2 Introducing AspectJ

This chapter covers

- Writing an AspectJ "Hello, world!" application
- Becoming familiar with the AspectJ language
- Weaving mechanisms
- Integrating with Spring

In chapter 1, we focused on general concepts in AOP. With those behind us, we can now look at one specific AOP implementation: AspectJ. AspectJ is an aspect-oriented extension to the Java programming language. Like any AOP implementation, AspectJ consists of two parts: the language specification, which defines the grammar and semantics of the language; and the language implementation, which includes weavers that take various forms such as a compiler and a linker. A weaver produces byte code that conforms to the Java byte-code specification, allowing any compliant Java virtual machine (VM) to execute those class files. The language implementation also offers support for integrated development environments (IDEs), to simplify building and debugging applications.

AspectJ started and initially grew as a special language that extends the Java language with new keywords. It also provided a special compiler that could understand those extensions. But recently, a lot has changed in its form as a language, as well as

in the weaver. First, AspectJ offers an alternative syntax based on the Java annotation facility to express crosscutting constructs. This lets you use a plain Java compiler instead of the special compiler. Second, AspectJ offers new options for weaving classes with aspects. Finally, it has gained a strong foothold in the Spring Framework with several integration options. All these changes have made adoption of AspectJ easier than ever before.

In this chapter, we'll examine important facets of AspectJ—starting with language constructs, passing through syntax and weaving choices, peeking into the Spring integration, and ending with tools support—from a high-level perspective. In the chapters that follow, we'll delve deeper into each of these facets.

2.1 Writing your first AspectJ program

You'll begin your journey by writing a simple application. This code introduces a few AspectJ concepts and gives you a feel for the language.

Installing AspectJ

You'll need to install AspectJ to work with the code in this book. Doing so is easy:

1. Download the AspectJ distribution from http://www.eclipse.org/aspectj/downloads.php.
2. Run `java -jar <path-to-downloaded-file>`. Doing so opens a wizard that will guide you through the installation process. When the wizard finishes, you'll have the AspectJ compiler and other tools, various AspectJ libraries (such as aspectjrt.jar and aspectjweaver.jar), and documentation.

Alternatively, you can use Eclipse along with the AspectJ Development Tools (AJDT) to run these examples in the IDE. See section 2.7.1 for details.

2.1.1 Setting up the example

Let's create a regular Java class, as shown in listing 2.1, which contains two methods that will print messages. Later in this section, you'll create a few aspects to introduce additional behavior without modifying the class.

Listing 2.1 Class encapsulating the message-delivery functionality

```
package ajia.messaging;

public class MessageCommunicator {
    public void deliver(String message) {
        System.out.println(message);
    }

    public void deliver(String person, String message) {
        System.out.println(person + ", " + message);
    }
}
```

The `MessageCommunicator` class has two methods: one to deliver a general message and the other to deliver a message to a specific person. Next, let's write a simple class to exercise the functionality of the `MessageCommunicator` class, as shown in listing 2.2.

Listing 2.2 Class to exercise the message-delivery functionality

```
package ajia.main;

import ajia.messaging. MessageCommunicator;

public class Main {
    public static void main(String[] args) {
        MessageCommunicator messageCommunicator
            = new MessageCommunicator();
        messageCommunicator.deliver("Wanna learn AspectJ?");
        messageCommunicator.deliver("Harry", "having fun?");
    }
}
```

When you compile the `MessageCommunicator` and the `Main` class together using the AspectJ compiler (ajc) and run the `Main` program, the output is as follows (Commands shown in this chapter are for Windows; please change them appropriately for your platform.):

```
> ajc ajia\messaging\MessageCommunicator.java ajia\main\Main.java
> java ajia.main.Main
Wanna learn AspectJ?
Harry, having fun?
```

Running the examples

The main reason behind showing the command-line operations is to remove some of the mystery about how AspectJ works. The downloadable sources include shell scripts that match the commands and also include the necessary setup. But most developers will prefer to run the code through the Ant and Maven scripts also provided with the downloadable sources. See appendixes B and C for detailed instructions on how to use these tools with AspectJ.

Every valid Java program is a valid AspectJ program. Therefore, you could use the AspectJ compiler (ajc) to compile the classes instead of a Java compiler such as javac.

Now that the basic setup is ready, let's add a few aspects to the system to improve the message-delivery functionality.

2.1.2 Adding an aspect

Consider authentication functionality: before delivering a message, you'd like to check whether the user has been authenticated. Without using AOP, you'd have to write code like the following:

```
public class MessageCommunicator {
    private Authenticator authenticator = new Authenticator();

    public void deliver(String message) {
        authenticator.authenticate();
        System.out.println(message);
    }
```

```
    public void deliver(String person, String message) {
        authenticator.authenticate();
        System.out.println(person + ", " + message);
    }
}
```

You have to add a call to `authenticate()` in each method that needs authentication, which leads to code tangling. Similar code must be present in all classes that require the authentication functionality—leading to code scattering. With AOP, you can do better.

Without changing a single line of code in the `MessageCommunicator` class from listing 2.1, you can enhance its functionality by adding an aspect to the system. Listing 2.3 shows the traditional syntax. Later, you'll see alternative syntax to implement the same functionality.

Listing 2.3 Aspect to secure access

```
package ajia.security;

import ajia.messaging.MessageCommunicator;

public aspect SecurityAspect {
    private Authenticator authenticator = new Authenticator();

    pointcut secureAccess()
        : execution(* MessageCommunicator.deliver(..));

    before() : secureAccess() {
        System.out.println("Checking and authenticating user");
        authenticator.authenticate();
    }
}
```

1 **Aspect declaration**

2 **Pointcut declaration**

3 **Advice**

The `Authenticator` class asks for credentials (username and password) when the `authenticate()` method is called for the first time in a thread. Upon successful authentication, it stores the user in a thread local so it doesn't ask for credentials in the same thread again. Upon failure, it throws a runtime exception. (I don't show the `Authenticator` here, for brevity's sake. If you like, you can download it from the book's web site: http://manning.com/laddad2/.)

