Dagger 2

Dependency Injection Framework

Overview

- **Why Dependency Injection**
- **About Dagger**
- **Basic Dagger Annotations:** Module, Component, Provides, Inject, Singleton, Binds
- **Subcomponents and Scopes**
- **Multibindings:** IntoSet, IntoMap
- **Dagger in Android:** @ContributesAndroidInjector, @AndroidInjector
- **Alternative DI Frameworks**

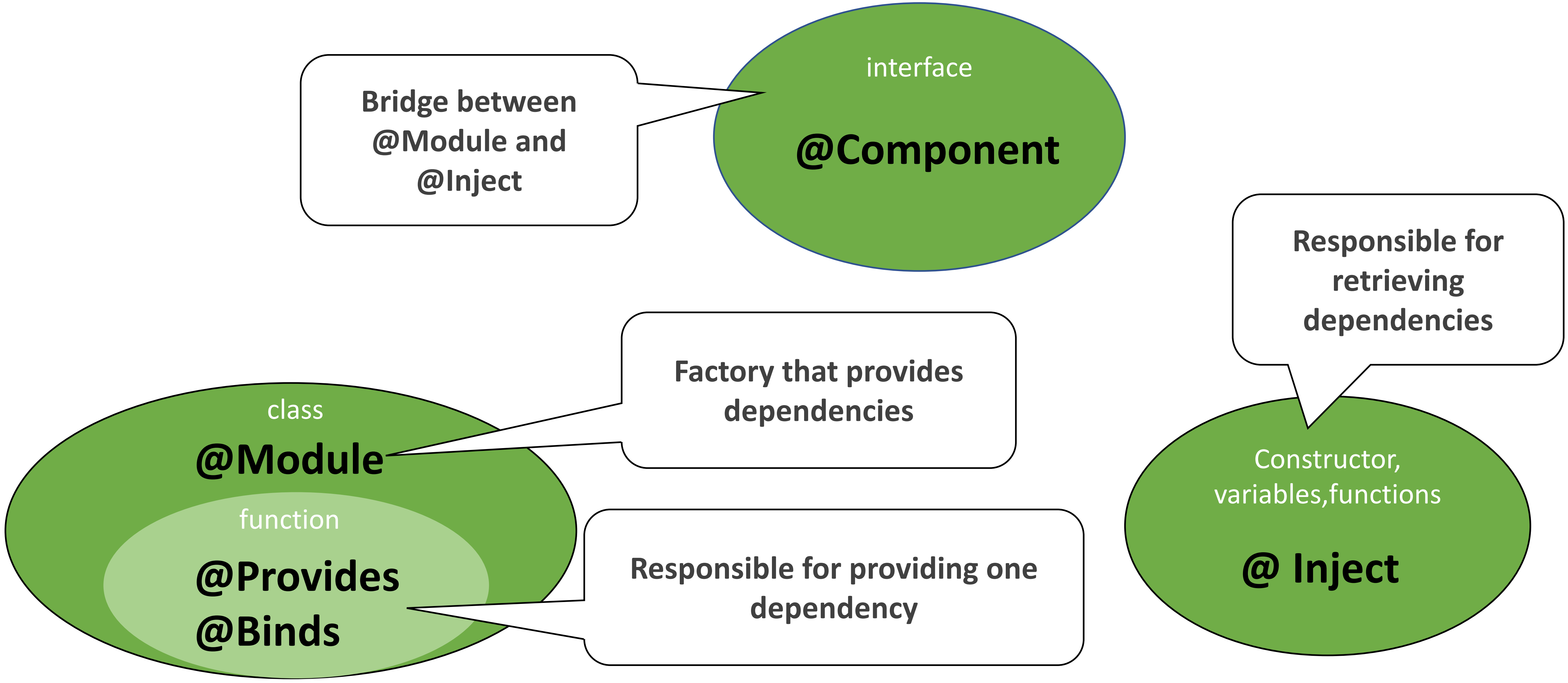
Why dependency Injection

- **Testability**
- **Structure**, especially in big Applications
- DI simplifies implementation of **scoping**
- Re-usable and interchangeable components

