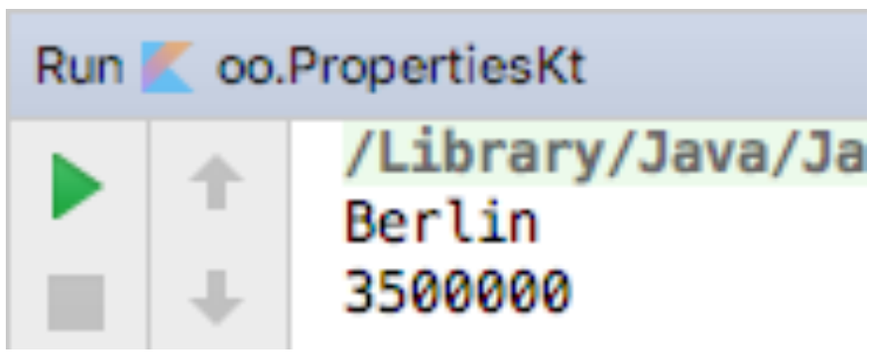
```
class City {

    var name: String = ""
        get() = field
        set(value) {
            field = value
        }

    var population: Int = 0

}

fun main(args: Array<String>) {
    val berlin = City()
    berlin.name = "Berlin"
    berlin.population = 3_500_000

    println(berlin.name)
    println(berlin.population)
}
```

Dies ist die Default-Implementation. Es würde sich nichts ändern

Erste Funktionalität im Setter

```
class City {

    var name: String = ""
        get() = field
        set(value) {
            if (value.isNotBlank()) {
                field = value
            }
        }

    var population: Int = 0

}

fun main(args: Array<String>) {
    val berlin = City()
    berlin.name = "Berlin"
    berlin.name = "  "
    berlin.population = 3_500_000

    println(berlin.name)
    println(berlin.population)
}
```

Was passiert?



Constructor: init{ … }

Variante 1

```
class Country(name: String, areaSqKm: Int) {
    val name: String
    val area: Int

    init {
        this.name = name
        this.area = areaSqKm
    }
```

init ist hier der Constructor-Ersatz.
ACHTUNG: Das ist noch eine javamäßige Zuweisung

Variante 2

```
class Country(name: String, areaSqKm: Int) {
    val name: String = name
    val area: Int = areaSqKm
}
```

Variante 3: parameter constructor

```
class Country(val name: String, val area: Int) {
}
```

Ein erstes Bsp

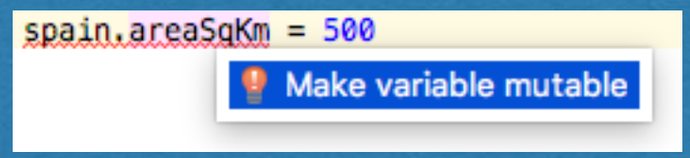
```
class Country(val name: String, val areaSqKm: Int) {

    fun print() = "$name, $areaSqKm km^2"

}

fun main(args: Array<String>) {
    val australia = Country("Australia", 7_700_000)

    println(australia.name)
    println(australia.areaSqKm)
}
```

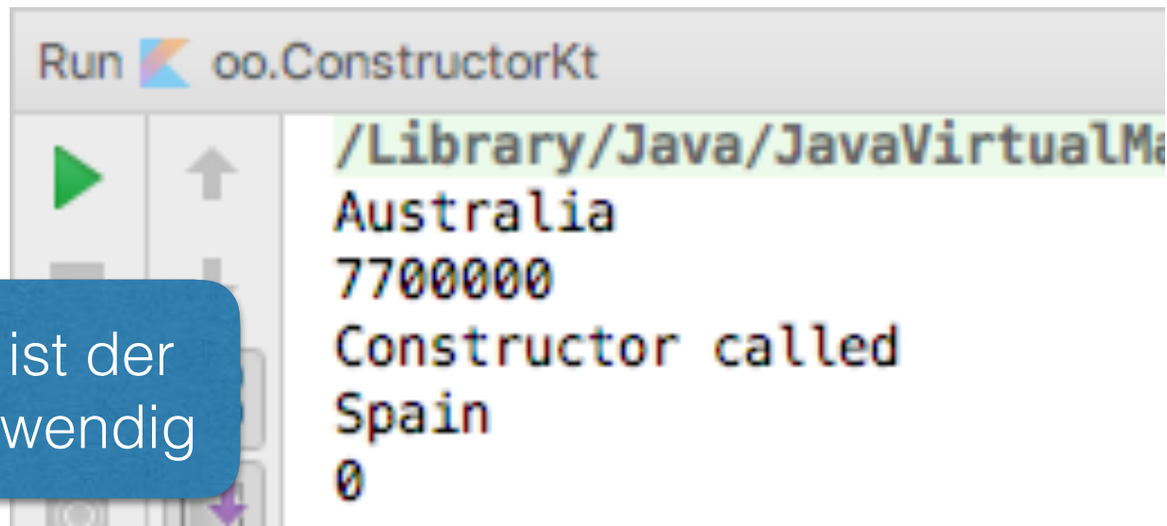
Multiple Constructor

```
class Country(val name: String, val areaSqKm: Int) {

    constructor(name: String) : this(name, 0) {
        println("Constructor called")
    }

    fun print() = "$name, $areaSqKm km^2"

}

fun main(args: Array<String>) {
    val australia = Country("Australia", 7_700_000)

    println(australia.name)
    println(australia.areaSqKm)

    val spain = Country("Spain")
    println(spain.name)
    println(spain.areaSqKm)
}
```

