Dependency injection made easy with Dagger2

Alessandro Candolini

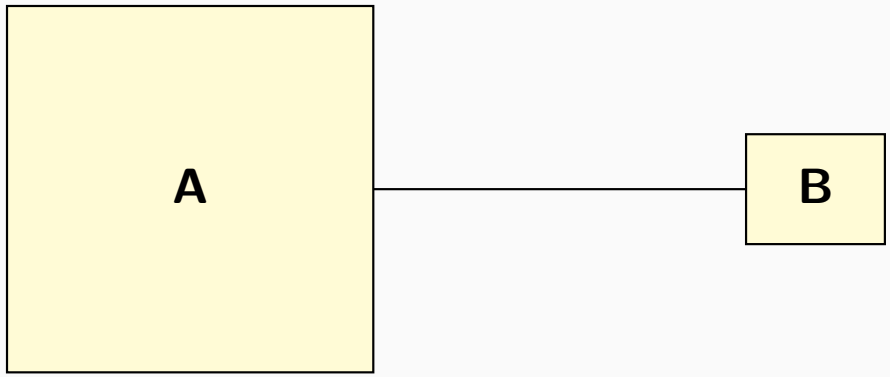
January 25, 2018

Agenda

1. Dependency injection principles
2. Dagger2
3. Dagger2 Android
4. Alternative patterns

Dependency injection principles

What is a dependency?



```
/** Class A (the client) */
class A {
    // ....
    fun doSomething() {
        b.log("text")
    }
}

/** Class B (dependency/service) */
class B {
    fun log(text : String) {
    }
}
```

```
// Option 1 - static methods

class A {
    fun doSomething() {
        B.log("text") // <- static method
    }
}

class B {
    companion object {
        fun log(text: String) {
        }
    }
}
```

Examples:

- Helper classes
- Utils classes
- Manager classes, etc. . .

Drawbacks:

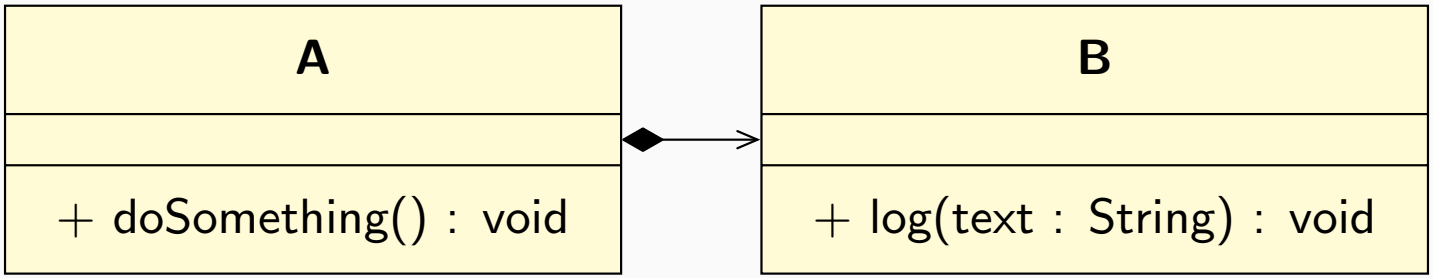
- A not testable in isolation (integration test of A & B)
- A *strongly coupled* to B (hardcoded dependency, no way to override/replace it)
- Lack of encapsulation (backdoor)
- *Hidden* dependency

More Examples:

- `Application.getStaticContext()`
- in order to move one class to a different module, you have to move “hundreds“ of classes. . .

```
// Option 2 - singletons

class A {
    fun doSomething() {
        B.log("text") // <- singleton
    }
}

object B {
    fun log(text: String) {
    }
}
```

```
// Option 3 - composition

class A {
    private val b : B = B() // <-- instantiate

    fun doSomething() {
        b.log("text")
    }
}

class B {
    fun log(text: String) {
    }
}
```



The *life* of the child is completely controlled by the parent.