Compile the classes along with the aspect. Now, when you run the program, you see the following output (the use of Java generic feature in the `Authenticator` class requires specifying the `-source 5` option to ajc):

```
> ajc  -source 5 ajia\messaging\MessageCommunicator.java
➥   ajia\main\Main.java
➥   ajia\security\SecurityAspect.aj ajia\security\*.java
> java ajia.main.Main
Checking and authenticating user
Username: ajia
Password: ajia
Wanna learn AspectJ?
Checking and authenticating user
Harry, having fun?
```

The SecurityAspect.aj file declares the `SecurityAspect` aspect. Note that you could have declared the aspect in SecurityAspect.java file, because AspectJ accepts both .aj and .java extensions for input source files. Although the file extension doesn't matter to the compiler, aspects typically use the .aj extension, and Java code uses the .java extension. Let's looks at the listing in more detail:

1. An aspect is a unit of modularization in AOP, much like a class is a unit of modularization in OOP. The declaration of an aspect is similar to a class declaration.
2. A pointcut selects interesting points of execution in a system, called *join points*. The aspect defines a pointcut `secureAccess()` that selects execution of all the methods named `deliver()` in the `MessageCommunicator` class. The `*` indicates that the pointcut matches any return type, and the `..` inside parentheses after `deliver` specifies that it matches regardless of the number of arguments or their types. In this example, the pointcut selects execution of both overloaded versions of `deliver()` in the `MessageCommunicator` class. You'll learn about join points and pointcuts in detail in the next chapter.
3. An advice defines the code to execute upon reaching join points selected by the associated pointcut. Here, you define a piece of advice to execute before reaching the join points selected by the `secureAccess()` pointcut. The `before()` part indicates that the advice should run prior to the execution of the advised join point—in this case, prior to executing any `MessageCommunicator.deliver()` method. In the advice, you authenticate the current user. With the aspect now present in the system, each time `MessageCommunicator.deliver()` is executed, the advice code performs the authentication logic before the method.

Now that you have the flavor of the AspectJ language, it's time for an overview of the language and its core building blocks.

2.2 AspectJ crosscutting construct

Recall the generic AOP model discussed in chapter 1. AspectJ is the most complete implementation of that model, supporting all its elements. In this section, we'll examine how AspectJ maps each model element into program constructs. Note that AspectJ offers two syntax choices: traditional and @AspectJ. This section uses the traditional syntax to study these building blocks. We'll examine the @AspectJ syntax in section 2.3.

We can classify the crosscutting constructs in the AOP model as *common* crosscutting constructs (join point, pointcut, and aspect), *dynamic* crosscutting construct (advice), and *static* crosscutting constructs (inter-type declarations and weave-time declarations). These constructs form the building blocks of AspectJ. Let's study the common crosscutting constructs first.

2.2.1 Common crosscutting constructs

AspectJ supports a few common constructs consisting of the join point, the pointcut, and the aspect. You can use these constructs with both dynamic and static crosscutting.

JOIN POINT

In AOP, and therefore in AspectJ, *join points* are the places where the crosscutting actions take place. Listing 2.1 has join points corresponding to the execution of the `deliver()` methods as well as calls to the `println()` method on the `System.out` object. Listing 2.2 has join points corresponding to the creation of `MessageCommunicator` and calls to the `deliver()` methods.

After you identify join points useful for a crosscutting functionality, you need to select them using the pointcut construct.

POINTCUT

A *pointcut* is a program construct that selects join points and collects context at those points. For example, a pointcut can select a join point that is an execution of a method. It can also collect the join-point context, such as the `this` object and the arguments to the method.

The following pointcut selects the execution of any public method in the system:

```
execution(public * *.*(..))
```

The wildcards `*` and `..` indicate that the pointcut selects regardless of the return type, declaring type, method name, and method parameters. Here, the only condition specified is that the access specification for the method must be `public`.

It's a good idea to name a pointcut so that other programming elements can use it (and so that programmers—including yourself—can understand the intention behind the pointcut). For example, you can name the earlier pointcut `publicOperation`, as follows:

```
pointcut publicOperation() : execution(public * *.*(..));
```

ASPECT

The *aspect* is the central unit in AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Additionally, aspects can contain data, methods, and nested class members, just like a normal Java class. Let's define an aspect that performs profiling that you'll update as you learn about more elements:

```
package ajia.profile;

public aspect ProfilingAspect {
}
```

Learning about common crosscutting constructs will pay off when you begin using them with dynamic and static crosscutting constructs. Let's see how, starting with the dynamic crosscutting construct.

2.2.2 Dynamic crosscutting construct: advice

AspectJ's dynamic crosscutting support comes in the form of advice. *Advice* is the code executed at a join point selected by a pointcut. Advice can execute before, after, or around the join point. The body of advice is much like a method body—it encapsulates the logic to be executed upon reaching a join point.

Using the `publicOperation()` pointcut from the previous section, you can advise all public methods of `MessageCommunicator` to profile them. Let's update `ProfilingAspect` (shown in listing 2.4) with profiling advice.

Listing 2.4 Profiling all public methods

```
package ajia.profile;

public aspect ProfilingAspect {
    pointcut publicOperation() : execution(public * *.*(..));

    Object around() : publicOperation() {
        long start = System.nanoTime();
        Object ret = proceed();
        long end = System.nanoTime();
        System.out.println(thisJoinPointStaticPart.getSignature()
                           + " took " + (end-start) + " nanoseconds");
        return ret;
    }
}
```

The advice records the start time, calls `proceed()` to continue executing the advised method, records the end time, and prints the time taken by the method execution. The `thisJoinPointStaticPart` variable is one of the three variables available in each advice that carry information about the currently advised join point, such as the method name, the `this` object, and method arguments.

When you compile this aspect along with the other code and execute it, you get the following output:

```
> ajc -source 5 ajia\messaging\MessageCommunicator.java ajia\main\Main.java
➥   ajia\security\SecurityAspect.aj ajia\security\*.java
➥   ajia\profile\ProfilingAspect.aj
> java ajia.main.Main
Checking and authenticating user
boolean ajia.security.Authenticator.isAuthenticated() took 840051
➥   nanoseconds
Username: ajia
Password: ajia
String[] ajia.security.Authenticator.getUserNamePassword() took 5248473759
➥   nanoseconds
void ajia.security.Authenticator.authenticate() took 5250886077 nanoseconds
Wanna learn AspectJ?
void ajia.messaging.MessageCommunicator.deliver(String) took 5252761734
➥   nanoseconds
Checking and authenticating user
boolean ajia.security.Authenticator.isAuthenticated() took 5028 nanoseconds
void ajia.security.Authenticator.authenticate() took 61740 nanoseconds
Harry, having fun?
void ajia.messaging.MessageCommunicator.deliver(String, String) took 307581
➥   nanoseconds
void ajia.main.Main.main(String[]) took 5253861315 nanoseconds
```

While dynamic crosscutting alters the program behavior, static crosscutting alters the programs structure. Let's see AspectJ's support for it.

