

KODEIN

KOtlín DEpendency INjection

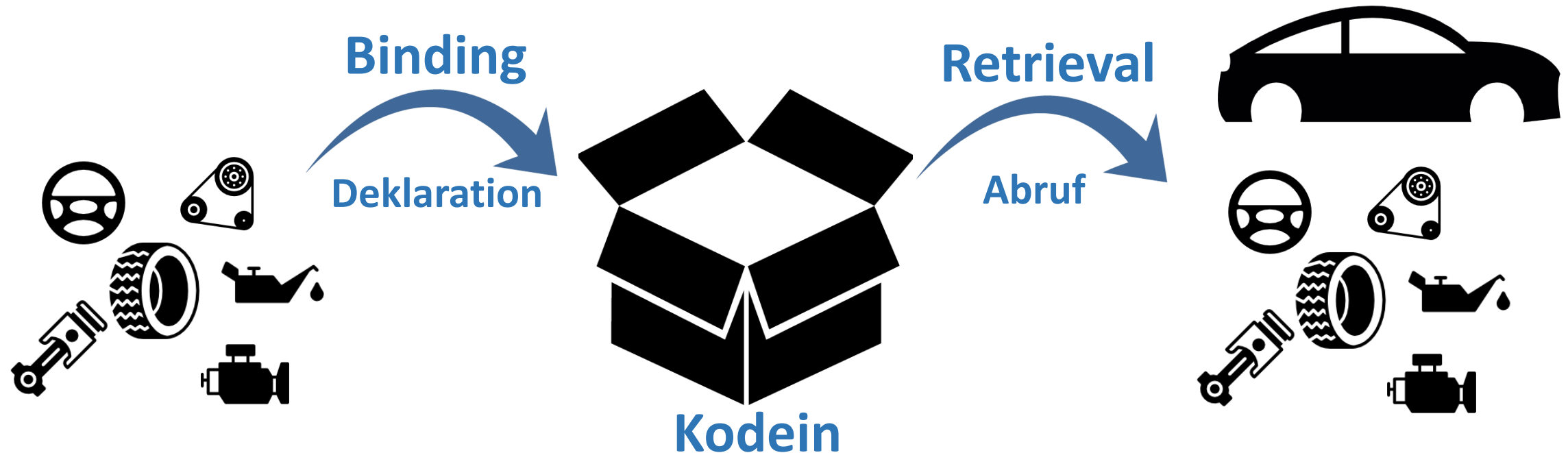
Outline

1. Was ist Kodein

2. Bindings: Abhängigkeiten deklarieren
3. Trennen - Modularisieren
4. Retrieval: Abhängigkeiten abrufen
5. Android and Kodein
6. Vor- und Nachteile von Kodein
7. Alternative DI-Frameworks

Was ist KODEIN

- Steht Für **KOTlin DEpendency INjection**
- Ist jedoch mehr ein “**dependency retrieval container**” – ein Behälter um Abhängigkeiten zu deklarieren und später hervorzuholen



Warum Dependency Injection?

- Einfache **Testbarkeit** und **Konfigurierbarkeit**
- **Struktur**, vorallem bei grossen Applikationen
- DI erlaubt **Scoping**, Singletons welche nur in einem gewissen Scope leben
- Die **Reihenfolge** von der Instanzierung der Abhängigkeiten muss nicht beachtet werden
- “**Lazy**” Instanzierung ist möglich

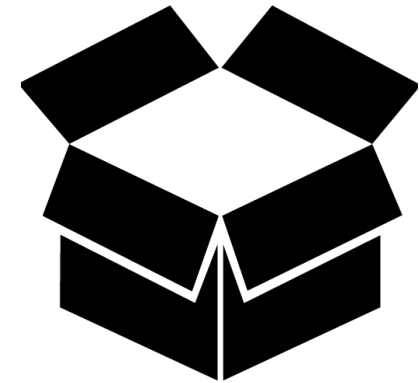
Outline

1. Was ist Kodein
- 2. Bindings: Abhängigkeiten deklarieren**
3. Trennen - Modularisieren
4. Retrieval: Abhängigkeiten abrufen
5. Android and Kodein
6. Vor- und Nachteile von Kodein
7. Alternative DI-Frameworks

