Foundations of AOP for J2EE Development

Renaud Pawlak, Lionel Seinturier, and Jean-Philippe Retaillé

Apress®

Foundations of AOP for J2EE Development

ISBN: 1-59059-507-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Houman Younessi
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Translator: Chelsea Creekmore
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editors: Linda Harmony, Ami Knox, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor and Artist: Wordstop Technologies Pvt. Ltd., Chennai
Proofreader: Elizabeth Berry
Indexer: John Collin
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 3

AspectJ

In the previous chapter, we presented the basic concepts of AOP with the notions of the aspect, the pointcut, the joinpoint, and the advice code.

In this chapter, we will illustrate the way that these concepts are implemented in AspectJ. The syntax and concepts presented here correspond to version 1.2.1 of the language.

Gregor Kiczales and his team, who are credited with the creation of AOP at the Palo Alto Research Center (PARC), are responsible for the invention and development of AspectJ—which is now the leading tool for AOP. The first versions of AspectJ were released in 1998 and, as of December 2002, the AspectJ project has left PARC and joined the open-source Eclipse community. Today, AspectJ is the most widely used aspect-oriented language.

THE HISTORY OF ASPECTJ

The histories of AspectJ and AOP are closely related. AspectJ has always been considered by Gregor Kiczales as the project that would illustrate the concepts of AOP. Although the notion of the aspect dates back to 1996, and the first versions of AspectJ were released in 1998, the ideas and research that culminated in AOP date back to before this time. Research in reflection in the 1980s and work on open implementations in the 1990s served as background for the development of AOP.

Invented in 1984 by Brian Smith, and studied and popularized by Patricia Maes in 1997, reflection is a programming technique that introduces a two-level architecture. The first level, called the *base level*, consists of the application. The second level, called the *meta level*, controls and supervises the base level. Although the notions of the aspect and the meta level differ, they share a common goal: to separate business functionalities from technical concerns. This separation aims to result in better modularization of programs. Prior to inventing the concept of the aspect, Kiczales spent time conducting research in the domain of reflection. In 1991, he was coauthor of *The Art of the Metaobject Protocol* (MIT Press, 1991).

The founding document of AOP was published and presented in 1997 by Kiczales during the European Conference on Object-Oriented Programming (ECOOP). Presentations had been held previously, in 1996, but the 1997 article is considered seminal. Simultaneously, the first prototypes of AOP languages appeared in 1996–97.

Christina Lopez, a member of Kiczales's team at the time and an important contributor, developed the D language and its implementation, DJava. The D language contained two types of aspects: distribution and concurrency-management. Soon after, Lopez and Kiczales realized that this new approach could be generalized and applied to other aspects. A general-purpose language that could implement any kind of aspect was needed.

In 1998, Kiczales and his team made the decision to switch from D to AspectJ. Soon after, the first implementations of AspectJ were released. At almost the same time, Aspect-Oriented Tcl Object System (A-TOS),

which was the first prototype of Java Aspect Components (JAC), was implemented. Since then, several versions of AspectJ have been released, and each one has included new features and/or bug fixes. The first major version of AspectJ, designated version 1.0, was released in November 2001. This was also the year during which AOP was fully recognized by the international computer-science community. A special edition of the leading journal, *Communications of the ACM*, was devoted to AOP.

In December 2002, the AspectJ project left PARC and joined the open-source Eclipse community. Since then, the AspectJ Development Tools (AJDT) plug-in has been developed. It enables you to write, compile, and run an aspect-oriented program within the IBM Eclipse IDE.

A First Application with AspectJ

This section presents a simple example of an aspect-oriented application with AspectJ. This example introduces the syntax for writing aspects, pointcuts, and advice code.

The example is an order-management application that manages client orders. The application implements a trace aspect, which traces the execution of the application and determines which methods are called and the order that they are called in.

The Order-Management Application

The order-management application enables a client to add items to an order and to compute the amount of that order. References to the items and their prices are stored in a catalog.

This application defines three classes: `Customer`, `Order`, and `Catalog`. The `Customer` class, which is shown in Listing 3-1, is the main entry point of the application.