Pseudo keywords in AspectJ

The `proceed` and other keywords such as `aspect`, `pointcut`, and `before` are really pseudo keywords that gain special meaning only in the right context. For example, it is perfectly legitimate to use a method named 'proceed' in Java classes. But when `proceed` is used in an around advice, it acquires a special meaning. This use of pseudo keywords enables AspectJ to work with any valid Java program that may already include AspectJ keywords.

2.2.3 Static crosscutting constructs

Static crosscutting comes in the form of inter-type and weave-time declarations.

INTER-TYPE DECLARATION

The *inter-type declaration* (ITD) (also referred to as *introduction*) is a static crosscutting construct that alters the static structure of the classes, interfaces, and aspects in the system. For example, you can add a method or field to a class, or declare a type to implement an interface. In an ITD, one type (an aspect) declares the structure for the other types (classes, interfaces, and even aspects)—hence the name.

The following statement makes the weaver assign the `AccessTracked` interface as the parent of the `MessageCommunicator` class:

```
declare parents: MessageCommunicator implements AccessTracked;
```

When this declaration is woven in, it has the same effect as declaring the `MessageCommunicator` class as follows:

```
public class MessageCommunicator implements AccessTracked {
    ...
}
```

Another form of ITD—*member introduction*—offers a way to add new methods and fields to other types. The following declaration adds the `lastAccessedTime` field and the `updateLastAccessedTime()` and `getLastAccessedTime()` methods to the `AccessTracked` type:

```
private long AccessTracked.lastAccessedTime;

public void AccessTracked.updateLastAccessedTime() {
    lastAccessedTime = System.currentTimeMillis();
}

public long AccessTracked.getLastAccessedTime() {
    return lastAccessedTime;
}
```

You can then advise methods in a type that implements `AccessTracked` (directly or through a `declare parents` statement) to update the last-accessed time, as shown in the following snippet:

```
before(AccessTracked accessTracked)
    : execution(* AccessTracked+.*(..))
```

```
      && !execution(* AccessTracked.*(..))
      && this(accessTracked){
    accessTracked.updateLastAccessedTime();
}
```

This code advises all methods of types that implement the `AccessTracked` interface (the + wildcard denotes subtypes) but not the method in `AccessTracked` (such as the introduced `updateLastAccessedTime()` method). The `this()` pointcut collects the tracked object so you can call the `updateLastAccessedTime()` method on it.

Let's put all these snippets in an aspect, as shown in listing 2.5, to see their effect.

Listing 2.5 Tracking the last-accessed time using an aspect

```
package ajia.track;

import ajia.messaging.MessageCommunicator;

public aspect TrackingAspect {
    declare parents: MessageCommunicator implements AccessTracked;

    private long AccessTracked.lastAccessedTime;

    public void AccessTracked.updateLastAccessedTime() {
        lastAccessedTime = System.currentTimeMillis();
    }

    public long AccessTracked.getLastAccessedTime() {
        return lastAccessedTime;
    }

    before(AccessTracked accessTracked)
        : execution(* AccessTracked+.*(..))
          && !execution(* AccessTracked.*(..))
          && this(accessTracked) {
         accessTracked.updateLastAccessedTime();
    }

    private static interface AccessTracked {
    }
}
```

To see the effect, modify the `Main` class to print the last-accessed time for the `MessageCommunicator` object, as shown in listing 2.6.

Listing 2.6 Modified `Main` class to print the last-accessed time

```
package ajia.main;

...

public class Main {
    public static void main(String[] args) {
        ...

        System.out.println("Last accessed time for messageCommunicator "
                           + messageCommunicator.getLastAccessedTime());
    }
}
```

When you compile and execute this class, the output is as follows:

```
> ajc -source 5 ajia\messaging\MessageCommunicator.java ajia\main\Main.java
➥   ajia\security\SecurityAspect.aj ajia\security\*.java
➥   ajia\profile\ProfilingAspect.aj ajia\track\TrackingAspect.aj
> java ajia.main.Main
...
Last accessed time for messageCommunicator 1250040714984
...
```

ITDs also offer a way to annotate program elements and deal with checked exceptions in a systematic manner, but we'll defer that discussion until chapter 5.

An important form of static crosscutting allows detecting and flagging the presence of join points, matching a pointcut during compilation. Let's see how.

WEAVE-TIME DECLARATION

The *weave-time declaration* is another static crosscutting construct that allows you to add weave-time warnings and errors when detecting certain usage patterns. Often, weaving is performed during compilation; therefore, these warnings and errors are issued when you compile the classes.

Consider `SecurityAspect` from listing 2.3. With this aspect in place, you might want to warn about direct calls to the `Authenticator.authenticate()` method. The following declaration will cause the compiler to issue a warning if any part of the system calls the prohibited method—except, of course, `SecurityAspect`:

```
declare warning
    : call(void Authenticator.authenticate()) && !within(SecurityAspect)
    : "Authentication should be performed only by SecurityAspect";
```

Note the use of the `call()` pointcut to select a method call (as opposed to selecting the method execution, which is always in the `Authenticator` class) and `!within()` to restrict selection of join points to only those occurring outside `SecurityAspect`. The weaver will report warnings when it detects the specified conditions along with other compile-time warnings such as use of a deprecated method.

Let's see this in action by modifying `SecurityAspect` to add this declaration:

```
package ajia.security;

public aspect SecurityAspect {

    ...

    declare warning
        : call(void Authenticator.authenticate())
          && !within(SecurityAspect)
        : "Authentication should be performed only by SecurityAspect";
}
```

You also modify the `Main` class to add `new Authenticator().authenticate()` (and the associated import statements) to test a violation. When you compile the code, the output is as follows:

```
> ajc -source 5 ajia\messaging\MessageCommunicator.java ajia\main\Main.java
➥   ajia\security\SecurityAspect.aj ajia\security\*.java
...\ajia\main\Main.java:13 [warning]
```

```
➥  Authentication should be performed only by SecurityAspect
new Authenticator().authenticate();
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        method-call(void ajia.security.Authenticator.authenticate())
        see also: ...\ajia\security\SecurityAspect.aj:15::0
```

As the output shows, the compiler detects and flags the violation and also points to the aspect with the corresponding `declare warning` statement.