Ein weiterer Constructor wurde angelegt

BEACHTE: areaSqKm kann nicht mehr verändert werden, da „val“





BEACHTE: Durch den Einsatz von Default-Values ist der Einsatz von mehreren Konstruktoren oft NICHT notwendig

Methoden

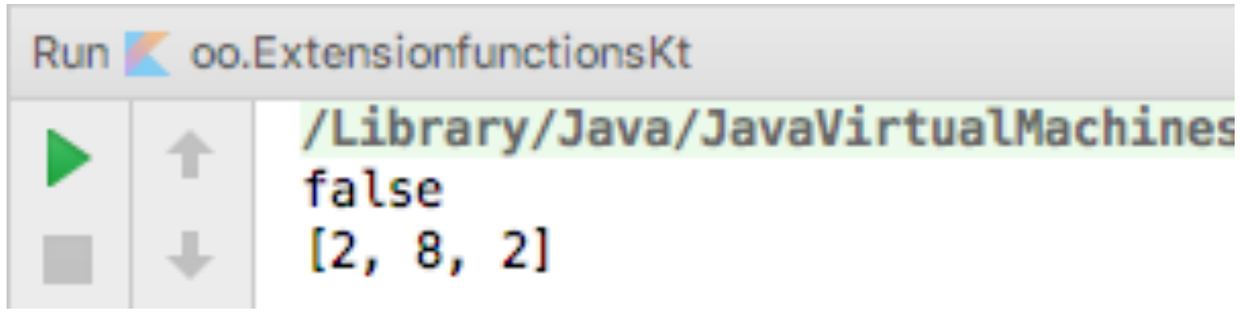
```
package oo

class Robot(val name: String) {

    fun greetHuman() {
        println("Hello human, my name is $name")
    }

    fun knowsItsName(): Boolean = name.isNotBlank()
}

fun main(args: Array<String>) {
    val fightRobot = Robot("Destroyer9000")

    if (fightRobot.knowsItsName()) {
        fightRobot.greetHuman()
    }
}
```

```
Run  oo.MethodsKt
/Library/Java/JavaVirtualMachines/jdk1.8.0_1
Hello human, my name is Destroyer9000
```

Extension Functions 1

```
// fun Int.isEven(): Boolean = (this % 2 == 0)
fun Int.isEven() = (this % 2 == 0)

fun main(args: Array<String>) {
    println(5.isEven())

    val naturals = listOf(2, 5, 11, 3, 8, 2)
    println(naturals.filter {it.isEven()})
}
```

Typ des Funktionswert kann vom Compiler abgeleitet werden



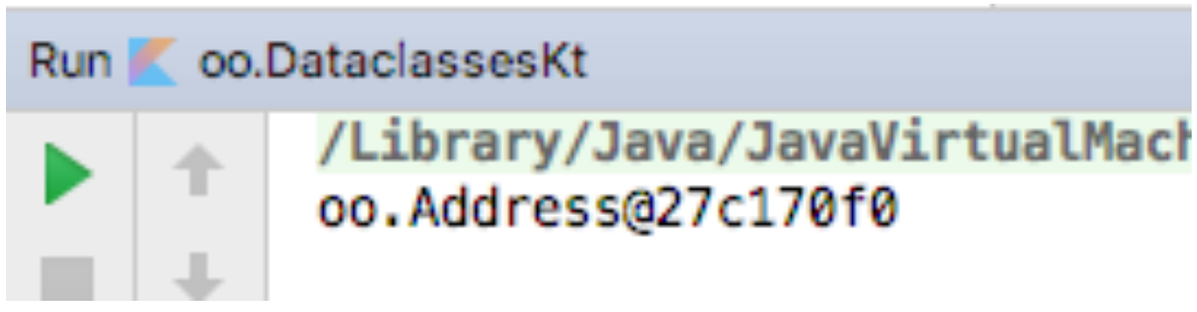
Extension Functions 2

```
fun City.isLarge() = population > 1_000_000

fun main(args: Array<String>) {

    val austin = City()
    austin.name = "Austin"
    austin.population = 950_000
    println(austin.isLarge())
}
```