Listing 3-1. *The Customer Class for the Order-Management Application*

```
package aop.aspectj;
public class Customer {

  public void run() {
    Order myOrder = new Order();
     myOrder.addItem("CD",2);
     myOrder.addItem("DVD",1);
     double amount = myOrder.computeAmount();
     System.out.println("Order amount: US$"+amount);
  }

  public static void main(String[] args) {
    new Customer().run();
  }
}
```

The `main` method creates an object from the `Customer` class and calls the `run` method. The latter method creates an order (a `myOrder` object), calls the `addItem` method two times with a reference and a quantity, computes the amount of the order, and displays the amount.

The orders are managed by the `Order` class, which is shown in Listing 3-2.

Listing 3-2. *The Order Class for the Order-Management Application*

```
package aop.aspectj;

import java.util.*;

public class Order {

  private Map items = new HashMap();

  public void addItem(String reference,int quantity) {
    items.put(reference,new Integer(quantity));
    System.out.println(
      quantity+" item(s) "+reference+ " added to the order" );
  }

  public double computeAmount() {
    double amount = 0.0;
    Iterator iter = items.entrySet().iterator()
    while ( iter.hasNext() ) {
      Map.Entry entry = (Map.Entry) iter.next();
      String item = (String) entry.getKey();
      Integer quantity = (Integer) entry.getValue();
      double price = Catalog.getPrice(item);
      amount += price*quantity.intValue();
    }
    return amount;
  }
}
```

The `Order` class records the items and the ordered quantities in a hash map (in the `items` field). This map is indexed with the item references. The associated map values are the quantities of the ordered items. The `addItem` method adds an item to the order and displays a message that reports on the operation. The `computeAmount` method iterates over the ordered items, determines each price, and returns the total amount of the order.

The price of each item is determined by the `Catalog.getPrice` method, which is shown in Listing 3-3.

Listing 3-3. *The Catalog Class for the Order-Management Application*

```
package aop.aspectj;

import java.util.*;

public class Catalog {

    private static Map priceList = new HashMap();
```

```
    static {
      priceList.put( "CD", new Double(15.0) );
      priceList.put( "DVD", new Double(20.0) );
    }

    public static double getPrice( String reference ) {
      Double price = (Double) priceList.get(reference);
      return price.doubleValue();
    }
}
```

The `Catalog` class records the price of each item in the `priceList` hash map. This table is indexed by the item references. The associated values are the item prices. The static code block initializes the `priceList` map with two items: a CD that costs US$15 and a DVD that costs US$20. The `getPrice` method returns the price of the item that was given as a parameter.

Execution

The order-management application has so far been written purely in Java. The principle of AOP is to leave applications unpolluted by code that is not related to the main functionality. As a result, AOP allows you to focus on the core business, which in this case is the management of orders.

The nonfunctional concerns, such as security, tracing, and the management of transactions, can be independently added through aspects. The output of the order-management application is shown in Listing 3-4.

Listing 3-4. *The Output of the Order-Management Application*

```
2 item(s) CD added to the order
1 item(s) DVD added to the order
Order amount: US$50.0
```

A First Trace Aspect

The business core of the application has now been developed. We will now proceed with the development of a first aspect. This aspect, developed separately from the classes, monitors each ordered item by displaying messages before and after the `addItem` method, which is defined in the `Order` class.

The AspectJ code for this aspect is shown in Listing 3-5.

Listing 3-5. *A First Trace Aspect for the Order-Management Application*

```
1 package aop.aspectj;
2
3 public aspect TraceAspect {
4
5   pointcut toBeTraced():
```

```
 6     call( public void Order.addItem(String,int) );
 7
 8   void around(): toBeTraced() {
 9     System.out.println("-> Before calling addItem");
10     proceed();
11     System.out.println("<- After calling addItem");
12   }
13 }
```

The AspectJ language extends the Java syntax with new keywords. In Listing 3-5, the first new keyword you encounter is `aspect`. Like a class, an aspect is named (in this case, `TraceAspect`) and can be defined in a package (in this case, `aop.aspectj`). An aspect can also be extended through inheritance, as you will see in the "Aspect Inheritance" section later in this chapter.