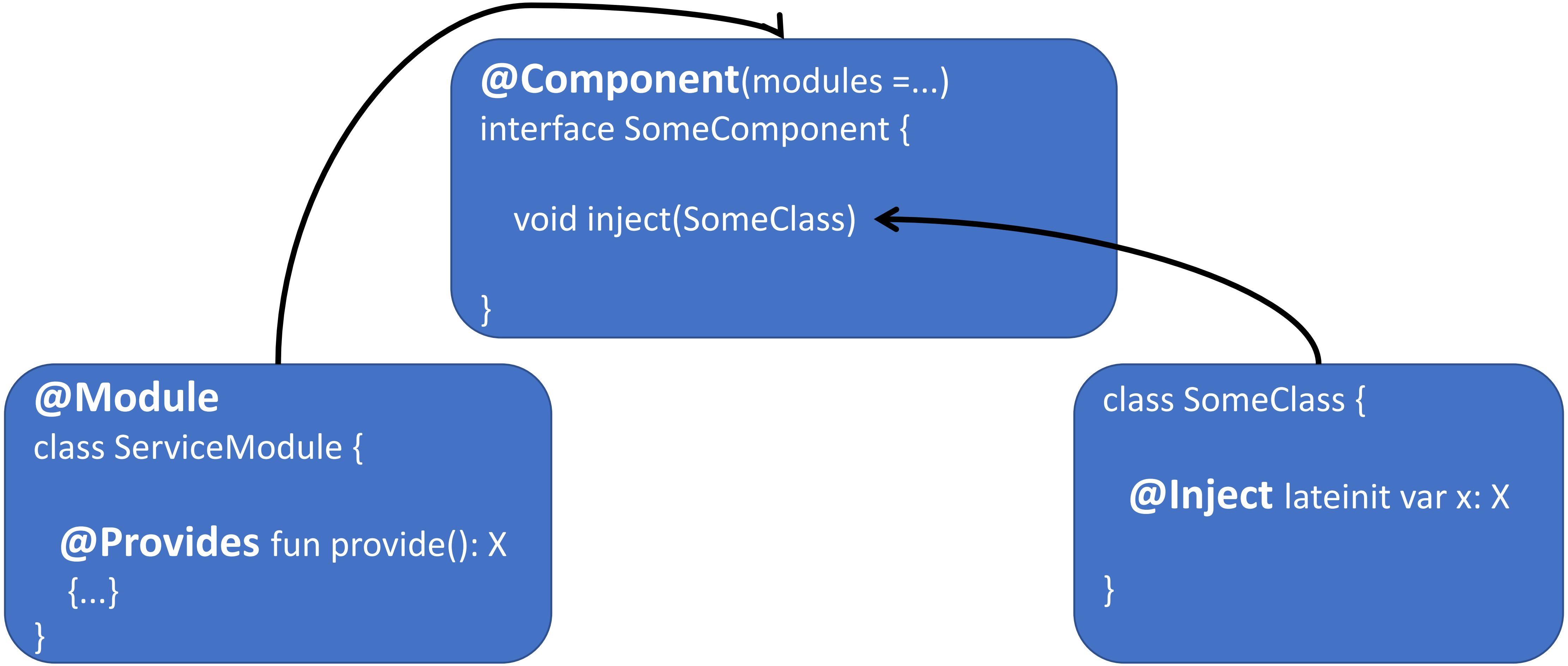
About Dagger 2

- Dependency framework
- Compile-time injection
- Simple, traceable and performant
- Scoped instances
- Owned by Google

Dagger Annotations


Bridge between @Module and @Inject
interface
@Component
Responsible for retrieving dependencies
Factory that provides dependencies
class
@Module
function
@Provides
@Binds
Responsible for providing one dependency
Constructor, variables,functions
@ Inject