Extension Method in einer eigenen Klasse

Data Classes 1

```
class Address(val street: String, val number: Int, val postCode:
String, val city: String)

fun main(args: Array<String>) {
    val residence = Address("Main street", 42, "1234", "New York")
    val residence2 = Address("Main street", 42, "1234", "New York")
    println(residence)
}
```



Data Classes 2

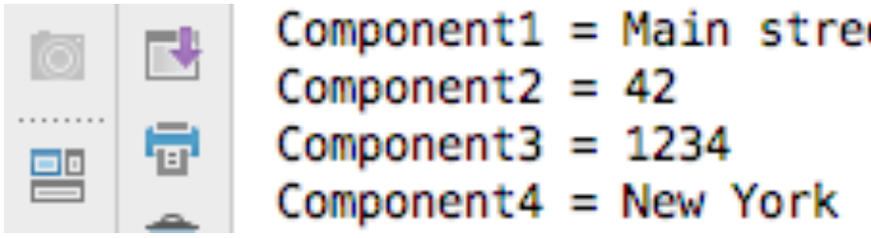
```
// Generates hashCode(), equals(), toString(), copy(), destructuring operator
data class Address(val street: String, val number: Int, val postCode: String, val city:
String)

fun main(args: Array<String>) {
    val residence = Address("Main street", 42, "1234", "New York")
    val residence2 = Address("Main street", 42, "1234", "New York")

    // toString()
    println(residence)

    // Referential equality
    println(residence === residence)

    // Structural equality, equals()
    println(residence == residence)

    // copy()
    val neighbour = residence.copy(number = 43)
    println(neighbour)

    // Destructuring operator
    println ("Component1 = ${residence.component1()}")
    println ("Component2 = ${residence.component2()}")
    println ("Component3 = ${residence.component3()}")
    println ("Component4 = ${residence.component4()}")
    val (street, number, postcode, city) = residence
    println("$street $number, $postcode $city")
}
```

```
Run  oo.DataclassesKt
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/
Address(street=Main street, number=42, postCode=1234, city=New York)
true
true
Address(street=Main street, number=43, postCode=1234, city=New York)
Component1 = Main street
Component2 = 42
Component3 = 1234
Component4 = New York
Main street 42, 1234 New York
```

Destructuring operator

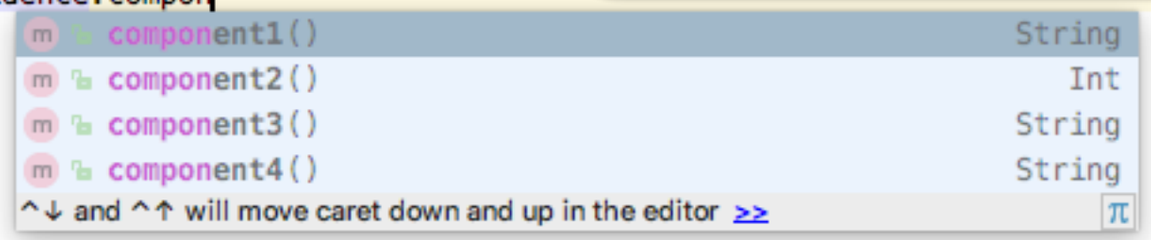
```
// Destructuring operator
println ("Component1 = ${residence.component1()}")
println ("Component2 = ${residence.component2()}")
println ("Component3 = ${residence.component3()}")
println ("Component4 = ${residence.component4()}“)
```

Jedes Klassenattribut kann mit einer component-Function abgerufen werden



```
Component1 = Main street
Component2 = 42
Component3 = 1234
Component4 = New York
```

```
// Destructuring operator
residence.compon
```




```
val (street, number, postcode, city) = residence

println("$street $number, $postcode $city")
```

I.N. verwendet man diese Version

Gleichheit

Equality

Edit Page

In Kotlin there are two types of equality:

— Referential equality (two references point to the same object);

— Structural equality (a check for `equals()`).

