KOtlin DEpendency INjection

Salomon BRYS

2017-03-24

Table of Contents

Introduction

Kodein is a very simple and yet very useful dependency retrieval container, it is very easy to use and configure.

Kodein allows you to:

- Lazily instantiate your dependencies when needed.
- Stop caring about dependency initialization order.
- Easily bind classes or interfaces to their instance or provider.
- Easily debug your dependency bindings and recursions.

Kodein does **not** *allow you to:*

- Automatically instantiate your dependencies via injected constructor and reflexivity. *For that, you need Guice.*
- Have dependency injection validated at compile time. *For that, you need Dagger.*

Kodein is a good choice because:

- It is small, fast and optimized (makes extensive use of `inline`).
- It proposes a very simple and readable declarative DSL.
- It is not subject to type erasure (like Java).
- It integrates nicely with Android.
- It proposes a very kotlin-esque idiomatic API.
- It can be used in plain Java.

Example

An example is always better than a thousand words:

```
val kodein = Kodein {
    bind<Dice>() with provider { RandomDice(0, 5) }
    bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }
}

class Controller(private val kodein: Kodein) {
    private val ds: DataSource = kodein.instance()
}
```

Install

With Maven

```
<dependency>
    <groupId>com.github.salomonbrys.kodein</groupId>
    <artifactId>kodein</artifactId>
    <version>3.3.0</version>
</dependency>
```

With Gradle

```
compile 'com.github.salomonbrys.kodein:kodein:3.3.0'
```

Using Proguard

If you are using Proguard, you need to add the following line to your proguard configuration file:

```
-keepattributes Signature
```

Migrating

If you are migrating from Kodein 2.8, you can read the migration document.

Bindings: Declaring dependencies

Example: initialization of a Kodein variable

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

Bindings are declared inside a Kodein initialization block, and they are not subject to type erasure (e.g. You can bind both a `List<Int>` and a `List<String>` to different list instances, providers or factories).

There are different ways to declare bindings:

Factory binding

This binds a type to a factory function, which is a function that takes an argument of a defined type and that returns an object of the bound type. Each time you need an instance of the bound type, the function will be called.

Example: creates a new Dice each time you need one, according to an Int representing the number of sides

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

Provider binding

This binds a type to a provider function, which is a function that takes no arguments and returns an object of the bound type. Each time you need an instance of the bound type, the function will be called.

Example: creates a new 6 sided Dice entry each time you need one

```
val kodein = Kodein {
    bind<Dice>() with provider { RandomDice(6) }
}
```

Singleton binding

This binds a type to an instance of this type that will lazily be created at first use. Therefore, the provided function will only be called once: the first time an instance is needed.

Example: creates a DataSource singleton that will be initialized on first access

```
val kodein = Kodein {
    bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }
}
```

Eager singleton binding

This is the same as a regular singleton, except that the provider method will be called as soon as the kodein instance is created and all bindings are defined.

Example: creates a DataSource singleton that will be initialized as soon as the binding block ends

```
val kodein = Kodein {
    // The SQLite connection will be opened as soon as the kodein instance is ready
    bind<DataSource>() with eagerSingleton { SqliteDS.open("path/to/file") }
}
```

Multiton binding

A multiton can be thought of a "singleton factory". A multiton guarantees to always gives the same object given the same argument. In other words, for a given argument, the first time a multiton is called with this argument, it will create an object; and will always yield that same object when called with the same argument.

Example: creates one random generator for each value

```
val kodein = Kodein {
    bind<RandomGenerator>() with multiton { max: Int -> SecureRandomGenerator(max) }
}
```

Referenced singleton or multiton binding

A referenced singleton is an object that is guaranteed to be single as long as a reference object can return it. A referenced multiton is an object that is guaranteed to be single for the same argument as long as a reference object can return it.

Kodein comes with three reference makers.

Soft & weak

These are objects that are guaranteed to be single in the JVM at a given time, but not guaranteed to be single during the application lifetime. If there are no more strong references to the objects, they may be GC'd and later, a new object re-created.

Example: creates an Cache object that will exist once at a given time

```
val kodein = Kodein {
    bind<Cache>() with refSingleton(softReference) { LRUCache(16 * 1024) } ①
}
```

① Because it's bound by a soft reference, the JVM will GC it before any `OutOfMemoryException` can occur.

Weak singletons use Java's `WeakReference` while soft singletons use Java's `SoftReference`.

Thread local

This is the same as the standard singleton binding, except that each thread gets a different instance. Therefore, the provided function is called once per thread that needs the instance.

Example: creates a Cache object that will exist once per thread

```
val kodein = Kodein {
    bind<Cache>() with refSingleton(threadLocal) { LRUCache(16 * 1024) }
}
```

Semantically, thread local singletons should use scoped singletons, the reason it uses a referenced singleton is because Java's `ThreadLocal` acts like a reference.

Instance binding

This binds a type to an instance **already created**.

Example: a DataSource binding to an already existing instance.

```
val kodein = Kodein {
    bind<DataSource>() with instance(SqliteDataSource.open("path/to/file")) ①
}
```

① Instance is used with parenthesis: it is not given a function, but an instance.

Tagged bindings

All bindings can be tagged to allow you to bind different instances of the same type.

Example: different Dice bindings

```
val kodein = Kodein {
    bind<Dice>() with provider { RandomDice(6) } ①
    bind<Dice>("DnD10") with provider { RandomDice(10) } ②
    bind<Dice>("DnD20") with provider { RandomDice(20) } ②
}
```

① Default binding (with no tag)

② Bindings with tags

You can have multiple bindings of the same type, as long as they are bound with different tags. You can have only one binding of a certain type with no tag.

The tag is of type `Any`, it does not have to be a `String`.