@Component @Module @Inject



```
@Component(modules =...)
interface SomeComponent {

  void inject(SomeClass)

}
```

```
@Module
class ServiceModule {

  @Provides fun provide(): X
  {...}
}
```

```
class SomeClass {

  @Inject lateinit var x: X

}
```

Example - Module

Scoping

```
@Module
class ServiceModule{
    @Provides @Singleton
    fun provideService() = NetworkService();


    @Provides @Named("network-url")
    fun s(): String = "https://xxx"
}
```

Qualifier: for more dependencies of the same type

Example - Component

The Module defined in the previous slide

```
@Component(modules = arrayOf(ServiceModule::class))
interface AppComponentExample{
    fun inject(app: DaggerKotlinApp)
}
```

The class which uses the dependencies declared in the DaggerKotlinApp

Example - ServiceApp

```
class DaggerKotlinApp: Application(){

    @Inject lateinit var networkService: NetworkService


    override fun onCreate() {
        super.onCreate()
        DaggerAppComponentExample.
                builder().
                build().
                inject(this)
    }
}
```

Injected dependency (initialized in Module-class)

The @Component annotation generates a class called Dagger[ComponentName] when compiling

This generated DaggerComponent-class has a builder

The class retrieving dependencies (in this case the DaggerKotlinClass) can inject itself

@Binds

- @Binds is a simplification

```
@Binds
abstract fun bindLoginViewModel(loginViewModel: LoginViewModel): ViewModel
```

=

```
@Provides
fun provideLoginViewModel(): ViewModel {
      return LoginViewModel()
}
```

@Binds

- **The Module holding @Bind-Methods needs to be abstract**
- **@Binds is performant**
- **What if the module contains @Provides and @Binds?**
  - Option 1: make @Provides-methods static (how to do this in Kotlin?)
  - Option 2: Split the module

Module with @Binds and @Provides in Kotlin

```
@Module
abstract class BindProvideModule {


    @Module
    companion object {

        @JvmStatic
        @Provides
        fun provideNavigContr(vm: SomeViewModel) = NavigationController()
    }

    @Binds
    abstract fun bindViewModel(base: ViewModel): SomeViewModel
}
```

Behind the scenes creates a static method in the JVM

Dagger - Setup

1. Create a Module-class annotated with @Module
2. Instantiate all the necessary dependencies in the Module-class with @Provides / @Bind
3. Create an interface annotated with @Component and add the Module-classes in the annotation
4. In the @Component class one or more functions inject with one Argument: the class that uses the dependencies

Exercise 1

Branch: chapter_03_dagger_exercise1

Look for: «TODO:chapter_03_exercise1»

Splitting dependencies in different Modules

- Two ways to split dependencies:

declare all the modules in the component

```
@Component(modules =
arrayOf(ServiceModule::class,FragmentModule::class))
interface AppComponentExample{
```

Include a module as a submodule

```
@Module(includes = arrayOf(FragmentModule::class))
class ServiceModule{
```

Scopes and Subcomponents

- @Singleton: scope is over the whole application
- Custom scopes can be generated using **sub-components** or **dependent- components**

Dependent Component

Explanation with UserScope Example

Dependent Component

The component it depends on

```
@Component(dependencies = arrayOf(AppComponent::class),
modules = arrayOf(UserModule::class))
@UserScope
interface UserComponent {

    @Component.Builder
    interface Builder {
        fun appComponent(appComponent: AppComponent): Builder
        @BindsInstance
        fun user(user: User): Builder
        fun build(): UserComponent
    }
}
```

Annotation class defining the name of the scope

We overwrite the defualt Builder which Dagger would generate

The user is added to the graph with @BindsInstance

Dependent Component

```
@Component(modules = arrayOf(ServiceModule::class
interface AppComponentExample{
    fun inject(app: DaggerKotlinApp)
    fun provideService(): Service
}
```