Bindings: Abhängigkeiten deklarieren

Initialisierung vom Kodein Abhängigkeits-Behälter:

```
val kodein = Kodein {  
    /* Bindings */  
}
```



Bei Kodein kann zwischen parametrisierten Typen unterscheiden:

z.B: List<String> und List<Int>

Bindings: Abhängigkeiten deklarieren

- Factory:

```
val kodein = Kodein {  
    bind<Dice>() with factory { sides: Int -> RandomDice(sides) }  
}
```

- Singleton

```
bind<Dice>() with singleton { SixSidedDice() }
```

- Provider

```
bind<Dice>() with provider { RandomDice(6) }
```

Bindings: Abhängigkeiten deklarieren

- Multiton

```
bind<Dice>() with multiton{ sides: Int -> RandomDice(sides) }
```

- Instance

```
bind<DataSource>() with instance(SqliteDataSource.open("path/file"))
```

- Constant

```
constant("maxThread") with 8
```


Bindings: Abhängigkeiten deklarieren

- Scoped Singleton
 - Scoped singletons leben solange der Scope lebt

```
val kodein = Kodein {  
    bind<User>() with scopedSingleton(requestScope) { User(it.session) }  
}
```

Bindings: Abhängigkeiten deklarieren

- Tagged Bindings
 - Die Abhängigkeiten können markiert werden, so dass verschiedene Instanzen vom gleichen Typ instanziiert werden können

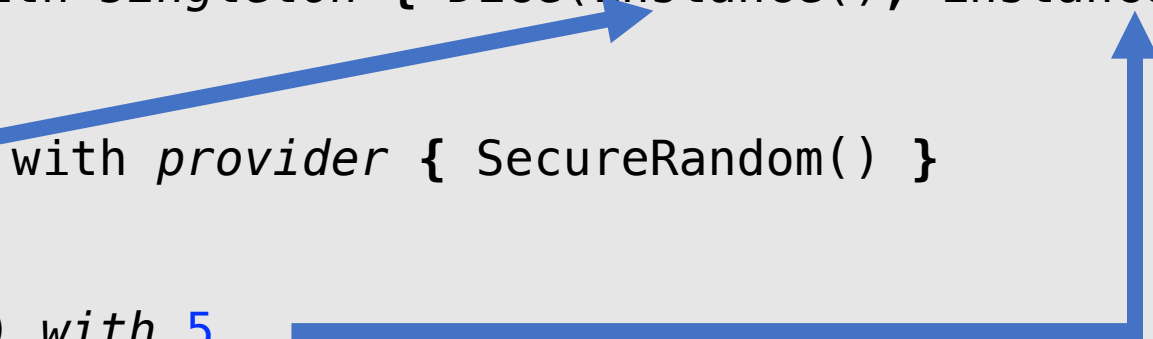
```
val kodein = Kodein {  
    bind<Dice>() with factory { sides: Int -> RandomDice(sides) }  
    bind<Dice>("DnD10") with provider { RandomDice(10) }  
    bind<Dice>("DnD20") with singleton { RandomDice(20) }  
}
```

Bindings: Abhängigkeiten deklarieren

- Transitive Abhängigkeiten

```
class Dice(private val random: Random, private val sides: Int) {  
    /*...*/  
}
```

```
val kodein = Kodein {  
    bind<Dice>() with singleton { Dice(instance(), instance("max")) }  
  
    bind<Random>() with provider { SecureRandom() }  
  
    constant("max") with 5  
}
```



Outline

1. Was ist Kodein
2. Bindings: Abhängigkeiten deklarieren
- 3. Trennen - Modularisieren**
4. Retrieval: Abhängigkeiten
5. Android and Kodein
6. Vor- und Nachteile von Kodein
7. Alternative DI-Frameworks

Modularisierung

- Die Bindings können getrennt werden
- Erlaubt Strukturierung und Übersicht
- Kann gleich konstruiert werden wie eine Kodein Instanz