In the philosophy held by the AspectJ creators, an aspect is a software entity that is largely similar to a class—in that both define a piece of code that abstracts and modularizes a concern. Although the concern is crosscutting in the case of an aspect, classes and aspects belong to the same level and must obey—as often as possible—the same rules.

In the examples that are distributed with AspectJ, the similarities go so far that classes and aspects[1] use the same `.java` extension.

Aspects define pointcuts and advice code; the following sections describe these elements.

A First Pointcut Descriptor

In an aspect, the `pointcut` keyword is used to define a *pointcut descriptor*. Pointcut descriptors can be named. In the example of the `TraceAspect` aspect, the pointcut descriptor is named `toBeTraced`.

Each pointcut descriptor is an expression (see line 6 in Listing 3-5) that denotes the set of associated joinpoints. Several different types of joinpoints can be used. However, this is not the case in the example of the `toBeTraced` pointcut descriptor, in which `call` is the only type of joinpoint.

The `call` joinpoint designates the points where a method is called. The signature of the called method is given in parentheses. The signature refers to the `addItem` method, which is defined in the `Order` class, has two parameters of type `String` and `int`, returns `void`, and is `public`. This signature is so precise that only one method fits this pointcut. In the "Wildcards" section later in this chapter, you will see that wildcards can be used to match several methods with the same pointcut descriptor.

The `toBeTraced` pointcut is associated with the `addItem` method; however, this does not mean that only one joinpoint exists for this pointcut. In fact, all the locations where the method is called match the pointcut. In the example in Listing 3-1, when `addItem` is called two times in the `run` method of the `Customer` class, both locations match the pointcut.

In general, the number of joinpoints associated with a pointcut is not predetermined—the number varies depending on the particular pointcut and its application. It is possible to write

1. In many applications, there is a clear difference between business functionalities and crosscutting functionalities. The former are implemented in classes and the latter in aspects. It can then be interesting to clearly state this difference by giving different extensions to file names. You could choose, for instance, `.aj` or `.ajava` for files containing aspects, and keep `.java` for files containing classes.

a pointcut descriptor with no associated joinpoints and, in this case, the aspect will not modify the application. Although this is possible, it is highly unlikely that such a pointcut descriptor would be useful—most of the time, this type of pointcut descriptor corresponds to an error.

Tools are available to detect these types of cases and avoid them. For example, the AJDT plug-in provides a view that gives the corresponding joinpoints of each defined pointcut. Figure 3-1 gives an illustration of this view. As shown in the `advises` node, the `toBeTraced` pointcut descriptor is associated with two joinpoints for the `Customer` class. If there were no associated joinpoints, the correctness of the pointcut would have been questioned.

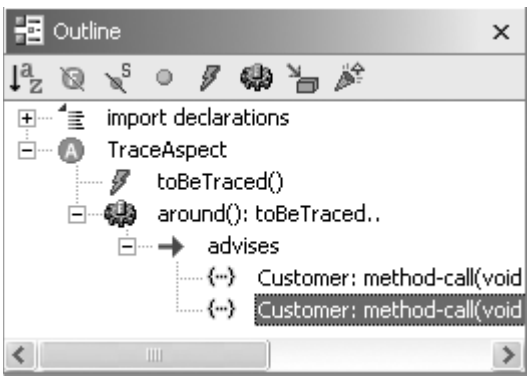

Figure 3-1. *The AJDT view of joinpoints associated with a pointcut descriptor*

A First Piece of Advice Code

The `TraceAspect` aspect defines only one piece of advice code. (See line 8 in Listing 3-5). Three types of advice code exist: "before," "after," and "around." For the first type, the advice code is executed before each joinpoint that is associated with the pointcut; for the second type, it is executed after; for the third type, it is executed before and after.

For any type of advice code, the joinpoints are designated by a pointcut, and each piece of advice code is associated with a pointcut. In the `TraceAspect` aspect, the advice code is of the "around" variety, and the associated pointcut is `toBeTraced`. The code is given between brackets. In this example, the "around" advice code is executed before and after the calls to the `addItem` method.