Constant binding

It is often useful to bind "configuration" constants. Constants are always tagged.

Example: two constants

```
val kodein = Kodein {
    constant("maxThread") with 8 ①
    constant("serverURL") with "https://my.server.url" ①
}
```

① Note the absence of curly braces: it is not given a function, but an instance.

You should only use constant bindings for very simple types without inheritance or interface (e.g. primitive types and data classes).

Direct binding

Sometimes, it may seem overkill to specify the type to `bind` if you are binding the same type as you are creating.

For this use case, you can transform any `bind<Type>() with scope` to `bind() from scope`.

Example: direct bindings

```
val kodein = Kodein {
    bind() from singleton { RandomDice(6) }
    bind("DnD20") from provider { RandomDice(20) }
    bind() from instance(SqliteDataSource.open("path/to/file"))
}
```

This should be used with care as binding a concrete class and, therefore, having concrete dependencies is an *anti-pattern* that later prevents modularisation and mocking / testing.

In the case of generic types, the bound type will be the specialized type,
e.g. `bind() from singleton { listOf(1, 2, 3, 4) }` registers the binding to `List<Int>`.

Transitive dependency

With those lazily instantiated dependencies, a dependency (very) often needs another dependency. Such object can have their dependencies passed to their constructor. Thanks to Kotlin's killer type inference engine, Kodein makes retrieval of transitive dependencies really easy.

Example: a class that needs transitive dependencies

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

It is really easy to bind `RandomDice` with its transitive dependencies, by simply using `instance()` or `instance(tag)`.

Example: bindings of Dice and of its transitive dependencies

```
val kodein = Kodein {
    bind<Dice>() with singleton { Dice(instance(), instance("max")) } ①

    bind<Random>() with provider { SecureRandom() } ②
    constant("max") with 5 ②
}
```

① Binding of `Dice`. It gets its transitive dependencies by using `instance()` and `instance(tag)`.

② Bindings of `Dice` transitive dependencies.

The order in which the bindings are declared has **no importance whatsoever**.

You can, of course, also use the functions `provider()`, `provider(tag)`, `factory()` and `factory(tag)`,

Finally, you can also pass the `kodein` object to the class so it can itself use the Kodein object to retrieve its own dependencies.

Example: bindings of Manager that is responsible for retrieving its own dependencies

```
val kodein = Kodein {
    bind<Manager>() with singleton { ManagerImpl(kodein) } ①
}
```

① ManagerImpl is given a Kodein instance.

Bindings separation

Modules

Kodein allows you to export your bindings in modules. It is very useful to have separate modules defining their own bindings instead of having only one central binding definition. A module is an object that you can construct the exact same way as you construct a Kodein instance.

Example: a simple module

```
val apiModule = Kodein.Module {
    bind<API>() with singleton { APIImpl() }
    /* other bindings */
}
```

Then, in your Kodein binding block:

Example: imports the module

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

Modules are **definitions**, they will re-declare their bindings in each kodein instance you use. If you create a module that defines a singleton and import that module into two different kodein instances, then the singleton object will exist twice: once in each kodein instance.

Extension (composition)

Kodein allows you to create a new kodein instance by extending an existing one.

Example: extends an already existing Kodein instance

```
val subKodein = Kodein {
    extend(appKodein)
    /* other bindings */
}
```

This **preserves scopes**, meaning that a singleton in the parent Kodein will continue to exist only once. Both parent and child Kodein objects will give the same instance.

Overriding

By default, overriding a binding is not allowed in Kodein. That is because accidentally binding twice the same (class,tag) to different instances/providers/factories can cause real headaches to debug.

However, when intended, it can be really interesting to override a binding, especially when creating a testing environment. You can override an existing binding by specifying explicitly that it is an override.

Example: binds twice the same type, the second time explitly specifying an override

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

By default, modules are not allowed to override, **even explicitly**. You can allow a module to override some of your bindings when you import it (the same goes for extension):

Example: imports a module and giving it the right to override existing bindings.

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

The bindings in the module still need to specify explicitly the overrides.

Sometimes, you just want to define bindings without knowing if you are actually overriding a previous binding or defining a new. Those cases should be rare and you should know what you are doing.

Example: declaring a module in which each binding may or may not override existing bindings.

```
val testModule = Kodein.Module(allowSilentOverride = true) {
    bind<EmailClient>() with singleton { MockEmailClient() } ①
}
```

① Maybe adding a new binding, maybe overriding an existing, who knows?

If you want to access an instance retrieved by the overridden binding, you can use overriddenInstance. This is useful if you want to "enhance" a binding (for example, using the decorator pattern).

Example: declaring a module in which each binding may or may not override existing bindings.

```
val testModule = Kodein.Module {
    bind<Logger>(overrides = true) with singleton { FileLoggerWrapper("path/to/file",
overriddenInstance()) } ①
}
```

① `overriddenInstance()` will return the `Logger` instance retrieved by the overridden binding.

Dependency retrieval

Example bindings that are used throughout the chapter:

```
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"
}
```

Retrieval rules

When retrieving a dependency, the following rules apply:

- A dependency bound with a `factory` can only be retrieved as a factory method: `(A) → T`.
- A dependency bound with a `provider`, an `instance`, a `singleton` or a `constant` can be retrieved:
  - as a provider method: `() → T`
  - as an instance: `T`

Via Kodein methods

On a Kodein object

You can retrieve a dependency via a Kodein instance.

Example: retrieving bindings

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

When using a provider, whether the provider will give each time a new instance or the same depends on the binding scope.

When asking for a type that was not bound, a `Kodein.NotFoundException` will be thrown.

If you're not sure (or simply don't know) if the type has been bound, you can use `*OrNull` methods.

Example: retrieving bindings that may not have been bound

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

You can retrieve a provider or an instance from a factory bound type by using `with` (this is called *currying*).