Referential equality

Referential equality is checked by the `===` operation (and its negated counterpart `!==`). `a === b` evaluates to true if and only if `a` and `b` point to the same object.

Structural equality

Structural equality is checked by the `==` operation (and its negated counterpart `!=`). By convention, an expression like `a == b` is translated to:

```
a?.equals(b) ?: (b === null)
```

I.e. if `a` is not `null`, it calls the `equals(Any?)` function, otherwise (i.e. `a` is `null`) it checks that `b` is referentially equal to `null`.

Enums 1

```
enum class Direction(degree: Double) {
    NORTH(0.0), EAST(90.0), SOUTH(180.0), WEST(270.0)
}

enum class Suits {
    HEARTS, SPADES, DIAMONDS, CLUBS
}

fun main(args: Array<String>) {

    val suit = Suits.SPADES

    val color = when (suit) {
        Suits.HEARTS, Suits.DIAMONDS -> "red"
        Suits.SPADES -> "black"
    }
}
```

Der Compiler erkennt, dass die Möglichkeiten nicht vollständig sind

```
s.kt
15      val color = when (suit) {
'when' expression must be exhaustive, add necessary 'CLUBS' branch or 'else' branch
```

Lösung 1: alle Möglichkeiten

```
val color = when (suit) {
    Suits.HEARTS, Suits.DIAMONDS -> "red"
    Suits.SPADES, Suits.CLUBS -> "black"
}
```

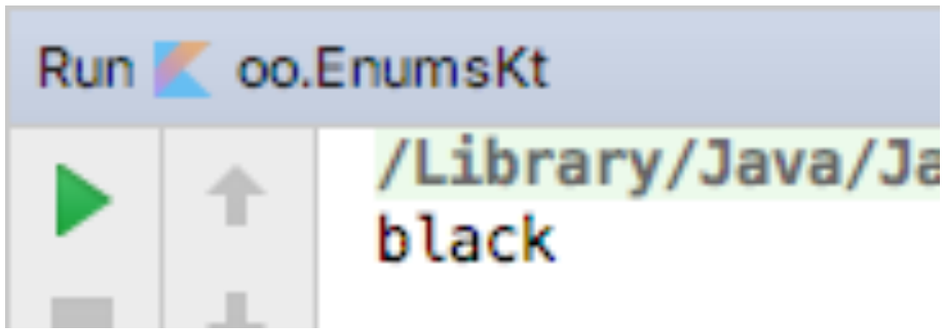
Lösung 2: else-Zweig

```
val color = when (suit) {
    Suits.HEARTS, Suits.DIAMONDS -> "red"
    Suits.SPADES -> "black"
    else -> "..."
}
```

Enums 2

```
enum class Direction(degree: Double) {
    NORTH(0.0), EAST(90.0), SOUTH(180.0), WEST(270.0)
}

enum class Suits {
    HEARTS, SPADES, DIAMONDS, CLUBS
}
```

```
fun main(args: Array<String>) {

    val suit = Suits.SPADES

    val color = when (suit) {
        Suits.HEARTS, Suits.DIAMONDS -> "red"
        Suits.SPADES, Suits.CLUBS -> "black"
    }

    println(color)
}
```



Inheritance 1

- Programming of deltas

- The similiarities are already encapsulated in the inheritance structure

Inheritance 2

```
// Minimal example
class Base
class Child : Base()
```

Make 'Base' open

In Kotlin sind Klassen per default final, dh es kann nicht davon abgeleitet werden

: wird anstelle von extends und implements verwendet

Der Unterschied zwischen Klssen und Interfaces besteht in der Klammer „()“ —> **Klassen** werden wie hier **mit Klammern** angegeben, da der Konstruktor aufgerufen werden muß, **Interfaces** hingegen werden **ohne Klammern** angegeben