In the `TraceAspect` example, AspectJ expects the methods included in the pointcut to define the return type as `void`. It follows that the `addItem` return type is `void`.

Advice code is a regular block of code. Any existing Java instruction, such as a method call, a variable assignment, a `for` statement, a `while` statement, or an `if` statement, can be used. AspectJ adds the new keyword `proceed` to this list. When reached, this keyword triggers the execution of the joinpoint. The instructions prior to `proceed` are executed before the joinpoint, and those following `proceed` are executed after. Hence, `proceed` delimits a "before" part and an "after" part in the advice code.

The execution of a program with "around" advice code is as follows:

1. The program is started.
2. Just before a joinpoint, the "before" part is executed.

3. The `proceed` method is called.
4. The corresponding joinpoint is executed.
5. The "after" part is executed.
6. Program execution resuming, just after the joinpoint.

The call to `proceed` is optional. Certainly, it is valid for "around" advice code to not call `proceed`. In such a case, the code associated with the joinpoint is not executed. After the execution of the advice code, the program execution resumes directly after the joinpoint.

Advice code can call `proceed` in some cases and not in others. This can be seen in the case of a security aspect that controls access to a method. If the user is authenticated, the call is authorized and the aspect calls `proceed`. If the user does not own sufficient rights, the call is rejected and the security aspect does not call `proceed`. In such a case, the joinpoint is not executed.

The "before" and "after" advice code blocks do not need to call `proceed`. They are designed to be executed before and after joinpoints. AspectJ raises an error if `proceed` appears in the body of "before" or "after" advice code.

Compiling

AspectJ is a compile-time weaver (since version 1.2, AspectJ can also weave aspect at load-time). The aspect and class input files are woven together by AspectJ to produce a final application. This application can then be executed as a regular Java application.

To compile the aspect with the three classes of the order-management application, call the `ajc` compiler from the operating-system command shell with the following command:

```
ajc TraceAspect.aj *.java
```

This command produces a set of `.class` files that contain the bytecode of the aspectized application.

The first versions of AspectJ produced Java code. The output of the `ajc` compiler was a Java program in which instructions from aspects and classes were merged. This Java code could then be compiled by the regular `javac` Java compiler. The advantage of this approach was that the produced code could easily be read by developers, who could then study the way that the pointcuts and advice code blocks were implemented.

Despite these benefits, this approach had several drawbacks. The first had to do with performance. The initial translation from Java to Java was costly. In addition, a syntax analysis of the Java code was needed before the result of the weaving could be generated—and writing a Java syntax analyzer that is fast enough to manage huge amounts of code is a difficult task. Even more, this task is redundant with the one performed when the produced code is compiled. The second disadvantage had to do with the source code needed to perform the weaving. In some cases, the application was commercial or came from a developer who did not release the source code. As a result, the Java-to-Java compilation could not be applied.

To avoid these difficulties, the new versions of AspectJ can weave bytecode in `.class` or `.jar` files and do not require source code. The `ajc` compiler provides the option `-injar` to use this feature.

Running

The order-management application, which is woven with the `TraceAspect` aspect, can be run by calling the Java virtual machine with the `Customer` class.

To be run in a UNIX or Windows shell, the command is as follows:

```
java aop.aspectj.Customer
```

The output of this command is shown in Listing 3-6.

Listing 3-6. *The Output of the Order-Management Application with the Trace Aspect*

```
-> Before calling addItem
2 item(s) CD added to the order
-> After calling addItem
-> Before calling addItem
1 item(s) DVD added to the order
-> After calling addItem
Order amount: US$50.0
```

In this run, each call to the `addItem` method corresponds to a joinpoint. The execution is trapped, and the advice code defined in the `TraceAspect` aspect applies. This is "around" advice code. The message -> *Before calling addItem* is displayed, the `proceed` instruction is called, the "after" part is executed, and the message -> *After calling addItem* is displayed. Note that the call to `proceed` executes the `addItem` method.

Pointcut Descriptors

In the previous section, we introduced the basic elements of the AspectJ syntax. We showed how to write, compile, and run a first aspect-oriented application.