Example: currying factories

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

In a Kodein aware class

You can have classes that implement the interface `KodeinAware`. Doing so has the benefit of having a simpler syntax for retrieval.

Example: a kodeinAware class

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

All methods that are available to `Kodein` are available to a `KodeinAware` class.

Via lazy properties

Lazy properties allow you to resolve the dependency upon first access.

Example: retrieving lazy properties

```
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")
}
```

`kodein.lazy.factoryOrNull`, `kodein.lazy.providerOrNull` and `kodein.lazy.instanceOrNull` are also available.

You can curry factories and retrieve a lazy property with the same `lazy` access.

Example: retrieving lazy curried factory properties

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

If you don't know yet the parameter to curry the factory with, you can pass a lambda. That way, the parameter will be fetched only when needed.

Example: retrieving lazy curried factory properties with lazy parameters

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

Via an injector

On an injector object

An injector is an object that you can use to inject all dependency properties in an object.

This allows your object to:

- Retrieve all its injected dependencies at once;
- Declare its dependencies without a Kodein instance.

Example: retrieving properties via an injector

```
class Controller() {
    private val injector = KodeinInjector() ①

    private val diceFactory: (Int) -> Dice by injector.factory() ②
    private val dataSource: DataSource by injector.instance() ②
    private val randomProvider: () -> Random by injector.provider() ②
    private val answerConstant: String by injector.instance("answer") ②

    private val kodein by injector.kodein() ③

    fun whenReady(kodein: Kodein) = injector.inject(kodein) ④
}
```

① Creating an injector

② Creating lazy properties.

③ Creating a lazy Kodein that will be available after injection.

④ Injecting all properties created by the injector.

If you try to access a property created by an injector **before** calling `injector.inject(kodein)`, a `KodeinInjector.UninjectedException` will be thrown.

`injector.factoryOrNull`, `injector.providerOrNull` and `injector.instanceOrNull` are also available.

As usual, you can curry factories by using `with`.

Example: creating curried factory properties

```
private val sixSideDiceProvider: () -> Dice by injector.with(6).provider()
private val tenSideDiceProvider: Dice by injector.with(10).instance()
```

In a Kodein injected class

You can have classes that implement the interface `KodeinInjected`. Doing so has the benefit of having a simpler syntax for injection.

Example: a kodeinInjected class

```
class MyManager() : KodeinInjected {
    override val injector = KodeinInjector()

    val ds: DataSource by instance()
}
```

All methods that are available to `KodeinInjector` are available to a `KodeinInjected` class.

Via a lazy Kodein

On a LazyKodein object

Sometimes, you don't directly have access to a Kodein instance. In these cases, if you don't want to use an injector, you can use `LazyKodein`.

Example: retrieving properties via an injector

```
class Controller() {
    private val kodein = LazyKodein { /* code to access a Kodein instance */ } ①

    private val diceFactory: (Int) -> Dice by kodein.factory() ②
    private val answerConstant: String by kodein.instance("answer") ②

    fun someFunction() {
        val dataSource: DataSource = kodein().instance() ③
    }
}
```

① Note the usage of `=` and not `by`.

② Creating lazy properties (I am using a `LazyKodein`, not `Kodein` instance).

③ To access a `Kodein` instance, I use `kodein()`.

You can create a LazyKodein with `Kodein.lazy`. When doing so, even the bindings will be declared only when the first retrieval happens.

Example: using Kodein.lazy

```
val kodein = Kodein.lazy { ①
    println("doing bindings")
    bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }
}

class Controller() {
    val ds: DataSource by kodein.instance()

    fun someFunction() {
        ds.open() ②
    }
}
```

① The `kodein` object is of type `LazyKodein`, not `Kodein`.

② Only there will "doing bindings" will be printed.

In a lazy Kodein aware class

You can have classes that implement the interface `LazyKodeinAware`. Doing so has the benefit of having a simpler syntax for lazy property creation.

Example: a LazykodeinAware class

```
class MyManager() : LazyKodeinAware {
    override val kodein = LazyKodein { /* code to access a Kodein instance */ }

    val ds: DataSource by instance()
}
```

All methods that are available to `LazyKodein` are available to a `LazyKodeinAware` class.

Class factories (such as loggers)

Sometimes you need to retrieve objects that are dependent to the class of the object whose retrieval is for.
The most obvious example is loggers: you need loggers that will print the name of the class name of *the class they are in*.

First, you need to declare a binding to a factory that takes a `Class` as argument.

Example: binding a logger

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

Then, you can retrieve such bound types by using `withClassOf`.

Example: retrieving a logger

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

If you are using a Kodein aware class, a Kodein injected class or a lazy Kodein aware class, then it's even easier: simply use `withClass`.

Example: retrieving a logger in a KodeinAware class

```
class MyManager(override val kodein: Kodein): KodeinAware {
    val logger: Logger = withClass().instance()
}
```

You can use `withClass` for factories that take a `Class<*>` as parameter, and `withKClass` for factories that take a `KClass<*>` as parameter.

In Java

While Kodein does not allow you to declare modules or dependencies in Java, it does allow you to retrieve dependencies using a Java friendly API. Simply give `kodein.typed` to your Java classes, and you can use Kodein in Java:

Example: using Kodein in Java

```
public class JavaClass {
    private final Function1<Integer, Dice> diceFactory;
    private final Datasource dataSource;
    private final Function0<Random> randomProvider;
    private final String answerConstant;

    public JavaClass(TKodein kodein) {
        diceFactory = kodein.factory(Integer.class, Dice.class);
        dataSource = kodein.instance(Datasource.class);
        randomProvider = kodein.provider(Random.class);
        answerConstant = kodein.instance(String.class, "answer");
    }
}
```

Remember that Java is subject to type erasure. Therefore, if you registered a generic Class binding such as `bind<List<String>>()`, in order to retrieve it you have to use `TypeReference` to circumvent Java's type erasure.