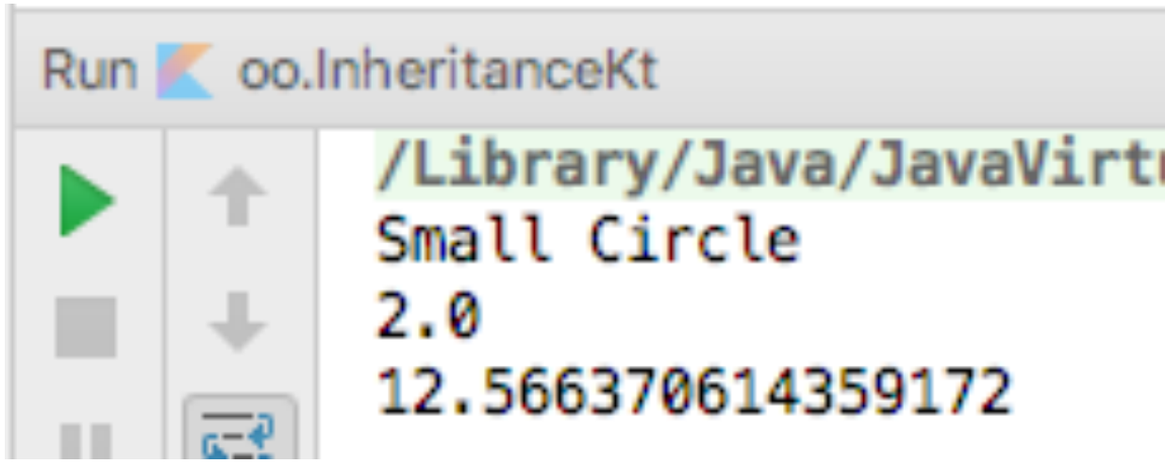
```
// Minimal example
open class Base
class Child : Base()
```

zum Thema „final class“ vgl Joshua Bloch: Effective Java

Überschreiben von Methoden

```
open class Shape(val name: String) {

    open fun area() = 0.0

}

class Circle(name: String, val radius: Double) : Shape(name) {

    override fun area() = Math.PI * Math.pow(radius, 2.0)
}

fun main(args: Array<String>) {
    val smallCircle = Circle("Small Circle", 2.0)

    println(smallCircle.name)
    println(smallCircle.radius)
    println(smallCircle.area())
}
```

„open“, um die Methode überschreiben zu können



Abstrakte Klassen und Methoden

```
abstract class Shape(val name: String) {

    abstract fun area(): Double
}

class Circle(name: String, val radius: Double) : Shape(name)
```

Besser: Im Gegensatz zum vorherigen Bsp muß man hier keine „sinnlosen“ Default-Werte (zB 0.0) angeben

```
Class 'Circle' is not abstract and does not implement abstract base class member
public abstract fun area(): Double defined in oo.Shape
```

Allerdings fehlt in der abgeleiteten Klasse (noch) die implementierte abstrakte Methode

```
fun main(args: Array<String>) {
    val shape = Shape("Name")
}
```

Cannot create an instance of an abstract class

```
fun main(args: Array<String>) {
    val shape = Circle("Large circle", 17.0)
    println(shape.area())
}
```

Abstrakte Klassen und Methoden sind implzit „open“, da es keinen Sinn machen würde, abstrakte Klassen und Methoden NICHT zu implementieren

Interfaces 1

```
interface Drivable {
    val a: Int
    fun drive()
}
```

Vereinbarung: Die Interfaces sollen immer mit „able“ enden

```
class Bicycle : Drivable{
    override val a: Int
        get() = 42
    override fun drive() {
        println("Driving bicycle")
    }
}

class Boat : Drivable {
    override val a = 43

    override fun drive() {
        println("Driving boat")
    }

}
```

Methoden und Attribute können in Interfaces überschrieben werden

Interfaces 2

```
package oo

interface Drivable {
    fun drive()
}

class Bicycle : Drivable{
    override fun drive() {
        println("Driving bicycle")
    }
}

class Boat : Drivable {
    override fun drive() {
        println("Driving boat")
    }
}

fun main(args: Array<String>) {
    val drivable: Drivable = Bicycle()
    drivable.drive()
}
```

BEACHTE: Das Objekt „drivable“ ist vom Typ „Drivable“, das bedeutet, man hat nur Zugriff auf Elemente (Methoden, Attribute), die im Interface „Drivable“ deklariert sind.
Das Objekt driveable könnte auch vom Typ „Bicycle“ oder „Boat“ sein

Good practise: use the most abstract interface

Default Implementations
sind zu vermeiden

```
interface Drivable {
    fun drive() {
        println("Driving something")
    }
}
```

Many Supertypes 1

```kotlin
abstract class Vehicle {
    open fun drive() {
        println("Driving")
    }
    abstract fun honk()
}
```

```kotlin
interface Drivable {
    fun drive() {
        println("Driving (interface)")
    }
}
```

Problem

```kotlin
class Sedan : Vehicle(), Drivable {

    override fun drive() {
        super.drive()
    }
}
```

Many supertypes available, please specify the one you mean in angle brackets, e.g. 'super<Foo>'