Beispiel

- Modul
 - Deklaration:

```
val apiModule = Kodein.Module {  
    /* bindings */  
}
```

- Importiere das Modul in die Kodein Instanz

```
val kodein = Kodein {  
    import(apiModule)  
    /* other bindings */  
}
```

Overriding

Die zweite Deklaration überschreibt die Erste

```
val kodein = Kodein {  
    bind<API>() with singleton { APIImpl() }  
    /* ... */  
    bind<API>(overrides = true) with singleton { OtherAPIImpl() }  
}
```

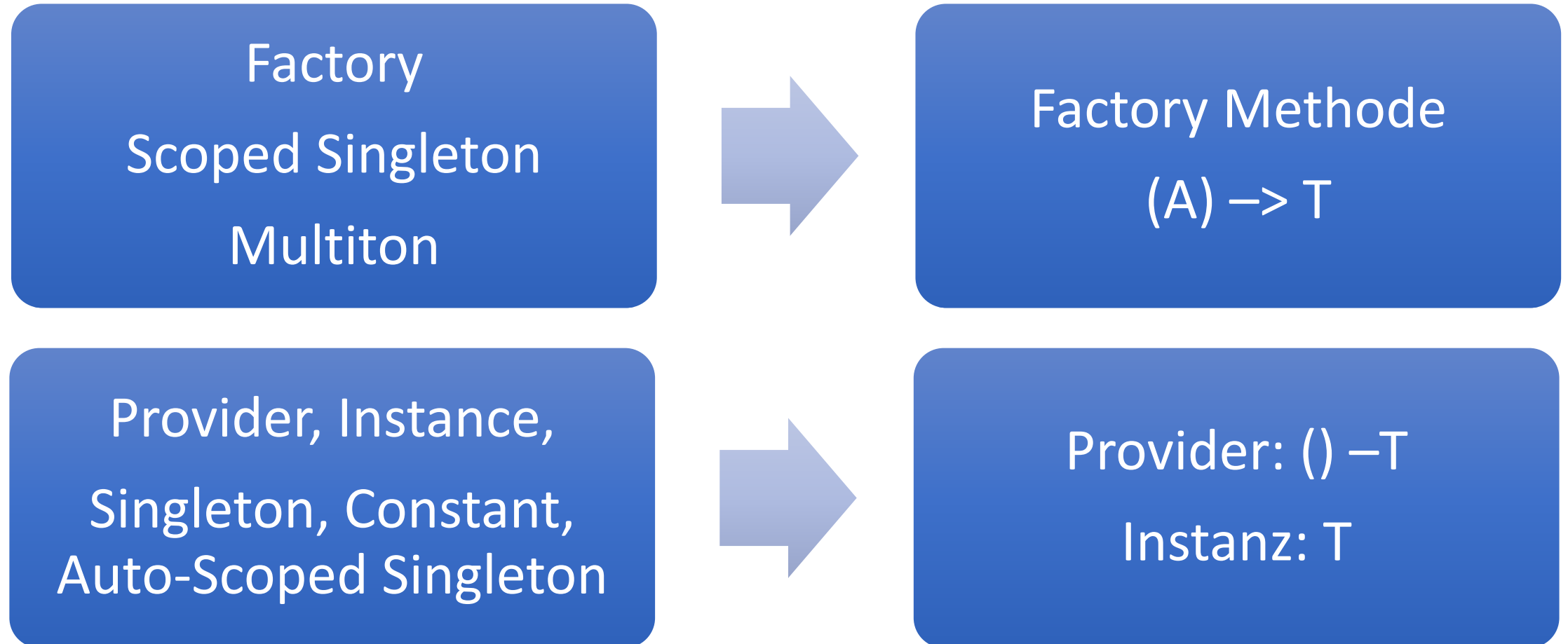
Importiert ein Modul und gibt die Erlaubnis zum Überschreiben

```
val kodein = Kodein {  
    /* ... */  
    import(testEnvModule, allowOverride = true)  
}
```

Outline

1. Was ist Kodein
2. Bindings: Abhängigkeiten deklarieren
3. Trennen - Modularisieren
- 4. Retrieval: Abhängigkeiten abrufen**
5. Android and Kodein
6. Vor- und Nachteile von Kodein
7. Alternative DI-Frameworks