Example: using TypeReference in Java

```
class JavaClass {
    private final List<String> list;

    public JavaClass(TKodein kodein) {
        list = kodein.instance(new TypeReference<List<String>>(){});
    }
}
```

Configurable Kodein

Maybe you want a Kodein instance that you can pass around and have different sections of your code configure its bindings.

Configurable Kodein is a Kodein extension that is not proposed *by default*, this paradigm is in a separate module.

Using or not using this is a matter of taste and is neither recommended nor discouraged.

Example creating, configuring and using a `ConfigurableKodein`.

```
fun test() {
    val kodein = ConfigurableKodein()

    kodein.addModule(apiModule)
    kodein.addModule(dbModule)

    val ds: DataSource = kodein.instance()
}
```

Install

With Maven

```
<dependency>
    <groupId>com.github.salomonbrys.kodein</groupId>
    <artifactId>kodein-conf</artifactId>
    <version>3.3.0</version>
</dependency>
```

Do not remove the "kodein" (or "kodein-erased") dependency. Both dependencies must be declared.

With Gradle

```
compile 'com.github.salomonbrys.kodein:kodein-conf:3.3.0'
```

Do not remove the "kodein" (or "kodein-erased") dependency. Both dependencies must be declared.

Configuring

You can import modules, extend kodein objects, or add bindings inside this `ConfigurableKodein` using `addImport`, `addExtend` and `addConfig`.

Example: adding a module inside the global Kodein

```
fun test() {
    val kodein = ConfigurableKodein()

    kodein.addModule(aModule)
    kodein.addExtend(otherKodein)

    kodein.addConfig {
        bind<Dice>() with provider { RandomDice(0, 5) }
        bind<DataSource>() with singleton { SqliteDS.open("path/to/file") }
    }
}
```

The Kodein object will effectively be constructed on first retrieval. Once it is constructed, trying to configure it will throw an `IllegalStateException`.

Retrieving

You can use a `ConfigurableKodein` object like any `Kodein` object.

Once you have retrieved the first value with a `ConfigurableKodein`, trying to configure it will throw an `IllegalStateException`.

Mutating

A `ConfigurableKodein` can be mutable.

Example: Creating a mutable ConfigurableKodein

```
val kodein = ConfigurableKodein(mutable = true)
```

Using a mutable `ConfigurableKodein` can lead to **very bad code practice** and **very difficult bugs**.
Therefore, using a mutable `ConfigurableKodein` **IS discouraged**.
Note that every time a `ConfigurableKodein` is mutated, its cache is entirely flushed, meaning that it has a real impact on optimization!
Please use the mutating feature only if you truly need it, know what you're doing, and see no other way.

A mutable `ConfigurableKodein` can be configured even *after first retrieval*.

Example: mutating a mutable ConfigurableKodein

```
fun test() {
    val kodein = ConfigurableKodein(mutable = true)

    kodein.addModule(aModule)

    val ds: DataSource = kodein.instance()

    kodein.addModule(anotherModule) ①
}
```

① This would have failed if the ConfigurableKodein was not mutable.

You can also use `clear` to remove all bindings.

The god complex: One True Kodein

Sometimes, you want one static Kodein for your entire application. E.g. you don't want to have to hold & pass a Kodein instance throughout your application.

For these cases, the `kodein-conf` module proposes a static `Kodein.global` instance.

Example creating, configuring and using the global one true Kodein.

```
fun test() {
    kodein.global.addModule(apiModule)
    kodein.global.addModule(dbModule)

    val ds: DataSource = kodein.global.instance()
}
```

Just like any `ConfigurableKodein`, `Kodein.global` must be configured **before** being used for retrieval, or an `IllegalStateException` will be thrown. It is possible to set `Kodein.global` to be mutable by setting `Kodein.global.mutable = true` but it **must** be set **before** any retrieval!

Being globally aware

You can use the `GlobalKodeinAware` interface that needs no implementation to be aware of the global kodein.

Example: a KodeinGlobalAware class

```
class MyManager() : KodeinGlobalAware {
    val ds: DataSource = instance()
}
```

Easy :)

Android

Kodein does work on Android!

You can use Kodein as-is in your Android project or use the very small util library `kodein-android`.

Install

How to use `kodein-android`:

1. Add this line in your `dependencies` block in your application `build.gradle` file:

   ```
   compile 'com.github.salomonbrys.kodein:kodein:3.3.0'
   compile 'com.github.salomonbrys.kodein:kodein-android:3.3.0'
   ```

   

   Both `kodein` and `kodein-android` dependencies must be declared.

   

   If you are using `kodein-erased`, then you must declare both dependencies : `kodein-erased` and `kodein-android` (but not `kodein`).

2. If you are using Proguard, you need to add the following line to your proguard configuration file:

   ```
   -keepattributes Signature
   ```

3. Declare the dependency bindings in the Android `Application`, having it implements `KodeinAware`.

Example: an Android Application class that implements KodeinAware

```
class MyApp : Application(), KodeinAware {
    override val kodein by Kodein.lazy { ①
        /* bindings */
    }
}
```

① Using Kodein.lazy allows you to access the `Context` at binding time.

Don't forget to declare the Application in the `AndroidManifest.xml` file!

4. In your Activities, Fragments, and other context aware android classes, retrieve dependencies!

There are different ways to access a Kodein instance and your dependencies.

Bindings & retrieval

Using a LazyKodein

`appKodein` is a property that will work in your context aware Android classes provided that your Application implements `KodeinAware`. From it, you can construct a `LazyKodein`.