Lösung

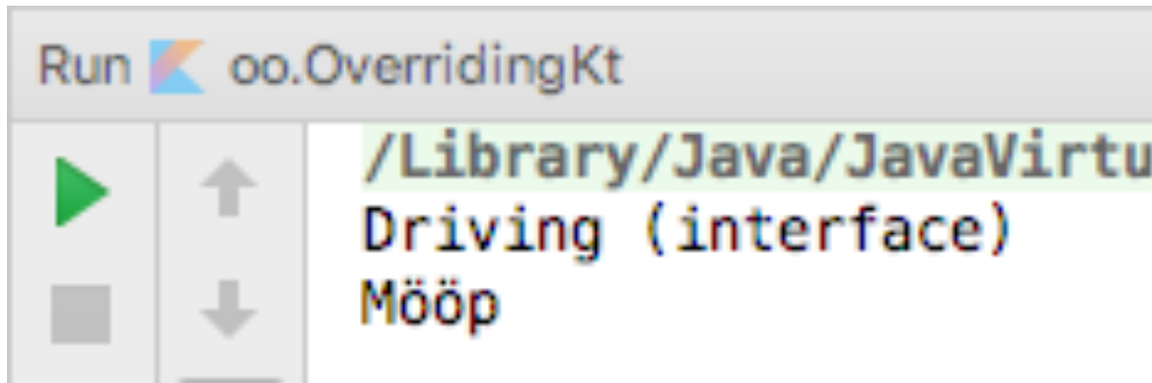
```kotlin
class Sedan : Vehicle(), Drivable {

    override fun drive() {
        super<Drivable>.drive()
    }
}
```

Many Supertypes 2

```
abstract class Vehicle {
    open fun drive() {
        println("Driving")
    }
    abstract fun honk()
}

class Sedan : Vehicle(), Drivable {

    override fun drive() {
        super<Drivable>.drive()
    }

    override fun honk() {
        println("Mööp")
    }
}

fun main(args: Array<String>) {
    val sedan = Sedan()
    sedan.drive()
    sedan.honk()
}
```

Probleme:

- Eigentlich sollte Vehicle das Interface „Drivable“ implementieren
- Weiters gibt es im Interface „Drivable“ eine Default Implementation - das ist zu vermeiden!



Inheritance

```
abstract class Vehicle(open val brand: String = "") {
    ...
}

class Sedan(override var brand: String = "BRAND") : Vehicle(), Drivable {

    ...
}
```

Diese Form der Vererbung ist möglich, es wird einer getter-Methode (bei val) eine Setter-Methode (bei var) hinzugefügt.

Smart Casts 1

```
fun Bicycle.replaceWheel() {
    println("Replacing wheel ...")
}

fun Boat.startEngine(): Boolean {
    println("Starting engine ...")
    return true
}

fun main(args: Array<String>) {

    val vehicle: Drivable = Bicycle()

    vehicle.drive()

    // instanceof <-> is
    if (vehicle is Bicycle) {
        // val b = (Bicycle) vehicle
        vehicle.replaceWheel()
    }
}
```

Smart cast to oo.Bicycle

Innerhalb des if-Statements kann der Compiler den Typ Bicycle ableiten und dass es sicher ist auf Bicycle zu casten

In Java müsste man zuerst casten, um auf die Methode „replaceWheel“ von Bicycle zugreifen zu können

Der SmartCast wird im Editor durch die grüne Farbhinterlegung angezeigt

Smart Casts 2

```
// instanceof <-> is
if (vehicle is Bicycle) {
    // val b = ((Bicycle) vehicle)
    vehicle.replaceWheel()
} else if (vehicle is Boat) {
    vehicle.startEngine()
}

if (vehicle is Boat && vehicle.startEngine()) {
    // ...
}

if (vehicle !is Boat || vehicle.startEngine()) {
    // ...
}

if (vehicle !is Bicycle) {
    return
}

vehicle.replaceWheel()
```

Hier sieht man, wie clever der Compiler erkennt, ob der Typ abgeleitet werden kann und dann automatisch einen SmartCast durchführt

Sogar hier erkennt der Compiler, daß der Programmfluss nur für Bicycles hier weitergeht

Visibilities (Scopes) 1

```
// private - same as in Java
// protected - same as in Java
// internal - visible inside the same module
// public - same as in Java (default)

open class Car(val brand: String, private val model: String) {

    protected fun tellMeYourModel() = model
}

class SpecificCar() : Car("", "") {

    fun a() {
        tellMeYourModel()
    }
}

fun main(args: Array<String>) {
    val car = Car("BRAND", "MODEL")
    car.brand
    car.tellMeYourModel()
}
```

wird verwendet, wenn kein scope angegeben wird

Cannot access 'tellMeYourModel': it is protected in 'Car'