Although it is simplistic, the order-management application illustrates a major AOP characteristic: the separation of business from technical code. The `Customer`, `Order`, and `Catalog` classes play the role of the business code, whereas the `TraceAspect` aspect implements the technical part of the application. This separation is made possible by the use of pointcuts. Pointcuts allow you to describe where the aspects must be applied in the code. Therefore, pointcuts give you a way of "talking about" the application by designating some of the strategic locations of that application.

In the following sections, we present the syntax of the AspectJ language. We begin with the syntax of pointcuts and follow with the syntax of advice code. You will see how more-generic aspects can be written, how different types of joinpoints can be included, and how more-complex pointcuts can be defined.

Wildcards

The `toBeTraced` pointcut, which was defined in the `TraceAspect` aspect in Listing 3-5, is not generic. It captures calls exclusively to the `addItem` method. In more complex cases, it is important for you to accurately define pointcuts that contain calls to several methods.

AspectJ provides a pattern language, which allows the definition of expressions that implicitly denote several methods or classes. These symbols (*, .., and +) are called wildcards. In the following sections, we explain the principle and definition of these wildcards and give usage examples of them.

Principle

When wildcards are used jointly with the different types of joinpoints that we describe later in this chapter (in the "Joinpoint Types" section), the wildcards act as a powerful syntax capable of describing many pointcuts. In return, this flexibility can lead to pointcut descriptors that are complex and subtle.

Other AOP environments, such as JAC and JBoss AOP (see Chapters 4 and 5), have made different choices. Their pointcut languages define fewer operators and, as a consequence, their expression powers are not as flexible as that of AspectJ. On the other hand, their pointcut descriptors are simpler to write, understand, and debug.

The discussion about the trade-offs between power and simplicity remains open. Yet, in computer science and other domains, the simple solutions are often the most widely used. Similarly, the simplest pointcuts are the ones more frequently reused. Therefore, we use the simple pointcuts to illustrate and promote AOP.

The following sections define the *, .., and + wildcards. They are presented according to the element types (method, class, signature, package, and subtype) that they denote.

Method and Class Names

The asterisk (*) can be used to denote method and class names. You will see in the "Method Signatures" and "Package Names" sections, later in this chapter, that the asterisk can also be used for signatures and package names.

For methods, the asterisk designates "some or all of the methods defined in a class."

The following expression designates all the public methods in the `Order` class that have two parameters of type `String` and `int` and that return `void`:

```
public void aop.aspectj.Order.*(String,int)
```

The asterisk can be used in combination with letters to designate all the method names that contain the substring "Item", for instance. In such a case, the expression is `*Item*`.

As stated, the asterisk can also be used for class names. The following expression designates all the public methods of all the classes in the `aop.aspectj` package that have two parameters of type `String` and `int` and that return `void`:

```
public void aop.aspectj.*.*(String,int)
```

Method Signatures

Names, parameters, return types, and access modifiers (`public`, `protected`, and `private`), can be used in AspectJ pointcut descriptors.

Method parameters can be omitted and replaced by two dots (`..`) to indicate "any parameter." The following expression illustrates the public methods in the `Order` class that have any parameters and that return `void`:

```
public void aop.aspectj.Order.*(..)
```

Hence, the symbol of two dots handles the polymorphism of Java methods.

The return type and the access modifier can be omitted and replaced by the asterisk (*).

The following expression denotes all the methods defined in the `Order` class, regardless of their parameters, return types, and access modifiers:

```
* * aop.aspectj.Order.*(..)
```

In addition, the asterisk that is used to replace the access modifier can be omitted without changing the meaning of the pointcut. Therefore, the following expression is equivalent to the one just presented:

```
* aop.aspectj.Order.*(..)
```

Package Names

The two-dots symbol can also be used with package names. For example, the following expression designates all the methods of any class named `Order` that is in any package of the `aop` hierarchy, regardless of the subpackage level:

```
* aop..Order.*(..)
```

The use of the two-dots symbol between `aop` and `Order` is of great importance. Expressions such as `aop.Order.*(..)` and `aop.*.Order.*(..)` look similar but lead to completely different results. Although both expressions involve the methods defined in an `Order` class, they differ in the following ways:

- The `aop..Order.*(..)` expression denotes the entire hierarchy that starts with the `aop` package.
- The `aop.Order.*(..)` expression denotes only the class that is defined in the `aop` package.
- The `aop.*.Order.*(..)` expression denotes only the classes that are defined in the direct subpackages of the `aop` package.