Retrieval- Abhängigkeiten abrufen



Retrieval- Abhängigkeiten abrufen

Beispiel Kodein Container:

```
val kodein = Kodein {  
    bind<Dice>() with factory { sides: Int -> RandomDice(sides) }  
    bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }  
    bind<Random>() with provider { SecureRandom() }  
    constant("answer") with "fourty-two"  
}
```

Standard Abruf:

```
val diceFactory: (Int) -> Dice = kodein.factory()  
val dataSource: DataSource = kodein.instance()  
val randomProvider: () -> Random = kodein.provider()  
val answerConstant: String = kodein.instance("answer")
```

OrNull-Funktionen können benutzt werden: factoryOrNull(), instanceOrNull(), providerOrNull()...

Retrieval- Abhängigkeiten abrufen

Currying Factories: “with”

```
private val sixSideDiceProvider: () -> Dice = kodein.with(6).provider()  
private val sixSideDice: Dice = kodein.with(6).instance()
```

KodeinAware Interface - um Abrufe zu vereinfachen:

```
class MyManager(override val kodein: Kodein) : KodeinAware {  
    val datasource: DataSource = instance()  
    val diceFactory: (Int) -> Dice = factory()  
}
```

Statt
kodein.instance()
reicht
instance()

Retrieval- Abhängigkeiten abrufen

Abruf mit lazy properties

```
class Controller(private val kodein: Kodein) {  
    private val diceFactory: (Int) -> Dice by kodein.lazy.factory()  
    private val randomProvider: () -> Random by kodein.lazy.provider()  
}
```

Unbekannte Parameter mit Lambda übergeben

```
private val randomSideDiceProvider: () -> Dice by kodein.with {  
    random.nextInt(20) + 1  
}.lazy.provider()
```

kodein.lazy.factoryOrNull, kodein.lazy.providerOrNull and kodein.lazy.instanceOrNull

Retrieval- Abruf mit Injector

- Ein Injektor ermächtigt:
 - All die injizierten Abhängigkeiten auf einmal abzurufen
 - Die Abhängigkeiten ohne Kodein-Instanz abzurufen

```
class Controller() {  
    private val injector = KodeinInjector()  
    private val diceFactory: (Int) -> Dice by injector.factory()  
    /*...*/  
    private val kodein by injector.kodein()  
    fun whenReady(kodein: Kodein) = injector.inject(kodein)  
}
```



Wenn eine property benutzt wird, bevor "injector.inject(kodein)" aufgerufen wurde, wird eine "KodeinInjector.UninjectedException" geworfen

Retrieval- Abruf mit KodeinInjected Implementation


- Einfachere Syntax für die Injection:

```
class MyManager() : KodeinInjected {  
    override val injector = KodeinInjector()  
    val ds: DataSource by instance()  
}
```



instance() statt injector.instance()

Retrieval- Benutzen von Kodein.lazy

```
val kodein = Kodein.lazy {  
    println("doing bindings")  
    bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }  
}  
  
class Controller() {  
    val ds: DataSource by kodein.instance()  
  
    fun someFunction() {  
        ds.open()   
    }  
}
```

Erst beim ersten Aufruf entsteht das Binding.
-> Erst hier wird "doing bindings" ausgegeben

Retrieval- Klassen Factories

- Das Objekt welches aufgerufen wird ist abhängig von der Klasse des Objektes welches den Zugriff benötigt:

```
val kodein = Kodein {  
    bind<Logger>() with multiton  
    { cls: Class<*> -> LogManager.getLogger(cls) }  
}
```

```
class MyManager(val kodein: Kodein) {  
    val logger: Logger = kodein.withClassOf(this).instance()  
}
```


Multibinding – Set Binding

- Das Binding

```
val kodein = Kodein {  
    bind() from setBinding<Configuration>() // Ein Set-Binding erstellen  
    //Verschiedene implementationen ins Set binden:  
    bind<Configuration>().inSet() with provider { FooConfiguration() }  
    bind<Configuration>().inSet() with singleton { BarConfiguration() }  
}
```