The AppComponent needs to expose all dependencies, which are used by the dependent Component

Dependent Component

```
private fun createUserSession(user: User) {
    userComponent = DaggerUserComponent.builder()
            .appComponent(DaggerApplication.app.appComponent)
            .user(user)
            .build()
}
```

Here we can use the overwritten UserComponent-Builder

We need access of the AppComponent (the DaggerApplication exposes it in this example)

Subcomponent

Explanation with ActivityScope Example

Subcomponent

Annotation to create a Subcomponent

Annotation class defining the name of the scope

```
@ActivityScope
@Subcomponent(modules = arrayOf(LoginModule::class))
interface LoginComponent {

    fun inject(loginActivity: LoginActivity): LoginActivity

    @Subcomponent.Builder
    interface Builder {
        fun build(): LoginComponent
        @BindsInstance
        fun loginActivity(loginActivity: LoginActivity):
Builder
    }
}
```

Binds the LoginActivity to the graph with @BindsInstance

Subcomponent

```
@Component(modules = arrayOf(ServiceModule::class))
interface AppComponentExample{
    fun inject(app: DaggerKotlinApp)
    fun loginBuilder(): LoginComponent.Builder
}
```

The AppComponent needs to expose the Builder of the Subcomponent

Subcomponent

- Inside the LoginActivity:

```
override fun initDagger(appComponent: AppComponent) {
    loginComponent = appComponent
            .loginBuilder()
            .loginActivity(this)
            .build()
    loginComponent.inject(this)
}
```

The Builder for the LoginComponent is exposed in the Parent Component

Subcomponent vs Dependent Component

Subcomponent	Dependent Component
The Dagger- generated LoginComponentImpl is an inner class of the DaggerAppComponent	The Dagger generated DaggerUserComponent depends on the AppComponent
LoginComponentImpl can directly access any dependency of the AppComponent	DaggerUserComponent accesses dependencies using the AppComponent Interface
Use it if the two components are coupled with each other (like Application and Activity)	Use it if you have less coupling between the two components

Scope

- The Scope is defined as follows

```
@Scope
@Documented
@Retention(value = RetentionPolicy.RUNTIME)
annotation class UserScope
```

- The scope is defined by the lifetime of the Component.

Multibindings

Binds different instances of a type into a Set (@IntoSet, @ElementsIntoSet) or into a Map(@IntoMap)

@IntoSet @ElementsIntoSet

```
@Module
class ServiceModule{
    @Provides @IntoSet
    fun provideOneString() = ”ONE"


    @Provides @ElementsIntoSet
    fun provideSomeStrings() = hashSetOf<String>("TWO", "THREE")
}
```

@Inject lateinit var allStrings: Set<String>

=> allStrings contains {"ONE", "TWO", "THREE"}

@IntoMap:

- Define the key (for simple classes, @StringKey @ClassKey exist already):

```
@MustBeDocumented
@Target(AnnotationTarget.FUNCTION) @Retention(AnnotationRetention.RUNTIME)
@MapKey
internal annotation class ViewModelKey(val value: KClass<out Number>)
```

- Puts DecimalNumber::class as a key and a String as a value into a Map

```
@Provides @IntoMap
@ViewModelKey(DecimalNumber::class)
fun bindDecimalNumber() = “DecimalNumber”
```

- Inject will generate a Map {DecimalNumber::Class-> ”DecimalNumber”)

Exercise 2

Branch: chapter_03_dagger_exercise2

Look for «TODO:chapter_03_dagger_exercise2»

Dagger in Android

How to inject Activities, Fragments etc in the Dagger graph

AndroidInjectionModule

- Part of dagger-android framework
- Contains bindings to ensures dagger-android
- Should be installed in the component of the Application class:

```
@Singleton
@Component(modules = arrayOf(
        AndroidInjectionModule::class,
        AppModule::class))
interface AppComponent{
```

