Reflection And Dependency Injection.

Pieter van den Hombergh
Thijs Dorssers
Richard van den Ham

Fontys Hogeschool voor Techniek en Logistiek

May 17, 2018

Reflection

Reflection in Java

Dependency injection

Roll your own

What is reflection

In computer science, reflection is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime.[1]

Is this special?

In the old days, for a program to be able to understand itself was not a big deal.

- Programs were written in assembly language, for one machine.
- Generating code is simple. The instructions are just bytes (or words), so generating bytes and therefor instructions is simple there too.
- Higher level languages lost this capability:
  - Typically do not know what the underlying machine is, hence no knowledge of the instruction format, memory layout etc.
- Some languages have a reflection API. With it you can call find e.g. fields and methods and invoke methods *without knowing them beforehand*.
- Reflection might (and sometimes does) introduce the potential of security loopholes, the same as assembler language did and does.

Pros and cons

- Reflection enables a program to inspect its own classes, but also other classes.
- Java has a security model, including a security manager, but it is not turned on automatically for all programs.
- In that case a program can circumvent the visibility attributes of fields and methods, accessing and even modifying fields that you expect to be `private` and or `final`.
- By enabling the security facility, you can prevent this kind of mischief.
- The reflection API got several updates over its lifetime.
- The update in Java 7 caused many security related problems and was a major cause for the delay of Java 8.

Start at the class

The `Class<T>` class provides the entry point for a first glance at how a class is defined.

- All object know their class, simply call `Class<?> ref.getClass()`.
- This `class` object can be interrogated.
  - `Constructor<?>[] getConstructors()`
  - `Field[] getDeclaredFields()`
  - `Method[] getDeclaredMethods()`
  - `Annotation[] getAnnotations()`
  - `int getModifiers()` (Note: visibilty etc is just a bit, packed in a int.)
- As you can see, the return types are all (arrays of) instances of some type. These types
  - can (recursively) be inspected in their turn on modifiers and annotations.

Reflection API

In Java 8 the reflection API (`java.lang.reflect`) consists of

- 14 interfaces
- 10 classes (excluding java.lang.Class)
- 4 exceptions.

DEMO

Form of Inversion Of Control

- With dependency injection you postpone the binding to an implementation to the moment of late linking, which is what Java does.
- E.g. only at run time will the implementer of various methods defined in the interfaces in `java.sql` and `javax.sql` be resolved.
- This means that your application is not bound to a specific implementation, but that that decision can be postponed to run time configuration.
- It reduces coupling.

By the way:

- Dependency injection is NOT dependent on the use of annotations. It is only that using annotations is quite popular.
- It DOES rely on polymorphism, because what is required is always declared as abstract static type (such as an interface) and of course the instance 'injected' is a concrete implementation of that abstract type.

Coupling is a problem to be avoided

Not just in testing.
Steps in avoiding dependencies:

- Program to interfaces, not to implementations.
- Avoid `new` in code, as this would still bind to an implementation.
- Receive the dependencies by *inserting* (setter/constructor) or **injection**
- This implies that not the class itself but the context (world, it's creator) is responsible of the creation and insertion.
- That also helps tests, because testing is a *different* Context.

The technology is often called **Context and dependency injection**. The service of injecting is provided by a "container". Think of it as the world the classes live in. The aquarium metaphor also works.
It is available in JEE (glassfish) since JEE6 (Glassfish 3).

Note that you can define your own annotations

Look at the annotations tutorial.

`@interface` is just another type with some interface like rules (like having no bodies).

Sounds difficult

Not really. You still ~~cook with water~~ program in java.
But more modern is to use annotations.
These are also provided in Glassfish[1]

```
class SimpleClass {

    @Inject
    private Oracle oracle;

    public void doSomeThing(){
    System.out.println(oracle.giveMeWisdom());
    }
}
```

[1] and any other JEE6 container

Nice, nice

But I want to write a test here, now who does the injecting in this case?

Who does the injecting?

Answer You, who else is there?

Answer Write your own injector.

Fine Now I have yet another problem  .

But it's simple See next slide

Injector helper

Just one method that can inject **@EJB**, **@Inject** and **@Resource**

```
public static void injectField( Object target,
        String fieldName,
        Object dep ) throws Exception {
    String cn = target.getClass().getCanonicalName();
    Field field = target.getClass().getDeclaredField( fieldName );
    Annotation ejb = field.getAnnotation( EJB.class );
    Annotation inject = field.getAnnotation( Inject.class );
    Annotation resource = field.getAnnotation( Resource.class );
    boolean annotationFound = ( null != ejb || null != inject
            || null != resource );
    assertTrue(
            "Missing annotation, none of "
            + "(EJB,Inject,Resource) found on  "
            + cn + "." + fieldName, annotationFound );
    field.setAccessible( true );
    field.set( target, dep );
}
```

The **@Resource** is available in the standard JDK/JRE. But there you will have to find an "container". From the `assertTrue` you may infer that we used this in test classes.