- Retrieval- Abrufen des Sets

```
val configurations: Set<Configuration> = kodein.instance()
```

Retrieval- Zusammenfassung

- Kodein Methoden
 - Bindings, Standard Abruf, Currying factories, KodeinAware
- Lazy Properties
- Injector
 - Ein Injector Objekt,
 - Eine Kodein Injected Klasse
- Lazy Kodein
 - Ein lazy Kodein Objekt
 - Eine lazy Kodein Injected Klasse

Exercise 1

Abhängigkeiten mit Kodein deklarieren und abrufen
Binding + Retrieval regardless of Android

Tag: 02_kodein_exercise1_TAG
Branch: step_02_kodein_exercise1

Lösung Exercise 1

Abhängigkeiten mit Kodein deklarieren und abrufen
Binding + Retrieval regardless of Android

Tag: 02_kodein_exercise1_solution_TAG
Branch: step_02_kodein_exercise1_solution

Outline

1. Was ist Kodein
2. Bindings: Abhängigkeiten deklarieren
3. Trennen - Modularisieren
4. Retrieval: Abhängigkeiten abrufen
- 5. Android and Kodein**
6. Vor- und Nachteile von Kodein
7. Alternative DI-Frameworks

Android - Application Klasse

- Abhängigkeiten können in der “Application” deklariert werden

```
class MyApp : Application(), KodeinAware {  
    override val kodein by Kodein.lazy {  
        /* bindings */  
    }  
}
```



„**Kodein.lazy**“ erlaubt uns zum Binding-Zeitpunkt auf den Kontext zuzugreifen

Bootstrapping Android – 2 Varianten

- Basierend auf Inheritance (Vererbung)
- **Interface Basierend**
 - Vorteil: wir können noch von anderen Klassen erben





Android – Kodein Basisklassen

Interface Basierend

- **ActivityInjector**
- **FragmentActivityInjector**
- **AppCompatActivityInjector**
- **FragmentInjector**
- ...

Basierend auf Vererbung

- **KodeinActivity**
- **KodeinFragmentActivity**
- **KodeinAppCompatActivity**
- **KodeinFragment**
- ...



Bootstrapping Android – Interface

1. Implementiere das entsprechende Interface
2. Instanziere den Injector
3. Die “provideOverridingModule()”-Funktion erlaubt uns Bindings höher in der Hierarchie zu überschreiben
4. Wir müssen uns um den Lifecycle des Injektors kümmern
 - Im onCreate() initialisieren
 - Im onDestroy() zerstören



```
class MyFragment : BaseFragment(), SupportFragmentManager {  
    override val injector: KodeinInjector = KodeinInjector()
```

1. Injector implementieren

```
    private val logTag: String by instance("log-tag")  
    private val app: Application by injector.instance()
```

2. Injector Instanziert

```
    override fun provideOverridingModule() = Kodein.Module {  
        bind<MyFragment>() with instance(this@MyFragment)  
        bind<String>("log-tag", overrides = true) with instance("MyFragment")  
    }
```

3. Bindings überschreiben

```
    override fun onCreate(savedInstanceState: Bundle) {  
        super.onCreate(savedInstanceState)
```

```
        initializeInjector()
```

4. Injector initialisieren

```
        Log.i(logTag, "OnCeate MainActivity in ${app.applicationInfo.className}")
```

```
    }  
  
    override fun onDestroy() {  
        destroyInjector()  
        super.onDestroy()
```

5. Injector zerstören

```
    }
```



Android Scopes

- Singletons welche nur während einem gewissen Android Scope leben sollen:
 - `androidContextScope`
 - `androidActivityScope`
 - `androidServiceScope`
 - `androidFragmentScope`
 - `androidSupportFragmentScope`
 - `androidBroadcastReceiverScope`



Android Scopes - Example

Beispiel: Activity-Scoped Binding:

```
val kodein = Kodein {  
    bind<Logger>() with scopedSingleton(androidActivityScope) {  
        LogManager.getNamedLogger(it.localClassName)  
    }  
}
```