Subtypes

The last wildcard, the plus sign (+), deals with type hierarchies and designates all the subclasses of a given class. The following expression denotes all the methods of the `Order` class and its subclasses:

```
* aop.Order+.*(..)
```

The plus sign can also be used after an interface name. In such a case, the pointcut designates all the methods of all the classes that implement the interface.

Because the plus sign can be used with either a class or an interface, it is referred to as a *subtyping operator*.

Wildcard-Usage Example

We will illustrate the usage of wildcards by showing a second version of the trace aspect, which is named `TraceAspect2`. The previous version, `TraceAspect`, intercepted only the calls to the

addItem method. The new version is more generic and intercepts the calls to all the methods defined in the Order class.

The code of the TraceAspect2 aspect is shown in Listing 3-7.

Listing 3-7. *Using Wildcards in Pointcut Descriptors*

```
 1 package aop.aspectj;
 2
 3 public aspect TraceAspect2 {
 4
 5   pointcut toBeTraced(): call(* aop.aspectj.Order.*(..));
 6
 7   Object around(): toBeTraced() {
 8      System.out.println("-> Before the call");
 9      Object ret = proceed();
10      System.out.println("<- After the call");
11      return ret;
12   }
13 }
```

The first difference between the TraceAspect and TraceAspect2 aspects is in the definition of the pointcut. (See line 5 in Listing 3-7). The pointcut descriptor is now call(* aop.aspectj.Order.*(..)). All the calls to the methods defined in the Order class are intercepted by the pointcut—whatever the methods' parameters and return types are.

A second difference can be found in the return type of the advice code block. Previously, the type was void. Now, the pointcut intercepts either a method that returns void (for example, addItem) or a method that returns double (for example, computeAmount). Strictly speaking, no common supertype exists for both void and double. As a convention, AspectJ considers that Object (see line 7 in Listing 3-7) is the common supertype for all Java primitive and object types. The result of proceed is stored in the ret variable (see line 9), which is returned at the end of the advice code block (see line 11).

Joinpoint Introspection

In the previous section, you learned that wildcards can be used to write generic pointcut descriptors. With wildcards, a pointcut can be associated with several methods. You will see in this section how to obtain some information about a joinpoint at run time. This is known as the *joinpoint-introspection mechanism*.

The term *introspection* refers to examining the inner cause of a given phenomenon and gaining information about it. In the context of AOP, the phenomenon is the joinpoint, and you want to retrieve information about the part of the actual program that allows the joinpoint to occur. A comparison can be made with the Java java.lang.reflect API that provides, for any given Java program, a description of the program's classes, fields, and methods. The program is then said to have *introspected* to obtain information about itself.

In the context of AOP, the same principle applies. The joinpoint examines itself to gain information about itself. This mechanism is useful for determining, for instance, the method call that caused the joinpoint to occur.

Introspection Syntax

The `thisJoinPoint` keyword implements the introspection mechanism in AspectJ. In regular Java, `this` is the reference to the current object. Similarly, in AspectJ, `thisJoinPoint` is a reference to an object describing the current joinpoint, which implements the predefined `org.aspectj.lang.JoinPoint` interface. Table 3-1 sums up the main methods defined in this interface. Two of them, `getSignature` and `getSourceLocation`, use predefined interfaces, which are `Signature` and `SourceLocation`, respectively. The AspectJ Javadoc documentation gives more details on the methods provided by these interfaces.