Visibilities (Scopes) 2

auf i kann nur innerhalb des Files zugegriffen werden

public ist ausgegraut, da es sowieso Default-Wert ist

```
private val i = 42
public fun a() = 17
```

Redundant visibility modifier more... (⌘F1)

```
open class Car(val brand: String, private val model: String) {

    protected fun tellMeYourModel() = model
}
```

```
fun main(args: Array<String>) {
    val car = Car("BRAND", "MODEL")
    car.brand
    println(i)
}
```

Empfehlung: Die Zugriffe und Sichtbarkeiten sind so weit als möglich einzuschränken (**information hiding**)

Companion Objects 1

```
class House(val numberOfRooms: Int, val price: Double) {

    companion object {
        val HOUSES_FOR_SALE = 10
        fun getNormalHouse() = House(6, 199_999.0)
        fun getLuxuryHouse() = House(42, 7_000_000.0)
    }
}
```

```
fun main(args: Array<String>) {
    House.
}
```

HOUSES_FOR_SALE
getLuxuryHouse()
getNormalHouse()
equals(other: Any?)

anstelle von „**static**“ in Java, daher sind zB die Attribute der Objekte (zB numberOfRooms) nicht verfügbar

Man kann dem Companion Objekt auch einen eigenen Namen geben, hier zB Factory (zum Erzeugen von Instanzen)

```
fun main(args: Array<String>) {
    House.Companion.
}
```

HOUSES_FOR_SALE
getLuxuryHouse()
getNormalHouse()

Default-Name

```
companion object Factory {
    val HOUSES_FOR_SALE = 10
    fun getNormalHouse() = House(6, 199_999.0)
    fun getLuxuryHouse() = House(42, 7_000_000.0)
}
```

Companion Objects 2

```
interface HouseFactory {
    fun createHouse(): House
}

class House(val numberOfRooms: Int, val price: Double) {

    companion object : HouseFactory {
        val HOUSES_FOR_SALE = 10
        fun getNormalHouse() = House(6, 199_999.0)
        fun getLuxuryHouse() = House(42, 7_000_000.0)
        override fun createHouse(): House = getNormalHouse()
    }
}

fun main(args: Array<String>) {
    val normalHouse = House.getNormalHouse()
    println(normalHouse)

    println(House.HOUSES_FOR_SALE)
}
```

Man kann auch Interfaces implementieren oder von Klassen ableiten

Singletons

```
import java.awt.event.MouseAdapter
import java.awt.event.MouseEvent

object CountryFactory {
    val a = 4
    fun createCountry() = Country("Australia")
}

object DefaultClickListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent?) {
        println("Mouse was clicked")
    }
}

fun main(args: Array<String>) {
    CountryFactory.a
    CountryFactory.createCountry()
}
```

Man kann auch Interfaces implementieren oder von Klassen ableiten

Unterschiede zu Java

Packages

```
package oo

import oo.House.Companion.getNormalHouse as createHouse
import oo.House.Companion.getLuxuryHouse
import java.awt.event.MouseAdapter
import java.awt.event.MouseEvent
import basics.getExternalInput

object CountryFactory {
    val a = 4
    fun createCountry() = Country("Australia")
}

object DefaultClickListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent?) {
        println("Mouse was clicked")
    }
}

fun main(args: Array<String>) {
    CountryFactory.a
    CountryFactory.createCountry()
    createHouse()
    getLuxuryHouse()
}
```

Man kann auch Companion Objekte importieren und diesen mit dem Schlüsselwort „as“ einen eigenen Namen geben

Auch Top-Level-Deklarationen können importiert werden

Generics

- erlaubt Klassen und Methoden, die vielfältiger verwendbar sind

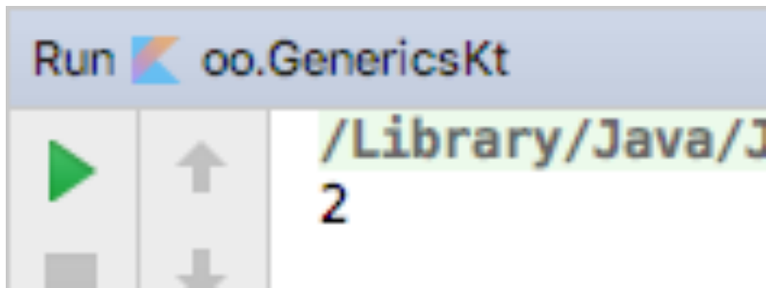
- Bsp Liste: ist für verschieden Objekte nur 1x implementiert:

- **val** *list*: List<Int> = *listOf*(1,2,3)

Generic Classes

```
import java.util.*

class Timeline<E> {

    val date2Data: MutableMap<Date, E> = mutableMapOf();

    fun add(element: E) {
        date2Data.put(Date(), element)
    }

    fun getLatest(): E {
        return date2Data.values.last()
    }
}

fun main(args: Array<String>) {
    val timeline: Timeline<Int> = Timeline()
    timeline.add(2)
    println(timeline.getLatest())
}
```

Man verwendet entweder E … Element oder T … Type



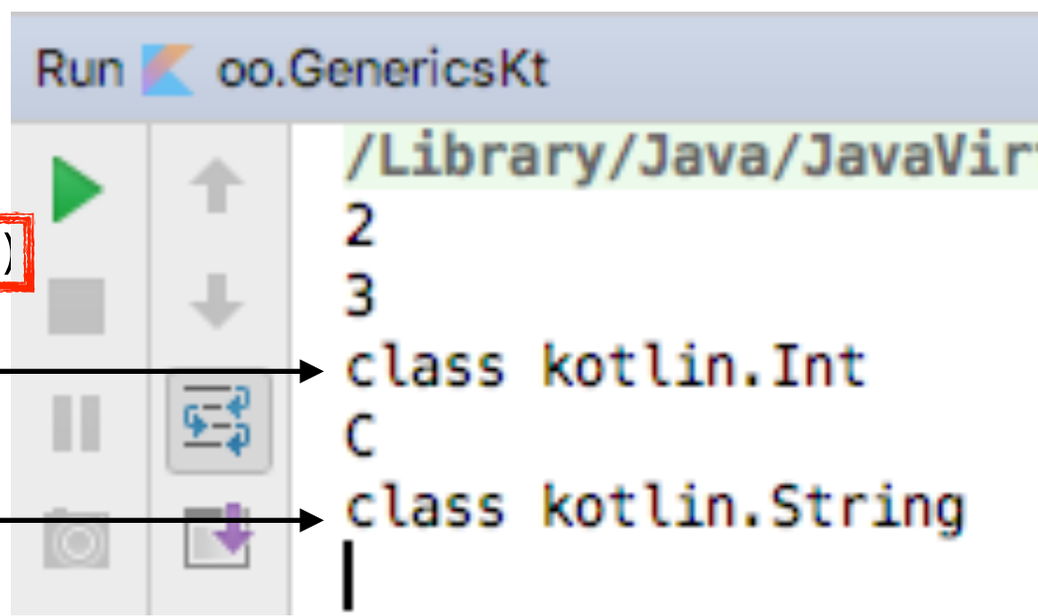
Generic Methods

```
import java.util.*

class Timeline<E> {

    val date2Data: MutableMap<Date, E> = mutableMapOf();

    fun add(element: E) {
        date2Data.put(Date(), element)
    }

    fun getLatest(): E {
        return date2Data.values.last()
    }
}
```

```
fun <E> timelineOf(vararg elements: E): Timeline<E> {
    val result = Timeline<E>()
    for (element in elements) {
        result.add(element)
    }
    return result
}
```

Der Typ kann auch abgeleitet werden

```
fun main(args: Array<String>) {
    val timeline: Timeline<Int> = Timeline()
    timeline.add(2)
    println(timeline.getLatest())

    val timeline2: Timeline<Int> = timelineOf(1, 2, 3)
    println(timeline2.getLatest())
    println(timeline2.getLatest()::class)
    val timeline3 = timelineOf("A", "B", "C")
    println(timeline3.getLatest())
    println(timeline3.getLatest()::class)
}
```



Kovarianz

```
package oo

open class Being
open class Person : Being()
class Student : Person()

fun main(args: Array<String>) {

    // Covariance = we can use a "more derived" type (a subtype)

    val people: MutableList<Person> = mutableListOf(Person(), Person())
    people.add(Student())  // covariance

    // Being does not fulfill the contract of class Person
    // people.add(Being())

    // Read-only Collections are covariant
    val p: Person = Student() // covariance
    val students: List<Person> = listOf<Student>()

    //val students2: MutableList<Person> = mutableListOf<Student>()
    // students2.add(Person())
}
```

Man kann hier zwar zur Deklaration (<Person>) Objekte vom Typ „Person“ hinzufügen, dem tatsächlich erstellten Listenobjekt (<Student>) darf aber keine „Person“ hinzugefügt werden. Daher ist so eine Konstruktion nur für immutable-Lists möglich

Noch Fragen?