This completes our discussion of various crosscutting constructs in AspectJ. By offering comprehensive support for various elements, AspectJ enables modularization of crosscutting concerns. You'll see examples based on these constructs throughout the book.

Until now, we've restricted ourselves only to the traditional AspectJ syntax and compiler as the weaver. But AspectJ offers an alternative syntax that we'll study next. Later in this chapter, we'll examine alternative weaving models.

2.3 AspectJ alternative syntax

The traditional syntax we've been using so far requires the use of the special ajc compiler early in the development process to compile aspects. This poses a potential barrier to adopting AspectJ. To simplify adoption, starting with AspectJ version 5, alternative syntax and weaving models are available, thanks to the merger of AspectJ with AspectWerkz, another implementation of AOP for Java.

The alternative @AspectJ syntax extends the language using the new annotation facility in Java 5. The main advantage of this syntax style is that you can compile your code using a plain Java compiler (for example, javac). As a result, the code works better with conventional Java IDEs and tools that don't understand the traditional AspectJ syntax. Furthermore, the proxy-based AOP framework in Spring uses this syntax, simplifying adoption of AspectJ if the project is already using Spring. The disadvantage of @AspectJ syntax is its verbosity in expressing the same constructs and its limitations in expressing certain constructs, especially in the static crosscutting category.

Let's create an @AspectJ version of the aspect from listing 2.3, as shown in listing 2.7.

Listing 2.7 Introducing security using the @AspectJ syntax

```
package ajia.security;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SecurityAspect {
    private Authenticator authenticator = new Authenticator();

    @Pointcut(
        "execution(* ajia.messaging.MessageCommunicator.deliver(..))")
    public void secureAccess() {}

    @Before("secureAccess()")
    public void secure() {
```

1 Aspect declaration

2 Pointcut declaration

3 Advice

```
        System.out.println("Checking and authenticating user");
        authenticator.authenticate();
    }
}
```

Following are the differences between listings 2.3 and 2.7:

1. Instead of using the `aspect` keyword, you use a class annotated with an `@Aspect` annotation. The ajc compiler, which understands the semantics associated with the `@Aspect` annotation, uses this information to treat the class as if it's an aspect.
2. Similarly, the `@Pointcut` annotation marks an empty method as a pointcut. You must specify the pointcut expression—the same expression as in the version using the traditional syntax (except for the use of the fully-qualified type name for `ajia.messaging.MessageCommunicator`). The name of the method serves as the pointcut name.
3. The `@Before` annotation marks a regular method as a before advice. The body of the method consists of the advice logic—this code is executed when a matching join point is executed.

First, let's compile the code using javac and execute the resulting code. Note that annotations such as `@Aspect`, `@Pointcut`, and `@Before` are contained in aspectjrt.jar and must be on your classpath regardless of whether you compile the aspects with ajc or javac:

```
> javac ajia\messaging\MessageCommunicator.java
➥   ajia\security\SecurityAspect.java ajia\security\Authenticator.java
➥   ajia\security\AuthenticationException.java ajia\main\Main.java
> java ajia.main.Main
Wanna learn AspectJ?
Harry, having fun?
```

The code compiled fine, but the aspect had no effect on the output. This may not surprise you; after all, javac has no idea of the meaning of annotations such as `@Aspect` and `@Pointcut`. In general, javac is aware of only a few standard annotations (such as `@Override` and `@SuppressWarnings`). You need to include an aspect weaver somewhere between compiling the source code and executing the byte code in the VM. The simplest way to use the @AspectJ syntax is to use the ajc compiler instead of javac. Alternatives include binary weaving, where code compiled using javac is then woven using ajc; and load-time weaving (LTW), where classes are woven as they're being loaded into the VM. We'll discuss these alternatives in the next section.

Now, let's compile the same sources using ajc:

```
> ajc -source 5 ajia\messaging\MessageCommunicator.java
➥   ajia\security\SecurityAspect.java ajia\security\Authenticator.java
➥   ajia\security\AuthenticationException.java ajia\main\Main.java
> java ajia.main.Main
Checking and authenticating user
Username: ajia
Password: ajia
Wanna learn AspectJ?
Checking and authenticating user
Harry, having fun?
```

The output is the same as that produced by the aspect in listing 2.3. (Note the `-source 5` option to allow Java 5 constructs such as annotations.)

The value proposition of @AspectJ syntax is a simplified adoption curve. The aspect's syntax is expressed using Java constructs, which reduces the mental block commonly associated with using yet another language. You're still using plain Java! In addition, tools such as compilers, IDEs, and code-coverage utilities work more easily with the @AspectJ syntax, because they're working with plain Java code.

So far, we've focused on the AspectJ syntax. Now, let's look at another puzzle piece that completes AspectJ: the weaver.

2.4 Weaving mechanisms

A weaver needs to weave together classes and aspects so that advice gets executed, inter-type declarations affect the static structure, and weave-time declarations produce warnings and errors. AspectJ offers three weaving models:

- Source weaving
- Binary weaving
- Load-time weaving

Regardless of the weaving model used, the resulting execution of the system is identical. The weaving mechanism is also orthogonal to the AspectJ syntax used; any combination of weaving mechanism and AspectJ syntax will produce identical results. In this section, we'll examine the weaving models offered by AspectJ. We'll revisit this topic in chapter 8.

2.4.1 Source weaving

In *source weaving*, the weaver is part of the compiler (all the examples in this chapter so far have used source code weaving). The input to the weaver consists of classes and aspects in source-code form. You can write the aspects in either the traditional syntax or the @AspectJ syntax.

The weaver, which works in a manner similar to a compiler, processes the source and produces woven byte code. The byte code produced by the compiler is compliant with the Java byte-code specification, which any standard compliant VM can execute.

Essentially, when used in this manner, ajc replaces javac. But note that unlike javac, ajc requires that all sources be presented together if you want woven byte code. If you present sources separately, the resulting byte code can be used as input for binary weaving or load-time weaving, discussed next.