Example:

- Custom views or adapters instantiating objects
- `Date date = new Date()\`
-

Drawbacks:

- A is in charge of instantiating B (additional responsibility)
- A can't be tested in isolation (integratin test of A and B together)
- A is strongly coupled to B (can't replace B rom outside and/or in testing)

```
// Externalise the dependency

class A(private val b : B) {
    fun doSomething() {
        b.log("text")
    }
}

class B {
    fun log(text: String) {
    }
}
//
val b : B = B()
val a : A = A(b) // <-- plug b
```

We can do even better. . .

```
class A(private val b : B) {
    fun doSomething() {
        b.log("text")
    }
}

interface B {
    fun log(text: String);
}

class AmazingB : B {
    override fun log(text : String) {
    }
}
//
val b : B = AmazingB()
val a : A = A(b)
```

Now we have:

- Full decoupling
- A loosely coupled to B (A knows anything about B but the contract)
- Inversion of control[1]: it's no longer responsibility of A to get its own dependencies

[1] Not yet actually... We miss an ingredient

There is a problem:

```
class C {
    fun qhdouwh() {
        val b = AmazingB() // <-- not good
        val a = A(b) // <-- not good
        a.doSOmething()
    }
}
```

We want (recursively)

```
class C(private a : A) { // A being now an interface
    fun qhdouwh() {
        a.doSOmething()
    }
}
```

Antipattern: we want...

```
class MainActivity : AppCompatActivity() {

    lateinit var presenter : Presenter

    override fun onCreate(bundle: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        presenter.doSomething()
    }
}
```

...instead we get

```
class MainActivity : AppCompatActivity() {

    lateinit var presenter : Presenter

    override fun onCreate(bundle: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val okHttp : OkHttp = /*...*/
        val gson : Gson = GsonBuilder = /*...*/
        val retrofit : Retrofit = /*...*/
        val repository : Repository = /*...*/
        val usecase : UseCase = /*...*/
        val presenter : Presenter = /*...*/
        presenter.doSomething()

    }
}
```

Question

If

- A is not in charge of getting B
- C should not be in charge of instantiating A and B

who is in charge?

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

It directly contrasts with the service locator pattern, which allows clients to know about the system they use to find dependencies.

Source: `https://en.wikipedia.org/wiki/Dependency_injection`

Ingredients:[2]

modules : it containes recipies (methods) to instantiate the dependencies

injector/component : wiring and feed the target with the dependencies it needs

- Often, the injector is also responsible of instantiating the client itself. Sometimes this is not possible (e. g., Android)
- Modules are pluggable in the injector

[2]Preparing the ground for dagger terminology, but here we are not using dagger yet