Table 3-1. *Main Methods Defined in `org.aspectj.lang.JoinPoint`*

Method Signature	Definition	Comment
`Object[] getArgs()`	Returns the joinpoint arguments	When the joinpoint deals with a method, `getArgs` returns the arguments of the call.
`Signature getSignature()`	Returns the joinpoint signature	The `Signature` interface gives a representation of the signature of the joinpoint. When the joinpoint deals with a method, `Signature` provides methods for retrieving the joinpoint method's name, access modifiers (`public`, `private`, and so on), class, and return type.
`String getKind()`	Returns the joinpoint type	
`SourceLocation getSourceLocation()`	Returns the localization of the joinpoint in the source code	`SourceLocation` is a predefined interface that provides access to the file name, line number, and class where the joinpoint is defined.
`Object getTarget()`	Returns the target object of the joinpoint	When the joinpoint deals with a method, `getTarget` returns the called object.
`Object getThis()`	Returns the source object of the joinpoint	When the joinpoint deals with a method call, `getThis` returns the calling object.

Usage Example

We now illustrate the joinpoint-introspection mechanism with a third version of the trace aspect, which is named `TraceAspect3`. In comparison to `TraceAspect2`, `TraceAspect3` now displays the following for each joinpoint:

- The name of the intercepted method call
- The parameters of the call
- The current object—in other words, the calling object
- The target object—in other words, the called object

The code of the `TraceAspect3` aspect that performs this introspection and displays these four pieces of information is shown in Listing 3-8.

Listing 3-8. *An AspectJ Program for Joinpoint Introspection*

```
package aop.aspectj;

public aspect TraceAspect3 {

pointcut toBeTraced(): call(* aop.aspectj.Order.*(..));

  Object around(): toBeTraced() {
    String methodName = thisJoinPoint.getSignature().getName();
    Object[] args = thisJoinPoint.getArgs();
    Object caller  = thisJoinPoint.getThis();
    Object callee = thisJoinPoint.getTarget();

    System.out.println("-> Method "+methodName+" begins");
    System.out.print("-> "+args.length+" parameter(s) ");
    for (int i = 0; i < args.length; i++)
      System.out.print( args[i]+" " );
    System.out.println();
    System.out.println("-> "+caller+" to "+callee);

    Object ret = proceed();
    System.out.println("<- Method "+methodName+"ends");
    return ret;
  }
}
```

The output of the order-management application with the `TraceAspect3` aspect is shown in Listing 3-9. Before each call to a method from class `Order`, three lines display respectively: the name of the called method, the parameters of the call, and the caller and the called object.

Listing 3-9. *The Output of the AspectJ Program for Joinpoint Introspection*

```
-> Method addItem begins
-> 2 parameter(s) CD 2
-> aop.aspectj.Customer@1cd2e5f to aop.aspectj.Order@19f953d
2 item(s) CD added to the order
-> Method addItem ends
-> Method addItem begins
-> 2 parameter(s) DVD 1
-> aop.aspectj.Customer@1cd2e5f to aop.aspectj.Order@19f953d
1 item(s) DVD added to the order
-> Method addItem ends
-> Method computeAmount begins
-> 0 parameter(s)
-> aop.aspectj.Customer@1cd2e5f to aop.aspectj.Order@19f953d
-> Method computeAmount ends
Order amount: US$50.0
```

Defining Joinpoints

In the previous sections, you learned that wildcards and introspection can be used to write generic pointcuts. Up until this point, we have shown joinpoints that deal only with method calls. In this section, we will present other types of joinpoints.

Joinpoint Types

The joinpoint types provided by AspectJ can deal with methods, fields, exceptions, constructors, static blocks and, finally, advice-code executions.

Methods

For methods, AspectJ defines two types of joinpoints: method calls (defined by the `call` keyword) and method executions (defined by the `execution` keyword). In both cases, an expression must be provided to designate the methods to be called or executed.

The first difference between the `call` and `execution` types concerns the context that the joinpoint occurs in. In the former case, the joinpoint occurs in the context of the calling code, whereas in the latter case, the joinpoint occurs in the context of the called code.

A second difference, which is a direct consequence of the first, is that the values returned by the `getThis` and `getTarget` introspection methods differ. For the `call` type, `getThis` returns a reference to the caller, and `getTarget` returns a reference to the callee. For `execution`, both `getThis` and `getTarget` return a reference to the callee. The `call` type can thus be seen as more general because both the caller and the callee are available.