2.4.2 Binary weaving

In *binary weaving*, input to the weaver—classes and aspects—is in byte-code form. The input byte code is compiled separately using the Java compiler or the AspectJ compiler. For example, you can use jar files or class files produced using the Java compiler.

Let's see binary weaving in action in a step-by-step manner. The goal is to compile classes and aspects without weaving and then weave the resulting binary (.class) files.

Binary weaver and linker

If you're familiar with languages such as C and C++, think of the weaver as a linker, which is a more accurate comparison. Much the same way a linker takes object files or libraries compiled using a compiler as input to produce an executable or another library, the weaver takes files containing byte code as input and produces woven byte code.

STEP 1: COMPILING THE JAVA SOURCES

Compile the code from listings 2.1 and 2.2 using javac. You could use ajc; but to illustrate the effect of binary weaving clearly, we're staying away from ajc. Use the `-d` option to specify the destination directory for the classes, to help you better understand the effect:

```
> javac -d classes ajia\messaging\MessageCommunicator.java
➥ ajia\security\Authenticator.java ajia\security\AuthenticationException.java
➥ ajia\main\Main.java
```

Unsurprisingly, executing the `Main` class shows that there is no effect of the aspect:

```
> java -classpath classes ajia.main.Main
Wanna learn AspectJ?
Harry, having fun?
```

STEP 2: COMPILING THE ASPECT

Next, compile the @AspectJ-styled aspect from listing 2.7, directing its output to a different directory:

```
> javac -d aspects ajia\security\SecurityAspect.java
```

If you wanted to use the traditional aspect from listing 2.3, you'd have to compile it using ajc:

```
> ajc -d aspects ajia\security\SecurityAspect.aj
```

Executing the `Main` class shows that the output still doesn't have any effect on the aspect:

```
> java -classpath classes;aspects ajia.main.Main
Wanna learn AspectJ?
Harry, having fun?
```

STEP 3: WEAVING THE ASPECT

To weave aspects into classes in binary form, you use binary weaving:

```
> ajc -inpath classes;aspects -aspectpath aspects -d woven
```

The `-inpath` option specifies the path to the classes that are weaving targets. Because you used javac to compile the aspects, you must also pass those to `-inpath` so that ajc can add the necessary support methods. The `-aspectpath` option specifies the path to the aspects to be woven in.

Executing the `Main` class shows that you've restored security to the system. Because you pass an explicit `-classpath` option, you must add the `CLASSPATH` you've set to make the AspectJ runtime available to the VM:

```
> java -classpath woven;%CLASSPATH% ajia.main.Main
Checking and authenticating user
Username: ajia
Password: ajia
Wanna learn AspectJ?
Checking and authenticating user
Harry, having fun?
```

Binary weaving can also take a combination of source and byte-code form as input (classes or aspects). For example, you may have classes compiled into a jar file and aspects available in source form.

An extension of binary weaver is load-time weaving.

2.4.3 Load-time weaving

A load-time weaver takes input in the form of binary classes and aspects, as well as aspects and configuration defined in XML format. A load-time agent can take many forms: a Java VM Tools Interface (JVMTI) agent, a classloader, or a VM- and application server–specific class preprocessor, which weaves the classes as they're loaded into the VM.

Let's use LTW to weave in the `SecurityAspect`, starting with the output of the first two steps in the previous section. LTW needs an XML file that specifies the weaving configuration. AspectJ supports a few locations for such a file; you'll use one of them by naming the file aop.xml and placing it in a directory named META-INF in a classpath component. Listing 2.8 shows the minimal XML file that serves this purpose.

Listing 2.8 aop.xml file specifying the configuration for LTW

```
<aspectj>
    <aspects>
        <aspect name="ajia.security.SecurityAspect"/>
    </aspects>
</aspectj>
```

This configuration instructs the weaver to weave in the `SecurityAspect`. As you'll see later in chapter 8, aop.xml can include a lot more configuration information, including pointcut definitions.

You enable LTW by including the `-javaagent` option when starting the VM:

```
> java -classpath classes;aspects
➥     -javaagent:%ASPECTJ_HOME%\lib\aspectjweaver.jar ajia.main.Main
Checking and authenticating user
Username: ajia
Password: ajia
Wanna learn AspectJ?
Checking and authenticating user
Harry, having fun?
```

The `-javaagent` option specifies that the aspectjweaver.jar file that comes as part of the AspectJ distribution should be used as the JVMTI agent. The weaver uses the information in the aop.xml file in listing 2.8 and weaves the aspects specified in the `<aspects>` section into classes as they're being loaded into the VM.

By now, you must be wondering how AspectJ performs its magic. In the next section, we'll take a quick look at how the source files are compiled into the byte code.

Spring-driven LTW

Spring 2.5 introduces Spring-driven LTW, an alternative way to configure AspectJ LTW for Spring applications. With it, for certain application and web servers, you can avoid modifying the launch script (required to specify the `-javaagent` option); instead, you modify the application context to express the desire to use LTW, and Spring handles the rest. We'll examine this weaving option in chapter 9.

2.5 AspectJ weaving: under the hood

Because the byte code produced by the AspectJ weaver must run on any compliant Java VM, it must adhere to the Java byte-code specification. This means the weaver must map crosscutting elements to Java constructs. In this section, we'll outline how the different elements in an AspectJ program map to pure Java byte code. Note that the discussion that follows presents a simplified view of AspectJ code transformation into pure Java byte code.

Here are the typical ways the AspectJ weaver maps various crosscutting elements to pure Java:

- Aspects map to classes, with each data member and method becoming the members of the class representing the aspect.
- Pointcuts are intermediate elements that map to methods. They may have associated auxiliary methods to help perform matching at runtime.
- Advice usually maps to one or more methods. The weaver inserts calls to these methods at *potential* locations matching the associated pointcut.
- Inter-type declarations of fields and methods are added directly to target classes.
- Weave-time warnings and errors have no real effect on byte code (they're stored in a binary form so that they can be applied through binary weaving). They cause the compiler to print warnings or abort compilation when there's an error.

Each mapped element carries annotations that help the weaver use the crosscutting information, even for aspects presented to it in byte code form. The annotations also help bring symmetry to the @AspectJ syntax discussed in section 2.3.