AndroidInjector<T>

- An interface that allows to inject android core types (Activity, Fragment) into the dagger graph
- AndroidInjector is extended by android-specific subcomponents
- Contains an AndroidInjector.Factory interface
- Contains an AndroidInjector.Builder interface which can be implemented by subcomponents Builder
- Usage: create a subcomponent

```
@Subcomponent(modules =...)
interface YourActivitySubcomponent : AndroidInjector<BaseActivtiy> {
    @Subcomponent.Builder
    abstract class Builder : AndroidInjector.Builder<BaseActivtiy>()
}
```

DispatchingAndroidInjector<T>

- Performs member-injection on instances of core Android types
- Internally it uses a Map that binds a concrete AndroidInjector.Builder to an AndroidInjector.Factory<T>

```
class RedditApp : Application(), HasActivityInjector {
    @Inject
    lateinit var activityInjector: DispatchingAndroidInjector<Activity>

    override fun activityInjector() = activityInjector

    override fun onCreate() {
        super.onCreate()
        AppInjector.init(this)
    }
}
```

Don’t forget!

Add the RedditApp which extends Application() to the Manifiest.xml !!!

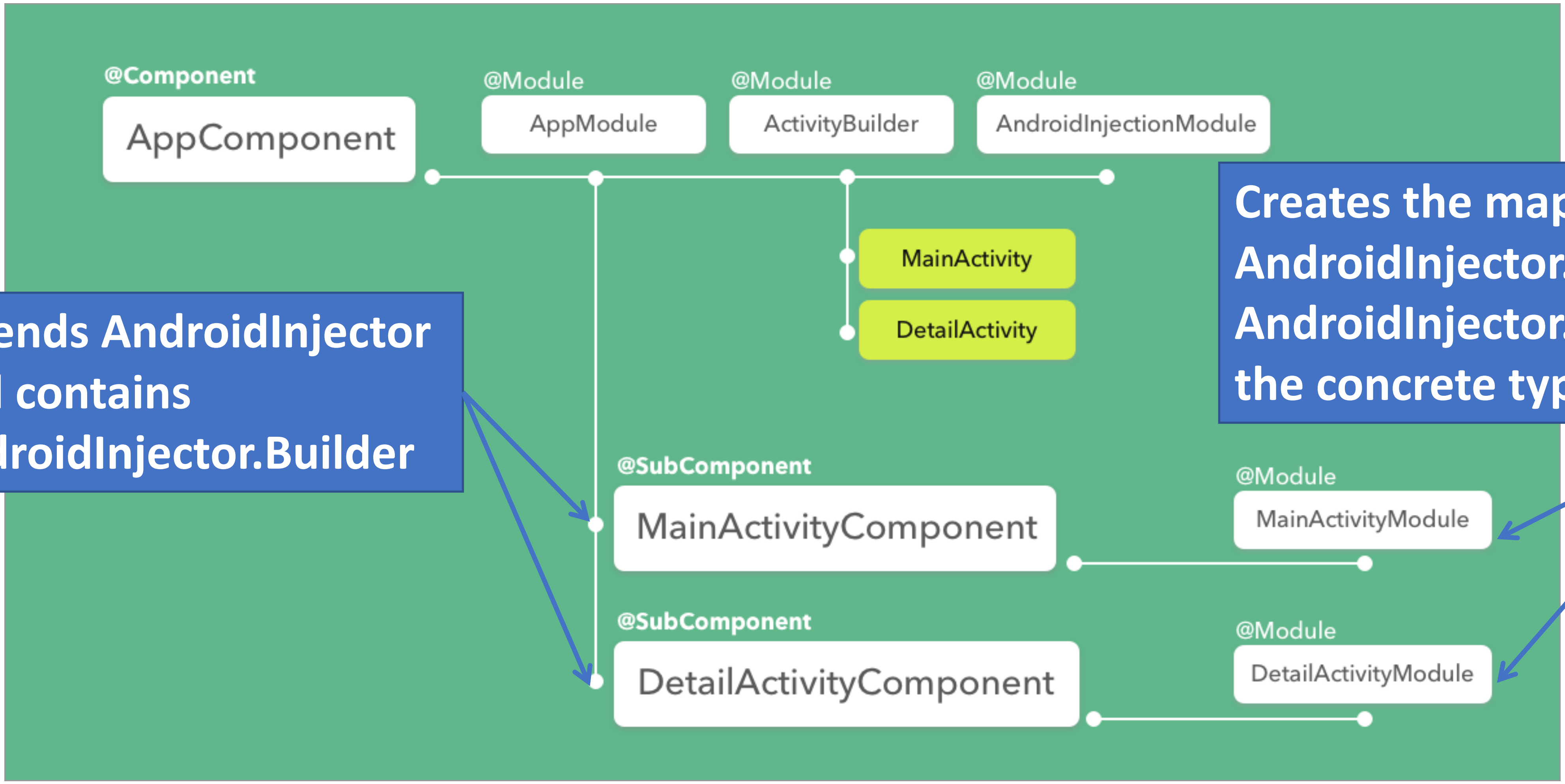
```
<application
    android:name="ch.zuehlke.reddit.RedditApp”
    …
```