Example: retrieving dependencies with LazyKodein in Android

```
class MyActivity : Activity() {
    val kodein = LazyKodein(appKodein)

    val diceProvider: () -> Dice by kodein.provider() ①

    override fun onCreate(savedInstanceState: Bundle?) {
        val random: Random = kodein().instance()   ②
    }
}
```

① `kodein` without parenthesis: creates a lazy property.

② `kodein` with parenthesis: gets the instance.

You cannot use `kodein` with parenthesis and access the Kodein instance while the activity is not initialized by Android.

Using an injector

Using an injector allows you to resolve all dependencies in `onCreate`, reducing the cost of dependency first-access (but more processing happening in `onCreate`).

Example: retrieving dependencies with an injector in Android

```
class MyActivity : Activity() {
    private val injector = KodeinInjector()

    val random: Random by injector.instance()

    override fun onCreate(savedInstanceState: Bundle?) {
        injector.inject(appKodein())
    }
}
```

Using this approach has an important advantage: as all dependencies are retrieved in `onCreate`, you can be sure that all your dependencies have correctly been retrieved, meaning that there were no non-declared dependency.
If you only use `instance` (no `provider` or `factory`), you can also be sure that there were no dependency loop.

Being aware in Android

`appKodein` cannot be accessed before an activity has been created, before a fragment has been attached, and so on. Because of this, it is not recommended to use `KodeinAware` in Android. Prefer using `LazyKodeinAware` or `KodeinInjected`.

Example: retrieving dependencies with LazyKodeinAware in Android

```
class MyActivity : Activity(), LazyKodeinAware {
    override val kodein = LazyKodein(appKodein)

    val diceProvider: () -> Dice by provider()
}
```

Example: retrieving dependencies with KodeinInjected in Android

```
class MyActivity : Activity(), KodeinInjected {
    override val injector = KodeinInjector()

    val random: Random by instance()

    override fun onCreate(savedInstanceState: Bundle?) {
        inject(appKodein())
    }
}
```

Bootstrapping Kodein on Android

To easily setup Kodein with your Android app, you can use the Android Injectors. They make it

simple to creating activities, fragments, services, and broadcast recivers that work with Kodein out of the box.

This method allows for deep Kodein integration into you Android components. You can choose to use Kodein without it.

There are two ways to use them depending on your needs; inheritance based and interface based.

Both methods provide you with:

- a `KodeinInjector` (through the `injector` property)
- a binding for `KodeinInjected` (which is the instance of your class)
- local bindings (bindings for that specific instance)
  - One example is a `KodeinActivity` or `ActivityInjector` will bind `Context` and `Activity` to itself, and `FragmentManager`, `LoaderManager`, and `LayoutInflater` to its instances of those classes.
- scope management (removing this component from the scope when it is destroyed so there are no memory leaks)
- the ability to override previously defined bindings

Don't forget to use an `Application` that is `KodeinAware`:

Example: setup kodein in the application

```
class MyApplication : Application(), KodeinAware {
    override val kodein by Kodein.lazy {
        import(autoAndroidModule(this@MyApplication))
        bind<String>("log-tag") with instance("MyApplication")
    }
}
```

Inheritance Based

Kodein provides base classes for the following Android components:

- Activity (`KodeinActivity`)
- FragmentActivity (`KodeinFragmentActivity`)
- AppCompatActivity (`KodeinAppCompatActivity`)
- Fragment (`KodeinFragment`)
- Support v4 Fragment (`KodeinSupportFragment`)
- Service (`KodeinService`)
- IntentService (`KodeinIntentService`)
- BroadcastReceiver (`KodeinBroadcastReceiver`)

All it takes to get started is to extend one of those classes, and you're ready to start injecting. Let's

see an example.

Example: using KodeinActivity to make injecting easier

```
class MyActivity : KodeinActivity() { ①
    private val logTag: String by instance("log-tag") ②
    private val app: Application by injector.instance() ③

    override fun provideOverridingModule() = Kodein.Module { ④
        bind<MyActivity>() with instance(this@MyActivity)
        bind<String>("log-tag", overrides = true) with instance("MyActivity")
    }

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

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

① Extending `KodeinActivity` provides us with a `KodeinInjector` and takes care of its lifecycle

② `KodeinActivity` implements `KodeinInjected` so we don't need to use the `injector` property if we don't want to

③ We can also use the `injector` if we want to

④ `provideOverridingModule` allows us to override bindings specified higher up in the dependency tree (for example, we override the "log-tag" `String` binding defined in `MyApplication`)

`KodeinActivity`, `KodeinFragmentActivity`, and `KodeinAppCompatActivity` will internally initialize their injector before they call `super.onCreate` (see the issue that necessitates this).

KodeinBroadcastReceiver

Because of how the injector's lifecycle is managed for a `KodeinBroadcastReceiver`, subclasses should override `onBroadcastReceived(Context, Intent)` instead of `onReceive(Context, Intent)`.

Interface Based

Kodein also provides a set of interfaces that provide the same functionality as the inheritance based method. The only difference is that the injector lifecycle must be managed. In almost every case, this can be accomplished by simply calling `initializeInjector` immediately after `onCreate` and `destroyInjector` immediately after `onDestroy`.

These are provided so that you can extend non-framework components if needed, because the JVM does not support multiple class inheritance.

The interfaces are:

- ActivityInjector
- FragmentActivityInjector
- AppCompatActivityInjector
- FragmentInjector
- SupportFragmentInjector
- ServiceInjector
- IntentServiceInjector
- BroadcastReceiverInjector

Example: using FragmentInjector to make injecting easier

