Retrieval - Abhängigkeiten abrufen - Kodein - Kotlin Workshop

Tuesday, 14 November 2017
08:41

Beispiel Bindung benutzt in diesem Skript

```
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"
}
```

Regeln um Abhängigkeiten zu abzurufen

- Eine Abhängigkeit gebunden mit factory, scopedSingleton oder multiton kann nur mittels factory-methode aufgerufen werden: (A) → T.
- Eine Abhangigkeit gebunden mit provider, instance, singleton, eagerSingleton, refSingleton, autoScopedSingleton oder constant kann aufgerufen werden mit:
  - as a provider method: () → T
  - as an instance: T

Via Kodein methods

Standard

Beispiel um Abhängigkeiten mit einer Kodein Instanz abzurufen:

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

Falls man nicht sicher bist ob der Typ gebunden worden ist können die *OrNull-Methoden benutzt werden:

```
val diceFactory: ((Int) -> Dice)? = kodein.factoryOrNull()
val dataSource: DataSource? = kodein.instanceOrNull()
val randomProvider: (() -> Random)? =
kodein.providerOrNull()
val answerConstant: String? =
kodein.instanceOrNull("answer")
```

Currying factories

Ein Provider oder Instanz von einem Factory-gebundenen Typ kann mit "with" benutzt werden:

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

Eine Klasse kann das KodeinAware Interface implementieren, um von der einfacheren Syntax zu profitieren: 'kodein.' kann weggelassen werden:

```
class MyManager(override val kodein: Kodein) :
KodeinAware {
    val datasource: DataSource = instance()
```

```
    val random: Random = instance()
    val diceFactory: (Int) -> Dice = factory()
    val d6: Dice = with(6).instance()
}
```

Via Lazy properties injizieren

```
class Controller(private val kodein: Kodein) {
    private val diceFactory: (Int) -> Dice by
    kodein.lazy.factory()
    private val dataSource: DataSource by
    kodein.lazy.instance()
    private val randomProvider: () -> Random by
    kodein.lazy.provider()
    private val answerConstant: String by
    kodein.lazy.instance("answer")

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

Wenn der Parameter mit welchem die factory gefüttert werden soll noch unbekannt ist, kann er als lambda übergeben werden. Der Parameter wird erst abgerufen wenn er gebraucht wird

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

Via Injector

Aus einem Injector

Ein Injector ist ein Objekt welches man benutzen kann um alle Abhängigkeits-Properties in einem Objekt zu injizieren
Das ermächtigt das Objekt:

- All seine injizierten Abhängigkeiten auf einmal abzurufen
- Die Abhängigkeiten ohne Kodein-Instanz zu abzurufen

```
class Controller() {
    private val injector = KodeinInjector()
    private val diceFactory: (Int) -> Dice by
    injector.factory()
    private val dataSource: DataSource by
    injector.instance()
    private val randomProvider: () -> Random by
    injector.provider()
    private val answerConstant: String by
    injector.instance("answer")
    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

Aus einer Kodein injizierten Klasse

Wenn KodeinInjected implementiert wird, profitiert man von einer einfacheren Syntax für die Injection: 'injector.' kann weggelassen werden

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

Via einem Lazy Kodein

Manchmal hat man keinen direkten Zugriff auf die Kodein Instanz, aber man weiss dass man später auf sie zugreifen kann. In diesem Fall kann Lazy Kodein benutzt werden.

```
class Controller() {
    private val kodein = LazyKodein { /* code to access a
    Kodein instance */ }
    private val diceFactory: (Int) -> Dice by
    kodein.factory() /*Creating lazy properties*/
    private val answerConstant: String by
    kodein.instance("answer")

    fun someFunction() { val dataSource: DataSource =
    kodein().instance() } /*Um auf Kodein instanz
    zuzugreifen, kodein() wird benutzt*/
}
```