Beispiel: Activity-Scoped Abhängigkeiten abrufen:

```
val logger: Logger = kodein.with(getActivity()).instance()
```



Android Auto Scopes

Beispiel: Auto-Activity-Scope:

```
val kodein = Kodein {  
    bind<Logger>() with autoScopedSingleton(androidActivityScope) {  
        LogManager.getNamedLogger(it.localClassName)  
    }  
}
```

Beim Aufruf muss der Context nicht mehr mitgegeben werden:

```
val logger: Logger = kodein.instance()
```



Android Auto Scopes

- Um Auto-Scopes verwenden zu können muss der LifecycleCallback in der Application-Klasse angegeben werden:

```
class RedditApp : Application(), KodeinAware{  
  
    override fun onCreate() {  
        super.onCreate()  
        registerActivityLifecycleCallbacks(  
            androidActivityScope.lifecycleManager  
        )  
    }  
}
```



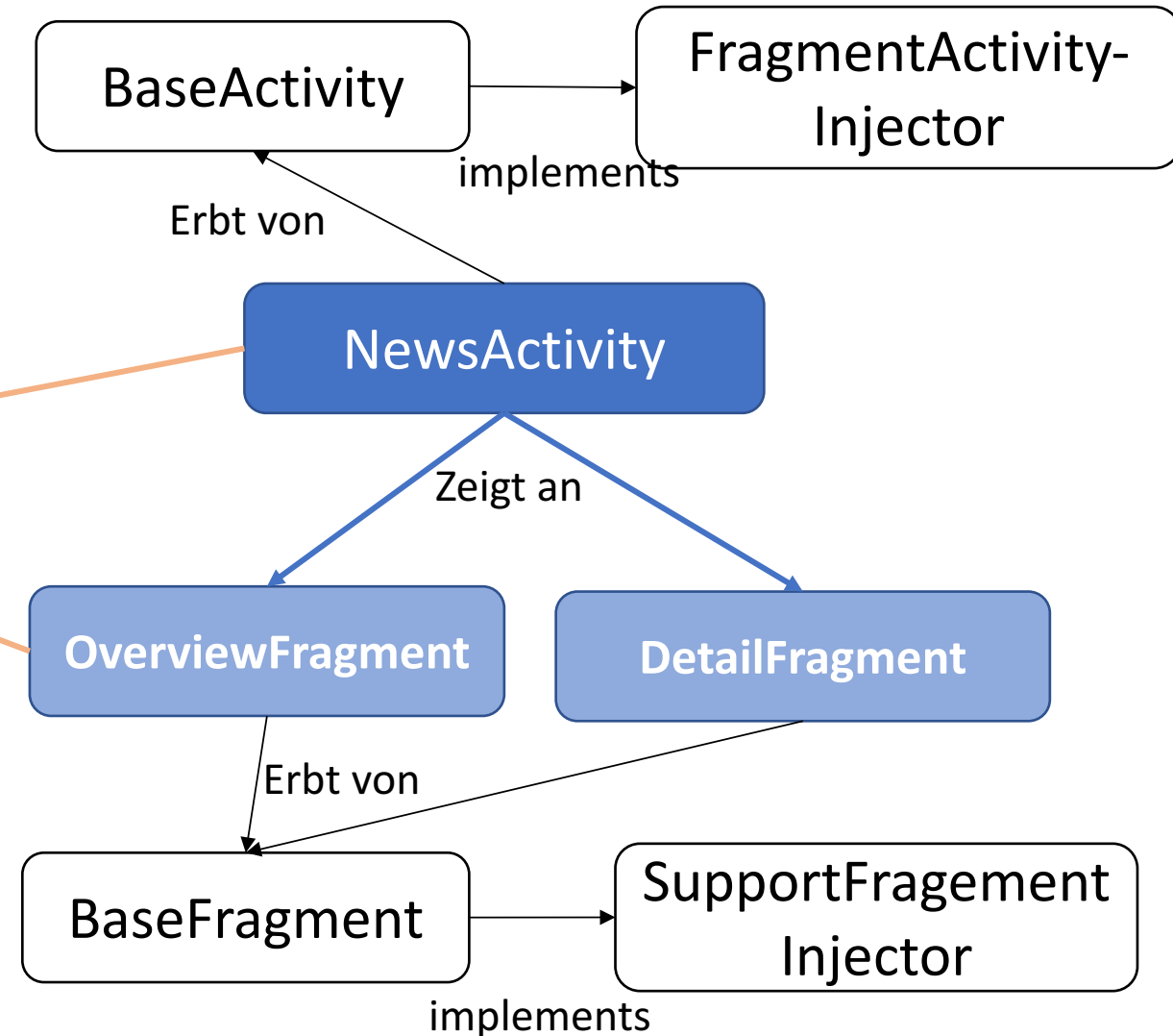
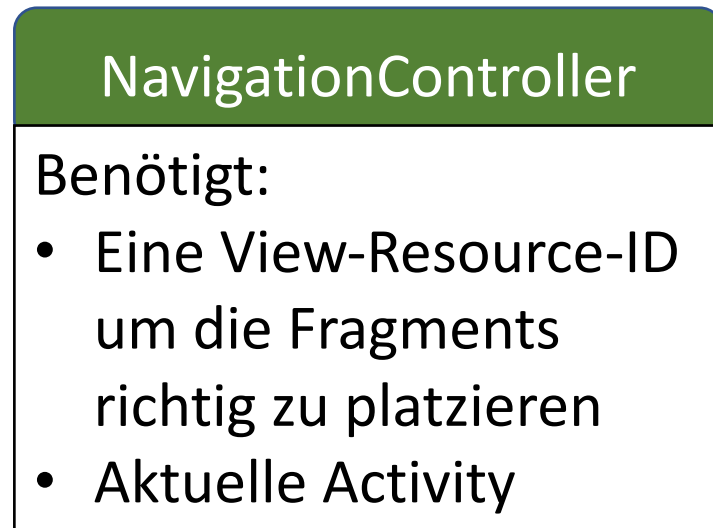
Zurück zur Reddit App!