We need to create the ActivityModule

- We need to create the map the DispatchingAndroidInjector is using to inject the core Android types: (similar to exercise 2)

```
@Module(subcomponents = arrayOf(YourActivitySubcomponent::class))
internal abstract class YourActivityModule {
    @Binds
    @IntoMap
    @ActivityKey(BaseActivtiy::class)
    internal abstract fun bindYourActivityInjectorFactory(builder:
YourActivitySubcomponent.Builder): AndroidInjector.Factory<out Activity>
}
```

Overview – what was created by us



Extends AndroidInjector and contains AndroidInjector.Builder

Creates the map to bind AndroidInjector.Builder to AndroidInjector.Factory of the concrete types

Boilerplate code

- For each Activity we need to create an ActivitySubcomponent and an ActivityModule
- Repetitive task!
- This was only an example for Activities, for Fragments we have a similar overhead of code

Solution -> @ContributesAndroidInjection

@ContributesAndroidInjector

- Generates for as the repetitive code: f.ex: ActivitySubcomponent and ActivityModule
- It is used in the Module

```
@ContributesAndroidInjector(modules = arrayOf(FragmentBuilderModule::class))
@ActivityScope
abstract fun contributeLoginActivity() : LoginActivity
```

Scope can be added here

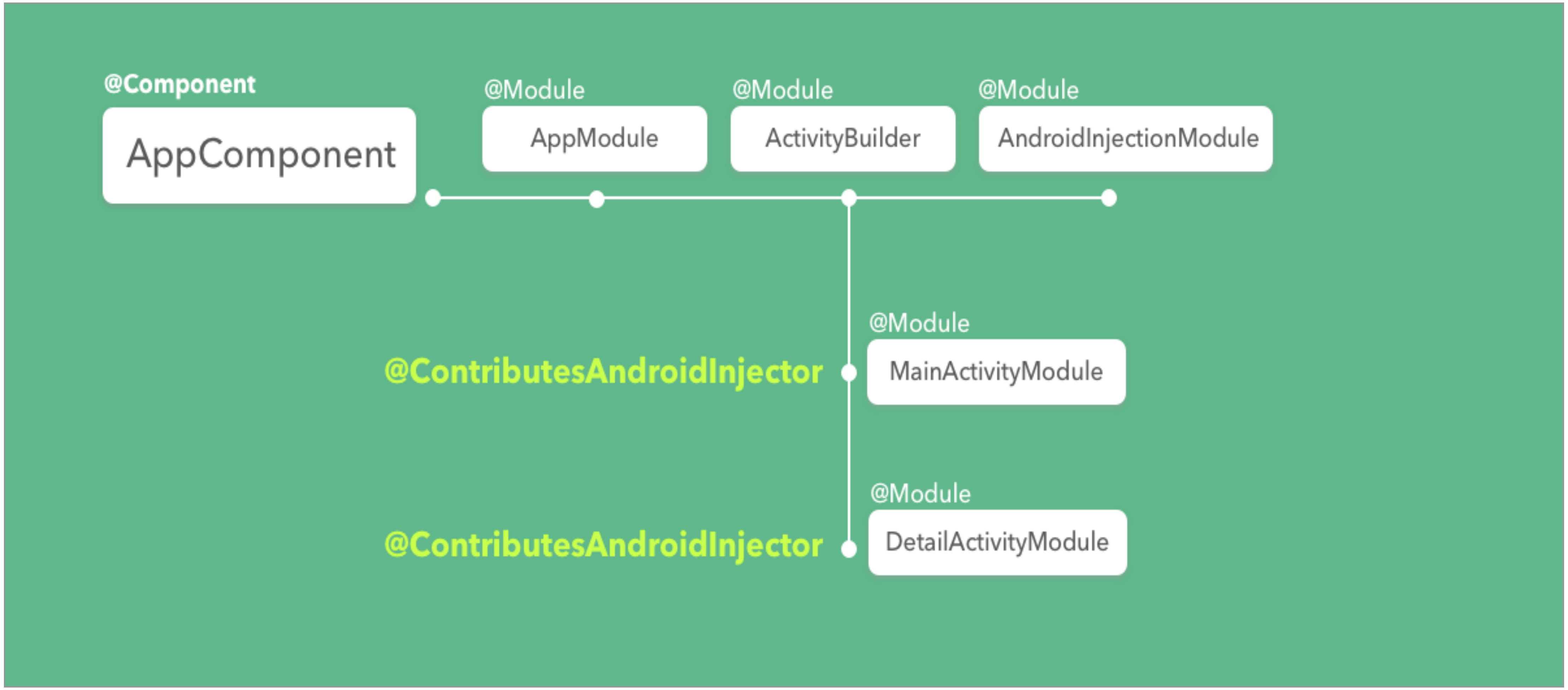
@ContributesAndroidInjector replaces

```
@Module(subcomponents = arrayOf(YourActivitySubcomponent::class))
internal abstract class YourActivityModule {
    @Binds
    @IntoMap
    @ActivityKey(BaseActivtiy::class)
    internal abstract fun bindYourActivityInjectorFactory(builder: 
YourActivitySubcomponent.Builder): AndroidInjector.Factory<out Activity>
}
```