```
class MyFragment : CustomFragment(), FragmentInjector { ①
    override val injector: KodeinInjector = KodeinInjector() ②

    private val logTag: String by instance("log-tag") ③
    private val app: Application by injector.instance() ④

    override fun provideOverridingModule() = Kodein.Module { ⑤
        bind<MyFragment>() with instance(this@MyFragment)
        bind<String>("log-tag", overrides = true) with instance("MyFragment")
    }

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

        initializeInjector() ⑥

        Log.i(logTag, "Calling onCreate from MainActivity in
${app.applicationInfo.className}")
    }

    override fun onDestroy() {
        super.onDestroy()
        destroyInjector() ⑦
    }
}
```

① Because we extends `CustomFragment` we cannot extend `KodeinFragment` so instead we implement `FragmentInjector`

② We have to provide an injector (typically all that entails is just creating a new instance of `KodeinInjector`)

③ `FragmentInjector` implements `KodeinInjected` so we don't need to use the `injector` property if we don't want to

④ We can also use the `injector` if we want to

⑤ `provideOverridingModule` allows us to override bindings specified higher up in the dependency tree (for example, we override the "log-tag" `String` binding defined in `MyApplication`)

⑥ Since we have to manage the injector's lifecycle we initialize it when the fragment is initialized; in `onCreate`

⑦ Since we have to manage the injector's lifecycle we destroy it when the fragment is destroyedl in `onDestroy`

When using `ActivityInjector`, `FragmentActivityInjector`, or `AppCompatActivityInjector` it is suggested to call `initializeInjector` before `super.onCreate` (see the issue that necessitates this).

BroadcastReceiverInjector

`BroadcastReceiverInjector` should be used like this:

Example: implementing BroadcastReceiverInjector

```
class MyBroadcastReceiver : CustomBroadcastReceiver(), BroadcastReceiverInjector {
    override val injector: KodeinInjector = KodeinInjector()

    final override var context: Context? = null

    final override fun onReceive(context: Context, intent: Intent) {
      super.onReceive(context, intent)

      this.context = context ①
      initializeInjector()
      // do something
      destroyInjector()
    }
}
```

① It is necessary to set `BroadcastReceiverInjector.context` before calling `initializeInjector`

Fragments

The parent of any `KodeinFragment`, `KodeinSupportFragment`, `FragmentInjector`, or `SupportFragmentInjector` **must** be `KodeinInjected`. This means a `parentFragment` **must** be one of `KodeinFragment`, `KodeinSupportFragment`, `FragmentInjector`, or `SupportFragmentInjector`. If there is no `parentFragment`, the `Activity` of the `Fragment` **must** be one of `KodeinActivity`, `KodeinFragmentActivity`, `KodeinAppCompatActivity`, `ActivityInjector`, `FragmentActivityInjector`, or `AppCompatActivityInjector`.

Inside a fragment, you can retrive a `LayoutInflater` using the tag `ACTIVITY_LAYOUT_INFLATER`. Using the tag will use a more optimized way of retrieving the `LayoutInflater`.

Android module

Kodein-Android proposes a module that enables easy retrieval, with a context, of a lot of standard android services.

This module is absolutely **optional**, you are free to use it or leave it ;).

Example: importing the android module

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

You can see everything that this module proposes in the AndroidModule.kt file.

To retrieve instances of bindings defined in this module, you can use `withContext`.

Example: using kodein to retrieve a LayoutInflater

```
class MyActivity : Activity(), LazyKodeinAware {
    override val kodein = LazyKodein(appKodein)

    val inflater: LayoutInflater by withContext(this).instance()
}
```

There is also an "auto" version of the module.

Example: importing the android module

```
class MyApplication : Application(), KodeinAware {
  override val kodein by Kodein.lazy {
    import(autoAndroidModule(this@MyApplication))
    /* other bindings */
  }
}
```

Retrieving instances of bindings from the autoAndroidModule does not require a `Context`.

Example: using kodein to retrieve a LayoutInflater

```
class MyActivity : Activity(), LazyKodeinAware {
    override val kodein = LazyKodein(appKodein)

    val inflater: LayoutInflater by instance()
}
```

You can see everything that this module proposes in the AndroidModule.kt file.

Android scopes

The context scope

There are times where you need an object to be a singleton, but only during the lifetime of a `Context`. You can use the `contextSingleton` scope to achieve this.

The context scope should be used when a binding could apply to either an `Activity` or a `Service`. When a binding is exclusively for an `Activity` or a `Service`, the activity or service scope should be used instead.

Example: using the context scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidContextScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the context the object is being created for.

To retrieve an object bound in the context scope, you need to inject a factory that takes the context as a parameter.

Example: retrieving a context scoped singleton

```
val logger: Logger = kodein.with(context).instance()
val sameLogger: Logger = kodein.with(context).instance() // this will be the same
object as logger
val otherLogger: Logger = kodein.with(otherContext).instance() // this will be a
different object than logger
```

The activity and service scope are special cases of the context scope. The bindings returned for an `Activity` or `Service` object from the context scope will be the same one returned for that object from the activity or service scope

The activity scope

If you want a singleton that lives only during the lifecycle of a specific `Activity`, and not any `Context`, you can use the activity scope.

Example: using the activity scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidActivityScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the activity the object is being created for.

As with the context scope, to retrieve objects bound in the activity scope, you need to inject a factory which takes the activity as a parameter.

Example: retrieving an activity scoped singleton

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

The auto activity scope

If you don't want to be required to explicitly provide an activity instance to inject your objects, you can use the "auto activity scope".

Example: using the auto activity scope

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

Example: retrieving an auto activity scoped singleton

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

In your `Application` class, in the `onCreate` method, you **must** add this line:

Example: registering kodein's lifecycle manager to enable the auto activity scope to work