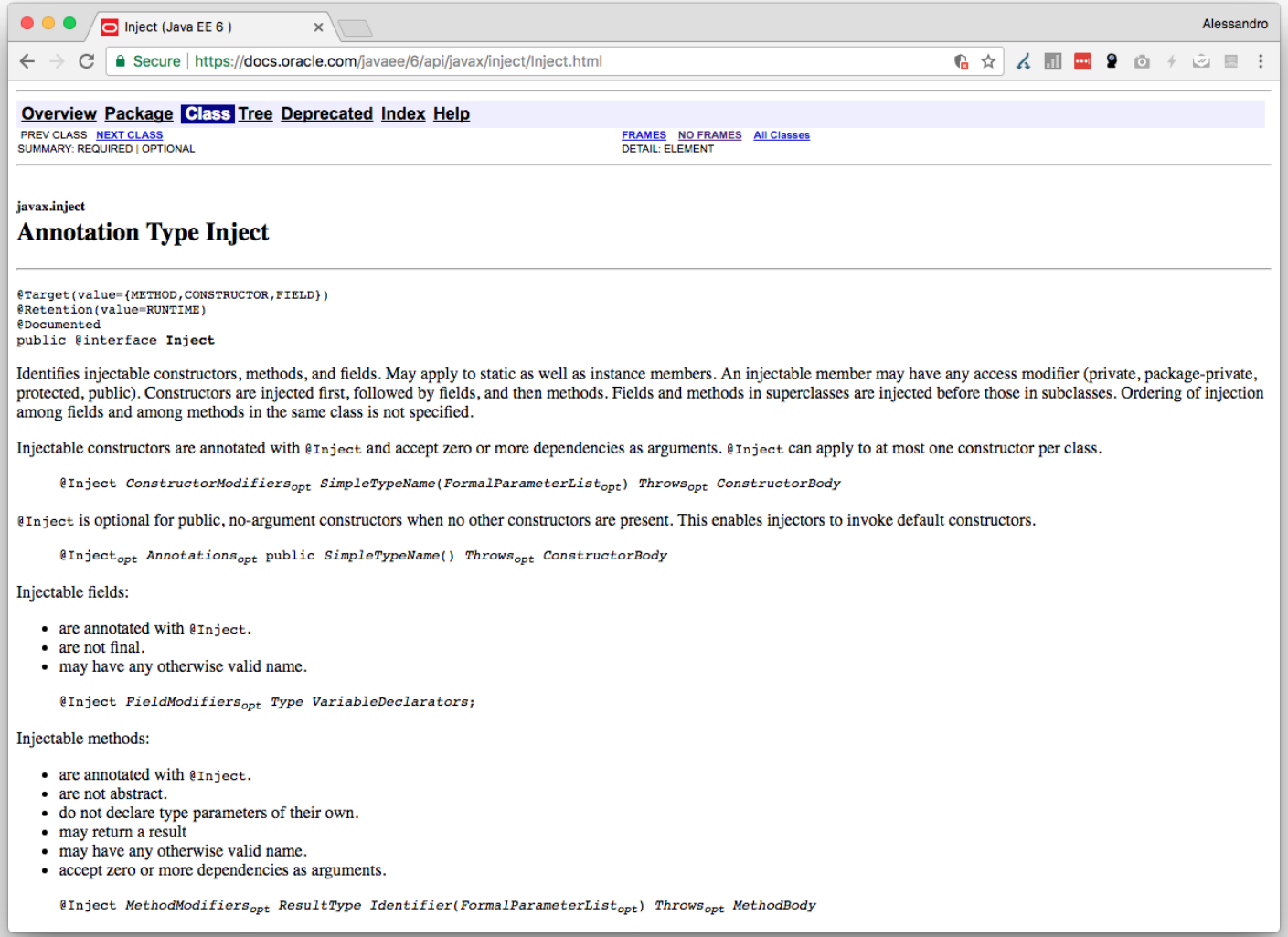
Injector

- ensures inversion of control (i. e., *duality*):
  - You reverse the control of the object dependencies from the object to the one who is calling the object
- restores single responsibility principle

```
// injector  (wiring things up)
interface Component {
    fun inject(activity : MainActivity)
}

// provides the dependencies
class Module {
    fun providePresenter(): Presenter {

    }
}
```

```
class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var presenter : Presenter

    override fun onCreate(bundle: Bundle?) {
        val injector = InjectorImplementation();
        injector.inject(this);
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        presenter.doSomething()
    }
}
```

Activity accepts the dependencies from an injector. It is no longer responsible for creating the dependecies needed, or to delegate instantiation to another object



https://docs.oracle.com/javaee/6/api/javax/inject/Inject.html

```
// provides the dependencies
class Module {
    fun providePresenter(): Presenter {
        val okHttp : OkHttp = ...
        val gson : Gson = GsonBuilder = ...
        val retrofit : Retrofit = ...
        val repository : Repository = /*...*/
        val usecase : UseCase = /*...*/
        return Presenter(usecase)
    }
}
```

We can be more granular:

```
class Module {
  fun providesOkHttp() : OkHttp = /*..*/
  fun providesGson() : Gson = /*..*/
  fun providesRetrofit(gson:Gson, okHttp : OkHttp)
       : Retrofit = /*..*/
  fun providesRepository(retrofit : Retrofit)
       : Repository = /*..*/
  fun providesUseCase(repository:Repository)
       : UseCase = /*..*/

  fun providePresenter(useCase: UseCase)
       : Presenter = PresenterImpl(usecase)
}
```

- modules provide a flexible, pluggable, *declarative* definition on how to instantiate dependencies in the modules
- we have just to implement the interface for the injector that builds the necessary dependencies and plug them into the target (wiring)

The end. . . (really?)

inconveniences

- Boilerplate in implementing the injector
- Combinatorial explosion when chaining mutual dependencies

Dependency injection frameworks help to *automatic organise* the graph of dependencies.

Few famous ones for Java:

- Pivotal's Spring Core Container (autowiring)
- Google's Guava
- Square's Dagger1
- Google's Dagger2

Two strategies (common in java):

- Reflection (runtime)
- Annotation processing (compile time)

Pros/cons. . .

Dagger2

Dagger is

- fully static
- compile-time
- annotation processing based
- dependency injection framework
- for Java/Android
- that organises dependencies into a *directed acyclic graph* (DAGger)

Dagger2 Android

Alternative patterns

- Cake pattern
- Reader monad
- Implicits (Scala)

Questions?