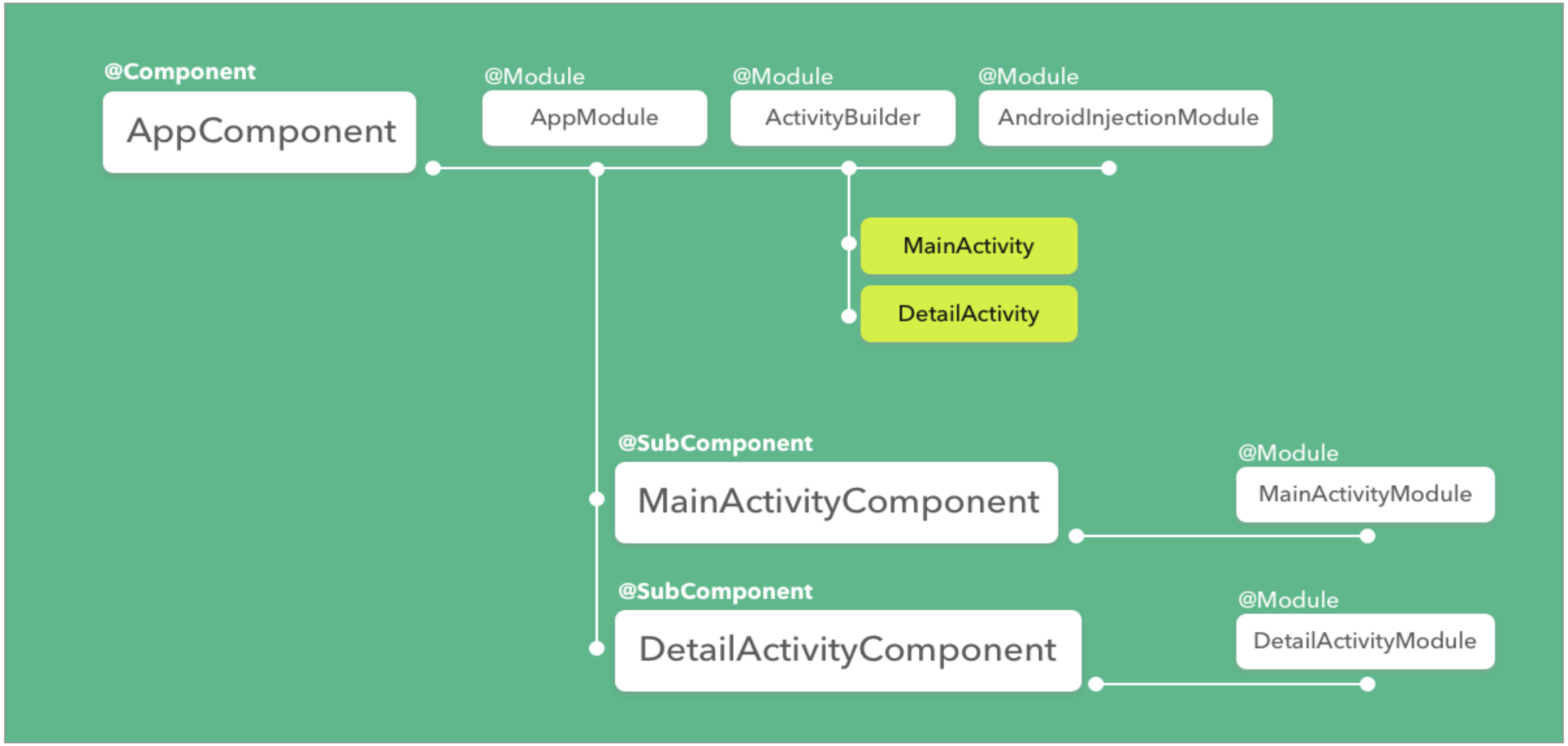
And

```
@Subcomponent(modules =...)
interface YourActivitySubcomponent : AndroidInjector<BaseActivtiy> {
    @Subcomponent.Builder
    abstract class Builder : AndroidInjector.Builder<BaseActivtiy>()
}
```

New Overview



Without ContributesAndroidInjector



Dagger in Android- Setup

1. Make sure to set the name of the application in the Manifest
2. Instantiate all the necessary dependencies in the Module-class with @Provides / @Bind
3. Create an interface «AppComponent» annotated with @Component and add the Module-classes in the annotation
4. Add the AndroidInjectionModule to the Component
5. Create a Component.Builder which allows to bind the Application using @BindsInstance
6. In the @Component class add one function to inject the Application
7. Let the Application implement HasActivityInjector
8. If the Activity has Fragments implement HasSupportFragmentInjector

Alternative DI frameworks for Android

- Kodein – Runtime Injection
- Koin – noch im alpha status

Links und Quellen

- https://android.jlelse.eu/dagger-2-part-i-basic-principles-graph-dependencies-scopes-3dfd032ccd82
- https://proandroiddev.com/dagger-2-component-relationships-custom-scopes-8d7e05e70a37
- https://proandroiddev.com/dagger-2-annotations-binds-contributesandroidinjector-a09e6a57758f
- https://medium.com/@iammert/new-android-injector-with-dagger-2-part-1-8baa60152abe