WARNING Thinking about language semantics in terms of transformed code helps take the mystery out of AspectJ. It also makes you appreciate the hard work the AspectJ weaver performs—and the hard work you no longer need to perform! But such thinking can bog you down in the details of the transformed code. A better approach is to begin thinking in terms of language semantics instead of implementation.

In light of this information, let's see how aspects and classes look after passing through the AspectJ weaver. Note that the weaver produces byte code and not Java code, as shown here. We're showing you this code only to give you an idea of the source code that is roughly equivalent to the byte code produced.

2.5.1 The compiled aspect

First, let's examine the code in a class that is equivalent to `SecurityAspect` from listing 2.3:

```
package ajia.security;

@Aspect
public class SecurityAspect {
    private Authenticator authenticator = new Authenticator();

    public static final SecurityAspect ajc$perSingletonInstance;

    @Pointcut(
        "execution(* ajia.messaging.MessageCommunicator.deliver(..))")
    void ajc$pointcut$$secureAccess$76() {}

    @Before("secureAccess()")
    public final void ajc$before$ajia_SecurityAspect$1$e248afa3() {
        System.out.println("Checking and authenticating user");
        authenticator.authenticate();
    }

    static {
        SecurityAspect.ajc$perSingletonInstance = new SecurityAspect();
    }

    ... method aspectOf() and hasAspect() (discussed in chapter 6) ...
    ... aspect initialization code ...
}
```

`SecurityAspect` is mapped to a class of the same name. By default, an aspect is a singleton, and users don't instantiate it explicitly. The static block in the aspect ensures that the singleton aspect instance is created as soon as the `SecurityAspect` class is loaded into the system—typically, during the execution of code that refers to the aspect. The pointcut is mapped to the `ajc$pointcut$$secureAccess$76()` element. The before advice is mapped to the `ajc$before$ajia_SecurityAspect$1$e248afa3()` method, whose body is identical to the advice body. Notice the annotations on the methods; they make the code identical to that produced by compiling an equivalent @AspectJ aspect. The AspectJ weaver weaves calls to these methods into the advised code, as you'll see next.

2.5.2 The woven class

Now, let's see the equivalent code for the `MessageCommunicator` class (from listing 2.1), after it has been woven with `SecurityAspect`:

```
package ajia.messaging;

import ajia.security.SecurityAspect;

public class MessageCommunicator {
    public void deliver(String message) {
```

```
        SecurityAspect.aspectInstance.
            ajc$before$ajia_SecurityAspect$1$e248afa3();
        System.out.println(message);
    }

    public void deliver(String person, String message) {
        SecurityAspect.aspectInstance.
            ajc$before$ajia_SecurityAspect$1$e248afa3();
        System.out.print(person + ", " + message);
    }
}
```

Recall that the `deliverMessage()` pointcut in `SecurityAspect` is defined to select both of the overloaded `deliver()` methods in `MessageCommunicator`. Accordingly, the `ajc$before$ajia_SecurityAspect$1$e248afa3()` call on the aspect instance `SecurityAspect.aspectInstance` is made from both methods.

Performance implications of AspectJ weaving

"How does it affect the performance of woven code?" is perhaps the most commonly asked question about AspectJ. The inquiring mind wants to know how a hand-woven implementation of crosscutting functionality compares with that implemented with AspectJ.

The code produced by the weaver answers this question well. Because the weaver encapsulates the advice in a method and calls it from appropriate places, there is virtually no overhead from the AspectJ weaver. Furthermore, because advice is well isolated in one place, you can easily add optimizations to that code—something you'd cringe at implementing in hundreds of places.

If you're looking for more details about weaving and its performance implications, read "Advice Weaving in AspectJ" by Erik Hilsdale and Jim Hugunin (http://hugunin.net/papers/aosd-2004-cameraReady.pdf).

We have examined how AspectJ operates under the hood. Now, let's look at a different weaving possibility. The Spring Framework, starting with version 2.0, offers integration with @AspectJ, but without using the AspectJ weaver you've seen so far.

2.6 Spring AspectJ integration

Spring, the most widely used lightweight framework for developing enterprise applications, offers its own form of AOP to modularize crosscutting concerns typically seen in enterprise applications. It uses the proxy design pattern to intercept the execution of methods on the target object. Due to the use of proxies, it exposes method execution join points only for objects created by the Spring container (commonly known as *Spring beans*). But it is a pragmatic solution for its target applications.

Starting with version 2.0, Spring offers several options to leverage AspectJ's power in an incremental manner. For example, it lets you use AspectJ pointcut expressions

in addition to its own pointcut expressions. In this section, we'll preview Spring AspectJ integration for aspects written in @AspectJ syntax. This will be helpful to you because you'll see many Spring-based examples in the book.

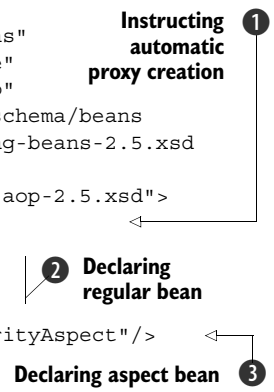
Let's continue to use the `MessageCommunicator` class from listing 2.1 and the annotation-style aspect from listing 2.7. First, you'll write a minimum application context file to define the needed configuration (see listing 2.9). Please read the Spring Framework documentation for details of the syntax of the application configuration file.

Listing 2.9 Defining the application context (applicationContext.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    <aop:aspectj-autoproxy/>

    <bean id="messageCommunicator"
          class="ajia.messaging.MessageCommunicator"/>

    <bean id="securityAspect" class="ajia.security.SecurityAspect"/>
</beans>
```



1. The `<aop:aspectj-autoproxy>` element tells Spring to automatically create proxies for the beans that can be advised by aspects (which are declared as beans as in 3).
2. The `<bean>` element creates a bean for the `MessageCommunicator` class.
3. The `<bean>` element creates a bean corresponding to the aspect declared using the @AspectJ syntax.

Next, you modify the `Main` class from listing 2.2 to create the application context and obtain the `messageCommunicator` bean from it. Then, you use the bean to deliver a few messages (see listing 2.10).

Listing 2.10 Using the Spring container application context

```
package ajia.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import ajia.messaging.MessageCommunicator;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context
            = new ClassPathXmlApplicationContext("applicationContext.xml");
        MessageCommunicator messageCommunicator
            = (MessageCommunicator)context.getBean("messageCommunicator");
```