```
class MyApplication : Application {
    override fun onCreate() {

registerActivityLifecycleCallbacks(androidActivityScope.lifecycleManage
r) ①
    }
}
```

① `androidActivityScope.lifecycleManager` is what enables the auto scope to work.

Objects that are bound in the auto `androidActivityScope` will always be injected according to **the last displayed activity**.

The service scope

If you want a singleton that lives only during the lifecycle of a specific `Service`, and not any `Context`, you can use the service scope.

Example: using the service scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidServiceScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the service the object is being created for.

As with the context scope, to retrieve objects bound in the service scope, you need to inject a factory which takes the service as a parameter.

Example: retrieving a service scoped singleton

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

The broadcast receiver scope

If you want a singleton that lives only during the lifecycle of a specific `BroadcastReceiver`, you can use the broadcast receiver scope.

Example: using the broadcast receiver scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidBroadcastReceiverScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the broadcast receiver the object is being created for.

To retrieve objects bound in the broadcast receiver scope, you need to inject a factory which takes the broadcast receiver as a parameter.

Example: retrieving a broadcast receiver scoped singleton

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

The fragment (and support fragment) scope

If you want a singleton that lives only during the lifecycle of a specific `Fragment`, you can use the fragment scope (or support fragment scope if you are using support lib fragments).

Example: using the fragment scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidFragmentScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the fragment the object is being created for.

Example: using the support fragment scope

```
val kodein = Kodein {
    bind<Logger>() with scopedSingleton(androidSupportFragmentScope) {
LogManager.getNamedLogger(it.localClassName) } ①
}
```

① `it` is the support fragment the object is being created for.

To retrieve objects bound in the fragment scope (or support fragment scope), you need to inject a factory which takes the fragment as a parameter.

Example: retrieving a fragment scoped singleton

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

Example: retrieving a support fragment scoped singleton

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

Android example project

Have a look at the Android demo project!

Debugging

Print bindings

You can easily print bindings with `println(kodein.container.bindings.description)`.

Here's an example of what this prints:

An example of kodein.container.bindings.description:

```
        bind<Dice>() with factory { Int -> RandomDice }
        bind<DataSource>() with singleton { SQLiteDataSource }
        bind<Random>() with provider { SecureRandom }
        bind<String>("answer") with instance ( Int )
```

As you can see, it's really easy to understand which type with which tag is bound to which implementation inside which scope.

Descriptions prints type names in a "kotlin-esque" way. Because Kodein does not depends on `kotlin-reflect`, it uses java `Type` objects that do not contains nullability information. As such, the type display does not include nullability. Still, it's easier to read `List<*>` than `List<? extends Object>`.

Recursive dependency loop

When it detects a recursive dependency, Kodein will throw a `Kodein.DependencyLoopException`. The message of the exception explains how the loop happened.

An example of recursive dependency loop:

```
com.github.salomonbrys.kodein.Kodein$DependencyLoopException: Dependency recursion:
        ┌──> bind<com.test.A>()
        ├──> bind<com.test.B>() ①
        ├──> bind<com.test.C>("yay") ②
        └──> bind<com.test.A>() ③
```

① `com.test.A` depends on `com.test.B`

② `com.test.B` depends on `com.test.C` with the tag "Yay"

③ `com.test.C` with the tag "Yay" depends on `com.test.A`, **we have found the dependency loop!**.

Performance

The type erasure problem

By default, Kodein is immune to type erasure, meaning that `bind<List<String>>()` and `bind<List<Int>>()` will represent two different bindings.
Similarly, `kodein.instance<List<String>>()` and `kodein.instance<List<Int>>()` will yield two different list.

To be erasure immune, kodein relies heavily on the `genericToken` function, **which is real a performance pitfall**.

To improve performance, you can use the `erased*` set of kodein functions, which are faster, but do suffer from type erasure!

Yes, #perfmatters. However, the humble opinion of the author is that:

- There is a balance to be found between performance, readability, security and debuggability.
- Optimisation is important in critical path, not *everywhere*.

Therefore, please make sure that, using the `erased*` function set is right for your use case, before blindly using it ;).

Using generic and erased function forms

Each kodein function that handles a type exists in two form: `generic` and `erased`.
For example, the `kodein.instance` function exists as `kodein.genericInstance` and `kodein.erasedInstance`.

By default, all type functions are aliases to their "generic*" counterpart.
For example, the `kodein.instance` function is an alias to `kodein.genericInstance`

So, when you know that you inject a type that is **not generic**, you can use `kodein.erasedInstance`:

Example: injecting a non generic object using the optimized method

```
val session: HttpSession = kodein.erasedInstance()
```

Using erased by default

If you know what you are doing, and ***why*** you are doing it, you can change the default methods to alias the `erased*` function set.

For this, you **must not** depend on the `kodein` module, and instead, depend on the `kodein-erased` module.

This means that, by default, `kodein.instance<List<String>>()` will look for an erased `List` binding.
Therefore, to bind a `List<String>` you **must** use `bindGeneric<List<String>>()`, and to retrieve it, you **must** use `kodein.genericInstance()`.

The `erased*` function set is located in a different package than the regular one: `com.github.salomonbrys.kodein.erased`.

Advanced use

OnReady callbacks

You can define callbacks to be called once the kodein instance is ready and all bindings are defined.
This can be useful to do some "starting" jobs.

Example: registering a callback at binding time

```
val appModule = Kodein.Module {
    import(engineModule)
    onReady {
        val engine = instance<Engine>()
        instance<Logger>().info("Starting engine version ${engine.version}")
        engine.start()
    }
}
```

Create your own scopes

Scoped singletons

Scoped singleton are singletons that are bound to a context and live while that context exists.

To define a scope that can contain scoped singleton, you must define an object that implements the `Scope` interface. This object will be responsible for providing a `ScopeRegistry` according to a context. It should always return the same `ScopeRegistry` when given the same context object. Standard ways of doing so is to use the `userData` property of the context, if is has one, or else to use a `WeakHashMap<C, ScopeRegistry>`.

To declare bindings in your scope, use `scopedSingleton`.

Example: defining a scope and binding a singleton inside it

