Chapter B

■ ■ ■

Errata

Hi guys, I am really thankful to all of you that have bought the book and have contributed to finding mistakes in it and in the code. Because of you, I have taken another careful look at the book and decided this errata was needed. A special thank you to Koray Tugay (`http://www.tugay.biz/`), as this errata was started because of his observations.

B.1 Book corrections

Chapter 5: Introducing Spring AOP

In page 216 there is a paragraph that can be considered incorrect.
Original:The only restriction, in Spring AOP at least, is that you can't advise final classes, because they cannot be overridden and therefore cannot be proxied.

Correct: The incorrect word there is **overridden**. And should be replaced with **extended**. Also, the context is incomplete. When a class does not implement an interface, the proxy is created by extending the class. Therefore, **a final class that does not implement an interface** cannot be proxied.

In page 248, section **Convenience Advisor Implementations** the sample code was wrongfully copied from the previous section. The correct section of code is:

```
package com.apress.prospring5.ch5;
...
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;

public class NamePointcutUsingAdvisor {
   public static void main(String... args) {
      GrammyGuitarist johnMayer = new GrammyGuitarist();

      NameMatchMethodPointcutAdvisor advisor =
          new NameMatchMethodPointcutAdvisor(new SimpleAdvice());
      advisor.setMappedNames("sing");
      advisor.setMappedNames("rest");

      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(johnMayer);
      pf.addAdvisor(advisor);

      GrammyGuitarist proxy = (GrammyGuitarist) pf.getProxy();
      proxy.sing();
      proxy.sing(new Guitar());
      proxy.rest();
      proxy.talk();
   }
}
```

In page 217, section **Creating Advice in Spring** it is said that Spring supports six flavors of advice. Some people say its only four. The reason for that is that most developers exclude `IntroductionInterceptor` and tend to wrap all `After` advice into a family. If we want to keep things simple, we can focus only on method advice and then we could reduce them to three: before, after and around advice. Which is also correct. it depends on what you are interested in.

In this book, those six types of advice are considered important. In Figure B.1 you can see the relationships between these types of objects. All interfaces extend `Advice`. Except for `IntroductionInterceptor`, all are

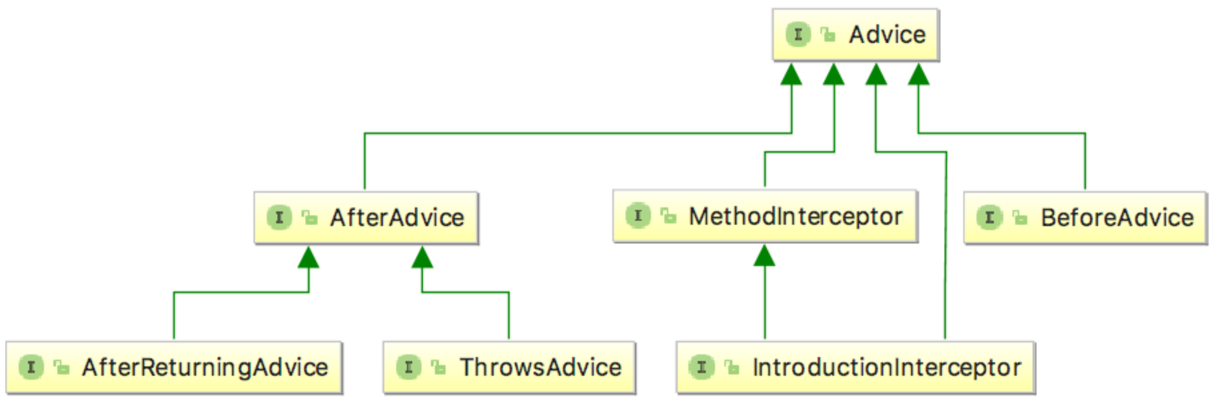

Figure B.1: Partial Spring advice hierarchy

method specific advice. The `IntroductionInterceptor` interface extends not only the `MethodInterceptor(` use for around advice), but also extends `DynamicIntroductionAdvice` that is a special type of advice, that allows additional interfaces that are not known in advance to be implemented by an advice, making this a type specific advice.

We could mention here also `ConstructorInterceptor`, but as constructors are a special type of methods, covering it did not seem necessary.

In the table 5-1, the row corresponding to the `After(finally)` advice contains an incorrect explanation.
Original:

Advice Name	**Interface**	**Description**
`After(finally)`	`org.springframework.aop .AfterAdvice`	After-returning advice is executed only when the advised method completes normally. However, the after (finally) advice will be executed no matter the result of the advised method. The advice is executed even when the advised method fails and an exception is thrown.

Table B.1: Advice Types in Spring (1)

The corrected version is depicted in the table snipped below.
In page 271, section **Configuring AOP Declaratively** the three options for using declarative configuration of Spring AOP are listed. It is mentioned that for `@AspectJ`-style annotations you need to include some AspectJ libraries in the classpath. The same applies for the second option in the list: using the Spring `aop namespace`,

Advice Name	Interface	Description
`After(finally)`	`org.springframework.aop .AfterAdvice`	An after-returning advice is executed only when the advised method completes normally. However, the after (finally) advice will be executed no matter the result of the advised method. The advice is executed even when the advised method fails and an exception is thrown. This interface type is only a marker interface for both after advice types supported by Spring. The After(finally) advice is not supported by Spring natively, the implementation has to be provided by an external library like AspectJ.

Table B.2: Advice Types in Spring (1)

because the XML configuration is the precursor of the annotation style configuration. And some type of advice, like the after (finally) advice is needed to be used, an implementation needs to be provided via an external dependency such as AspectJ, as it is not supported natively by Spring.

Chapter 16: Web Applications

Because the book was written when Spring 5 was still under construction, Spring Boot 2 as well, when libraries upgrades happen code samples might stop working as intended. When it comes to applications secured with Spring Security, Spring Security 5.0.0.RC1 came with fixes for 150+ issues, and quite a few of them were related to password security. And thus `PasswordEncoder` implementations are now required. When configuring in memory authentication, passwords were until this version stored in the compiled code in their original form, no encoding whatsoever. This behaviour can still be kept, especially for educational applications that focus on other details. Below you can see two samples of code on how to do this:

```
 @Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
   auth
      .inMemoryAuthentication()
      .withUser("user").password("{noop}user").roles("USER");
}

// or
import org.springframework.security.crypto.password.NoOpPasswordEncoder;
...
 @Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
  auth
      .inMemoryAuthentication()
      .passwordEncoder(NoOpPasswordEncoder.getInstance())
      .withUser("user").password("user").roles("USER");
 }
```

But who knows, maybe you will need to build an application that makes use of an in memory authentication style (usually used only for test environments and educational applications) and stores the passwords in the code. Thus, you will need for your passwords to be encrypted, so they won't be visible in the decompiled jar. Spring Security supports quite a few password encoder implementation out of the box, and you can even implement your own. In the following example, a simple Spring implementation `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder` was used.

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
  PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
  auth
     .inMemoryAuthentication()
     .passwordEncoder(passwordEncoder)
     .withUser("user").password(passwordEncoder.encode("user")).roles("USER");
    }
```