```
        messageCommunicator.deliver("Wanna learn AspectJ?");
        messageCommunicator.deliver("Harry", "having fun?");
    }
}
```

Now, let's compile all these classes and execute the `Main` class:

```
> javac ajia\messaging\*.java ajia\main\*.java ajia\security\*.java
> java ajia.main.Main
Checking and authenticating user
Username: ajia
Password: ajia
Wanna learn AspectJ?
Checking and authenticating user
Harry, having fun?
```

You see the same output as when you use an AspectJ weaver. Behind the scenes, the Spring container creates a proxy around the `messageCommunicator` bean and intercepts methods called on it according to the `SecurityAspect` defined using the @AspectJ syntax. As a result, you can use the aspects written using @AspectJ in a Spring application without needing the AspectJ weaver. We'll examine the details in chapter 9.

So far in this chapter, you've used only command-line tools to work with aspects. But in real life, virtually no one works without a good IDE. You also need support for documentation so the crosscutting information is available outside the IDE. In the next section, we'll look at the logistical support provided by AspectJ.

2.7 AspectJ logistics overview

AspectJ offers a complete set of tools ranging from a compiler to IDE support. We've already examined the compiler and load-time weaver. Let's look at a few important tools in more detail. This will help you when you download the book's source code and try out the examples.

2.7.1 IDE integration

IDE support offers an integrated approach to editing, compiling, executing, and debugging tasks. AspectJ eases the development process by providing integration with Eclipse. The integration with the IDE is achieved through the AspectJ Development Tools (AJDT) plug-in. Using this integration, you can edit, compile, and debug your project the same way you would a project written in Java.

TIP AJDT offers plug-ins for various versions of Eclipse. You can download an appropriate version from http://www.eclipse.org/ajdt/downloads. Better still, you can use the free SpringSource Tools Suite (STS), which is an Eclipse distribution that includes many useful plugins, including AJDT, targeted for developing Spring-based applications.

Figure 2.1 shows how the example in this chapter looks in the Eclipse IDE. Note that the Spring IDE, an Eclipse plug-in for developing Spring-based applications, offers similar functionality for Spring AspectJ integration, as discussed in section 2.6.

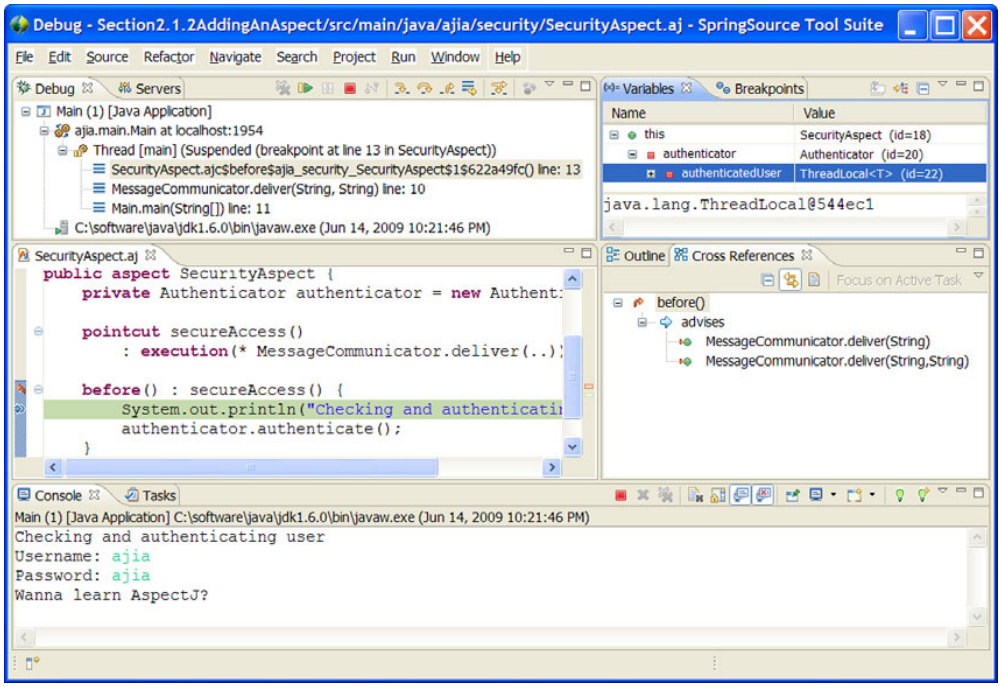


Figure 2.1 Developing applications using Eclipse-AspectJ integration (using STS distribution). The overall feel for editing, building, and debugging is like a plain Java project. The IDE also shows how crosscutting elements affect the various parts of the system.

NOTE AspectJ itself uses AspectJ for its implementation. And AJDT uses AspectJ to weave into Eclipse's Java Development Tools (JDT), to offer AspectJ integration.

In figure 2.1, Eclipse shows the standard views, along with a *Cross References* view that shows how an advice applies to different parts of the code. This view helps in diagnosing problems where advice doesn't apply to the expected set of methods. If you see any mismatch, you can modify the corresponding pointcuts and reexamine the list of the advised methods.

The AJDT plug-in also offers a way to visualize the big picture effects of crosscutting concerns using the *Visualiser*.

What about other IDEs?

Earlier versions of AspectJ supported other IDEs besides Eclipse—NetBeans, JBuilder, and Java Development Environment for Emacs (JDEE)—by offering open source plug-ins for each of them. But those plug-ins haven't been kept up to date. (An effort is currently under way to revive the NetBeans plug-in. See http://www.jroller.com/ramlog/entry/using_the_aspectj_plug_in1 for details.)

What about other IDEs? *(continued)*

This is perhaps a reflection of the market reality—no other IDE is as popular as Eclipse.

One IDE that has a good market and mind share is IntelliJ IDEA. Although an AspectJ plug-in is available for it (http://intellij.expertsystems.se/aspectj.html), it works only with the @AspectJ syntax.

Note that for any IDE without direct support for AspectJ, the possibility of using the @AspectJ syntax makes the lack of direct AspectJ support a less pronounced issue. Because the code is still plain Java, as far as the IDE is concerned, you can edit code, take advantage of code completion, and so on. If the IDE allows you to replace the default compiler, you can replace it with ajc. If not, you can introduce a post-compilation step (often supported in IDEs) to run ajc to perform binary weaving. Debugging works fine too, because the method representing advice is still executed as if there was a real call.

When you use the @AspectJ syntax in an IDE that doesn't support AspectJ directly, you lose the source-code markers that indicate advice applicability, along with the crosscutting references view. You can alleviate this problem to an extent by using ajbrowser (which comes as a part of the AspectJ distribution)—a standalone tool that shows how weaving rules affect different parts of a program.

Although Eclipse IDE integration shows crosscutting information in a crosscutting references view, you'll often need the same information in a static document.

2.7.2 AspectJ documentation tool

The AspectJ documentation tool—ajdoc—extends Javadoc to provide crosscutting information in static form. You invoke ajdoc in a way similar to Javadoc:

```
> ajdoc -source 5 ajia\messaging\*.java ajia\main\*.java ajia\security\*.aj
➥   ajia\security\*.java ajia\profile\*.aj ajia\track\*.aj
```

It produces HTML files similar to the ones produced by Javadoc, except the elements carry additional information showing how aspects and classes interact (see figure 2.2).

The output produced by ajdoc offers a simple way to examine the crosscutting structure without needing an IDE. Because ajdoc isn't tied to a specific IDE, you can use it alongside the IDE of your choice even if it doesn't support AspectJ directly.

The tools offered by AspectJ are indispensable during the development process. If you haven't already done so, download and install AspectJ and AJDT as well as this book's source code.

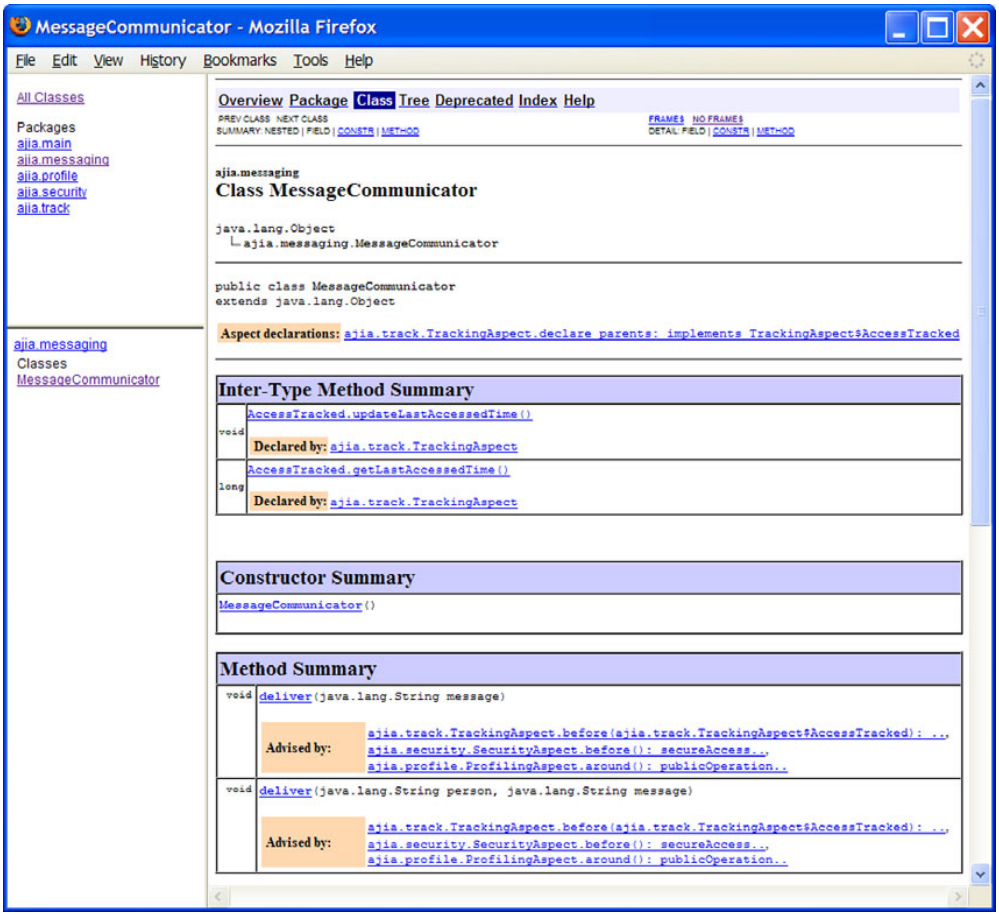


Figure 2.2 Output produced by ajdoc, which works similarly to Javadoc. In addition to the regular documentation, it provides markers to show crosscutting information.

2.8 Summary

AspectJ adds AOP constructs—pointcuts, advice, aspects, inter-type declarations, and weave-time declarations—to Java, creating a powerful language that facilitates modularizing crosscutting concerns while retaining the benefits of Java, such as platform independency. You can add new functionality without changing code in the core modules and without those modules being aware of what you've done.

Aspect-oriented programming in AspectJ is simple: choose where you want to crosscut, choose the kind of action you need to perform, and programmatically specify both of them. The AspectJ language exposes the necessary join points in a Java program. Pointcuts let you choose the join points you want to affect, and advice allows you to specify the action at those join points. The static crosscutting mechanism enables you to modify the static structure of the system. AspectJ complements—and

doesn't compete with—Java. By utilizing its power to modularize the crosscutting concerns, Java programmers no longer need to recode multiple modules when implementing or changing a crosscutting concern.

The new additions to AspectJ—the various syntax options, weaving models, and integration with Spring—make adopting AspectJ much easier. It's typical for Spring developers to start with the Spring AspectJ integration, using proxy-based weaving, and to learn the power of AOP and AspectJ through experience. Then, they often move to advanced AOP techniques using AspectJ weaving, often along with the @AspectJ syntax. At this point, load-time weaving is often a common choice due to its simplicity in getting started. Later, when they're looking for even more advanced usages, they may go for the traditional syntax along with a combination of source, binary, and load-time weaving.

In this chapter, we studied the core AspectJ concepts from 20,000 feet. It's now time to get a closer view. The next chapter introduces the join-point model that is at the heart of AOP. The three chapters that follow will discuss dynamic crosscutting, static crosscutting, and aspects. We'll then proceed to examine the @AspectJ syntax, weaving mechanisms, and Spring integration. All this information will enable you to write aspects that are useful in complex Java applications, which we'll explore in part 2 of the book.