```
object myScope: Scope<Request> { ①
    override fun getRegistry(context: Request): ScopeRegistry
            = context.userData.getOrPut("registry") { ScopeRegistry() } as
ScopeRegistry ②
}

val kodein = Kodein {
    bind<Logger>() with scopedSingleton(myScope) { LogManager.getNamedLogger(it.name)
} ③
}
```

① The scope's context type is `Request`.

② Creates a `ScopeRegistry` in the context `Request` if there is none.

③ `it` is the context. There will be at most one `Logger` per `Request` object.

To retrieve a scoped singleton bound type, you must retrieve **a factory** and then provide it the context.

Example: using a scope

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

Auto Scoped singletons

Scoped singletons are not always ideal since you need the context to retrieve any object. Sometimes, the context is static. For these times, you can use an "auto scoped singleton". An auto scoped singleton is responsible for fetching both the `ScopeRegistry` and the context.

To define an auto scope that can contain auto scoped singleton, you must define an object that implements the `AutoScope` interface.

To declare bindings in your scope, use `autoScopedSingleton`.

Example: defining an scope and binding a singleton inside it

```
object myScope: AutoScope<Request> { ①
    override fun getRegistry(context: Request): ScopeRegistry
            = context.userData.getOrPut("registry") { ScopeRegistry() } as
ScopeRegistry ②
    override fun getContext(): Request
            = StaticContext.getCurrentRequest() ③
}

val kodein = Kodein {
    bind<Logger>() with autoScopedSingleton(::myScope) {
LogManager.getNamedLogger(it.name) } // `it` is the context.
}
```

① The scope's context type is `Request`.

② Same as `Scope.getRegistry`.

③ Get the context from a static environment.

To retrieve an auto scoped singleton bound type, you can retrieve **a provider** or **an instance**.

Example: using an auto scope, without knowing it

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

If your auto scope does not depends on a context, and always yields the same `ScopeRegistry`, then it's very simple:

Example: defining a static auto scope

```
object myScope: AutoScope<Unit> {
    private val _registry = ScopeRegistry()
    override fun getRegistry(context: Unit) = _registry
    override fun getContext() = Unit
}
```

Factories

A factory function is an extension function to `Kodein.Builder` that returns a `Factory<A, T>`. You can use the `CFactory<A, T>` class for ease of use. If your scope is a provider scope (such as singleton), you can use the `CProvider<T>` class for ease of use. Have a look at existing scopes in the factories.kt file. The `singleton` scope is very easy to understand and is a good starting point.

Use the typed API

Accessing and using `kodein.typed` is not reserved to Java. You can use it in Kotlin to access an API that directly uses `Type`, `TypeToken` or `Class` objects.

In fact, most `Kodein` extension functions such as `kodein.instance<Type>()` are inline methods that proxy to this typed API.

When defining bindings, in the `Kodein.Builder`, you can access the `typed` property to bind factories to a `Type`, a `TypeToken` or a `Class`.

A `KodeinInjector` also provides a typed API, simply use `injector.typed`.

Use TypeToken

Kodein almost never uses java classes, because they are subject to type erasure. To get a real type, Kodein uses the `genericToken` function.

The `genericToken` function produces a `TypeToken` object, which only contains a type.

- `genericToken<String>().type` is the String's java `Class`.
- `genericToken<List<String>>.type` is a `ParameterizedType` describing exactly a `List<String>`.

`TypeToken` is a generic interface, that way, even if it's `type` is a weird type, functions that use a `TypeToken` (such as the typed API) can preserve type safety.

Bind the same type to different factories

Yeah, when I said earlier that "you can have multiple bindings of the same type, as long as they are bound with different tags", I lied. Because each binding is actually a *factory*, the bindings are not `([BindType], [Tag])` but actually `([BindType], [ArgType], [Tag])` (note that providers and singletons are bound as `([BindType], Unit, [Tag])`). This means that any combination of these three information can be bound to it's own factory, which in turns means that you can bind the same type without tagging to different factories.

Please be cautious when using this knowledge, as other less thorough readers may get confused with it.

Use the container API

The KodeinContainer is the sacred Kodein object that contains all bindings and is responsible for retrieval. You can access it with `kodein.container`. In it, each `Factory` is bound to a `Kodein.Key`.

In fact, all `Kodein.typed` functions are proxies to this container API.

When defining bindings, in the `Kodein.Builder`, you can access the `container` property to bind factories to a `Kodein.Key` or a `Kodein.Bind`.

Explore bindings

You can access a **copy** of the bindings map with `kodein.container.bindings`.
From this `Map<Kodein.Key, Factory<*, *>>`, you can explore all bindings, their keys and factories.

The bindings.kt file exposes several extension functions to this map that can be useful for exploring it.

API reference

The API reference can be found here!

Contributing

Contributions are very welcome and greatly appreciated! The great majority of pull requests are eventually merged.

To contribute, simply fork the project on Github, fix whatever is iching you, and submit a pull request!

I am sure that this documentation contains typos, inaccuracies and languages error (English is not my mother tongue). If you feel like enhancing this document, you can propose a pull request that modifies README3.adoc. (The documentation page is auto-generated from it).

Let's talk!

You've read so far?! **You're awesome!**
Why don't you drop by the Kodein Slack channel on Kotlin's Slack group?

Donate

Kodein is free to use for both non-profit and commercial use and always will be.

If you wish to show some support or appreciation to my work, you are free to **donate**!

This would be (of course) greatly appreciated but is by no means necessary to receive help or support, which I'll be happy to provide for free!