Tag: 02_kodein_exercise2_TAG
Branch: step_02_kodein_exercise2

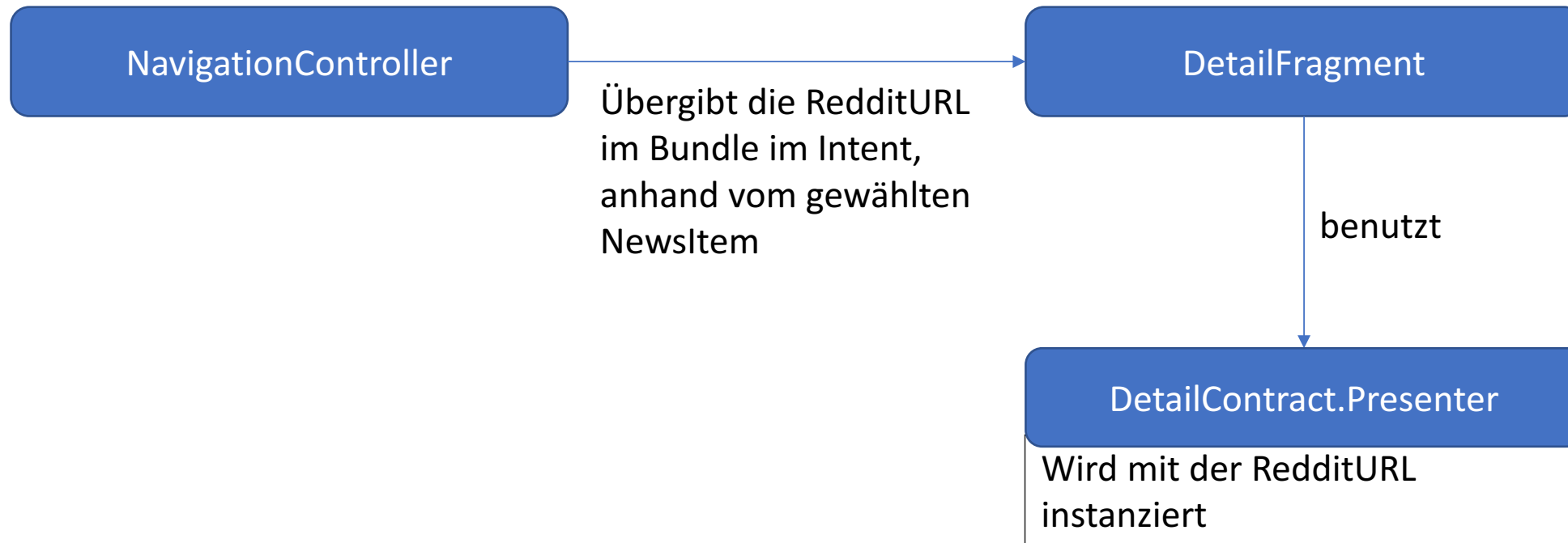


Neue Struktur





Anwendungsbeispiel– Exercise 2a



Wie kann DetailContract.Presenter mit Kodein instanziiert werden?

Suche im Code nach: "TODO: kodein_exercise2a"



Anwendungsbeispiel– Exercise 2b

```
java.lang.IllegalArgumentException: Parameter specified as non-null is null: method  
kotlin.jvm.internal.Intrinsics.checkNotNull, parameter context at  
com.github.salomonbrys.kodein.android.androidActivityScope.getRegistry(AndroidScopes.kt)  
...  
at com.github.salomonbrys.kodein.internal.KodeinContainerImpl$_transformBinding$1.invoke(KodeinContainerImpl.kt:129)  
at com.github.salomonbrys.kodein.InjectedExceptionKt$toInstance$1.invoke(InjectedException.kt:189)  
at kotlin.SynchronizedLazyImpl.getValue(Lazy.kt:130) at  
ch.zuehlke.sbb.reddit.features.news.overview.OverviewFragment.getMNavigationController(OverviewFragment.kt:0)  
at ch.zuehlke.sbb.reddit.features.news.overview.OverviewFragment.showRedditNewsDetails(OverviewFragment.kt:117)
```

- Der NavController im OverviewFragment kann nicht abgerufen werden. Context ist null!
- Was nun?



Anwendungsbeispiel– Exercise 2b

- Lazy kann nicht benutzt werden, da wir Interface-basierend und mit Injector die Abhängigkeiten abrufen
- Benutze **autoScopedSingleton**
