Introducing Spring AOP

Besides Dependency Injection (DI), another core feature that the Spring Framework brings to the developer community is Aspect-Oriented Programming (AOP). Although it used to be difficult to learn, understand, and implement, thanks to Spring’s intensive use of AOP within the framework and a simplified AOP programming model that Spring provides, AOP has become a technique that developers use on day-to-day development, especially when developing Spring-based applications. AOP is often referred to as a tool for implementing crosscutting concerns. When you “cut through” the unfamiliar terminology, you use AOP for modularizing individual pieces of logic, known as concerns, and you apply these concerns to many parts of an application. Logging and security are typical examples of crosscutting concerns that are present in many applications. Consider an application that logs the start and end of every method for debugging purposes. You will probably refactor the logging code into a special class, but you still have to call methods on that class twice per method in your application in order to perform the logging. Using AOP, you can simply specify that you want the methods on your logging class to be invoked before and after each method call in your application.

It is important that you understand that AOP complements OOP, rather than competes with it. OOP is very good at solving a wide variety of problems that we, as programmers, encounter. However, if you look at the logging example again, it is quite plain to see that OOP is lacking when it comes to implementing crosscutting logic on a large scale. Using AOP on its own to develop an entire application is practically impossible, given that AOP functions on top of OOP. Likewise, although it is certainly possible to develop entire applications using OOP, you can work smarter by employing AOP to solve certain problems that involve crosscutting logic.

We are going to cover AOP in this chapter and the next. In particular, this chapter covers the following topics:

- **AOP basics**: Before we begin discussing Spring’s AOP implementation, we cover the basics of AOP as a technology. Most of the concepts covered in this section are not specific to Spring and can be found in any AOP implementation. If you are already familiar with another AOP implementation, then feel free to skip this section.
- **Types of AOP**: There are two distinct types of AOP: static and dynamic. In static AOP, like that provided by AspectJ’s (http://eclipse.org/aspectj/) compile-time weaving mechanisms, the crosscutting logic is applied to your code at compile time, and you cannot change it without modifying the code and recompiling. With dynamic AOP, like Spring AOP, crosscutting logic is applied dynamically, at runtime. This allows you to make changes in the distribution of crosscutting without recompiling the application. These types of AOP are complementary, and, when used together, they form a powerful combination that you can use in your applications.
Spring AOP architecture: In this section, we get down to the nitty-gritty of Spring’s AOP implementation. Spring AOP is only a subset of the full AOP feature set found in other implementations like AspectJ. In this section, we take a high-level look at which features are present in Spring, how they are implemented, and why some features are excluded from the Spring implementation.

Proxies in Spring AOP: Proxies are a huge part of how Spring AOP works, and you must understand them to get the most out of Spring AOP. In this section, we look at the two kinds of proxy: the JDK dynamic proxy and the CGLIB proxy. In particular, we look at the different scenarios in which Spring uses each proxy, the performance of the two proxy types, and some simple guidelines to follow in your application to get the most from Spring AOP.

Using Spring AOP: In this section, we present some practical examples of AOP usage. We start off with a simple “Hello World” example to ease you into Spring’s AOP code, and we continue with a detailed description of the different AOP features that are available in Spring, complete with examples.

In this chapter, we cover Spring AOP in isolation from much of the rest of the framework. In Chapter 7, we take a much more framework-oriented view of Spring AOP, including how to configure AOP using ApplicationContext.

AOP Concepts

As with most technologies, AOP comes with its own specific set of concepts and terms. It is important that you understand what these terms mean before we explain how to use AOP in an application. The following list explains the core concepts of AOP:

- **Joinpoints**: A joinpoint is a well-defined point during the execution of your application. Typical examples of joinpoints include a call to a method, the Method Invocation itself, class initialization, and object instantiation. Joinpoints are a core concept of AOP and define the points in your application at which you can insert additional logic using AOP.

- **Advice**: The code that is executed at a particular joinpoint is the advice. There are many different types of advice, such as before, which executes before the joinpoint, and after, which executes after it. In OOP, an advice comes in the form of a method within a class.

- **Pointcuts**: A pointcut is a collection of joinpoints that you use to define when advice should be executed. By creating pointcuts, you gain fine-grained control over how you apply advice to the components in your application. As mentioned previously, a typical joinpoint is a Method Invocation. A typical pointcut is the collection of all Method Invocations in a particular class. Often you can compose