- Beim Abruf muss der Context nicht angegeben werden
- Übergabe von Context wird von der android-kodein library behandelt

Suche im Code nach: “TODO: kodein_exercise2b”

Lösung Exercise 2

Tag: 02_kodein_exercise2_solution_TAG
Branch: step_02_kodein_exercise2_solution

Outline

1. Was ist Kodein
2. Bindings: Abhängigkeiten deklarieren
3. Trennen - Modularisieren
4. Retrieval: Abhängigkeiten abrufen
5. Android and Kodein
- 6. Vor- und Nachteile von Kodein**
7. Alternative DI-Frameworks

Vor- und Nachteile von Kodein

- **Vorteile**

- Einfach zu lernen
- Sehr gut nutzbar für Android (Activity Scopes usw...)

- **Nachteile**

- Laufzeit Injection -> Kodein.NotFoundException

Outline

1. Was ist Kodein
2. Bindings: Abhängigkeiten deklarieren
3. Trennen - Modularisieren
4. Retrieval: Abhängigkeiten abrufen
5. Android and Kodein
6. Vor- und Nachteile von Kodein
- 7. Alternative DI-Frameworks**

Alternative DI-Frameworks

- **Dagger 2**

- Schwierig zu lernen
- Mehr boilerplate Code
- Vorteil: Compile-Zeit Injection
- Mehr Struktur

- **Kein DI-Framework -> ServiceProvider**

- Bei grossen Apps unübersichtlich
- Wer braucht welche Abhängigkeiten, ist nicht klar

- **Koin**

- Ist noch im Alpha-Status
- Kein Scoping

Quellen

- https://salomonbrys.github.io/Kodein/#_introduction
- <https://medium.com/@AllanHasegawa/from-dagger2-to-kodein-a-small-experiment-9800f8959eb4>