A method can be associated with a `call` joinpoint and an `execution` joinpoint at the same time. In such a case, the code is executed in the following order:

1. The "before" part of the advice code that is associated with the `call` joinpoint
2. The "before" part of the advice code that is associated with the `execution` joinpoint
3. The method
4. The "after" part of the advice code that is associated with the `execution` joinpoint
5. The "after" part of the advice code that is associated with the `call` joinpoint

Fields

The `get` and `set` joinpoint types intercept the instructions that read and write a field, respectively. These types are useful when you want to implement aspects that manipulate the state of an object. For instance, in the case of a persistence aspect, the state of an object needs to be stored in a file or a database. When intercepted, the read and write operations can be easily redirected to the file or the database.

The `get` and `set` types take an expression as a parameter that denotes the set of fields included in the pointcut. The definition of this expression contains three parts for each field: its type, the class that defines it, and its name. All of these parts can contain wildcards.

The following expression intercepts all the read operations on the `items` field of type `Map` that is defined in the `Order` class:

```
get( Map Order.items )
```

As for method calls and executions, the full class names (in other words, those including package names) as well as the wildcards (*, .., and +) can be used when writing the expressions for `get` and `set`.

Exceptions

The `handler` type corresponds to a joinpoint that occurs when a `catch` block of instructions begins. This type allows you to define aspects that perform compensation treatments when exceptions are thrown.

For example, this type of joinpoint can be used to log the messages that are generated by the thrown exceptions in a running application. Another usage example is defining a common treatment for all the exceptions of a given type. However implemented, the handling of exceptions with an aspect usually lightens the application—making it far more readable and maintainable.

The `handler` type is associated with an exception name. A name can contain the wildcards (*, .., and +). For example, the following expression intercepts the executions of the blocks that catch the `java.io.IOException` exception or one of its subtypes:

```
handler( java.io.Exception+ )
```

With the current version of AspectJ, only "before" advice code can be defined for pointcuts that use the `handler` type. Hence, some code can be executed at the beginning of a `catch` block but not at the end.

Constructors

AspectJ can define pointcuts that include class constructors. To achieve this, two joinpoint types are available: `initialization` and `preinitialization`. The `initialization` joinpoint corresponds to the actual execution of the declared constructor, excluding a possible call to an inherited constructor. The `preinitialization` joinpoint corresponds to the initialization code that is executed before the execution of the constructor, including any default field initializations and any field initializations that have been declared within the class body.

As for the `call` and `execution` types, the `initialization` type takes an expression as a parameter that denotes the constructor or the set of constructors to be intercepted. This expression contains a class name, the `new` keyword, and a signature, and it can contain the usual wildcards (*, .., and +).

For example, the following expression intercepts the executions of all the constructors, regardless of their signatures, that are defined in the `Customer` class:

```
initialization( Customer.new(..) )
```

With AspectJ, "before" and "after" advice code is valid with initialization pointcuts, but "around" advice code is not.

Static Code Blocks

In Java, static code blocks define the instructions that are executed while a class is being initialized—in other words, when the class is loaded in the virtual machine. These blocks are often used to initialize static fields.

Several static code blocks can be associated with a single class. In such a case, the order they are executed in corresponds to the order of their definition. The `staticinitialization` joinpoints correspond to the executions of these static blocks.

With `staticinitialization`, advice code can be executed before and after a static block. For instance, the following expression intercepts the execution of all static blocks that are defined in the `Catalog` class:

```
staticinitialization( Catalog )
```

As for other pointcut descriptors, class names can be associated with package names and wildcards.

Advice-Code Execution

The last existing variety of AspectJ joinpoint, `adviceexecution`, corresponds, as its name suggests, to the execution of advice code. Therefore, you can define an aspect that modifies the execution of another aspect. However, the `adviceexecution` type should be used with caution. If it is used carelessly, there is a high risk of obtaining endless loops during the execution of the application.

All the joinpoint types we have previously presented accept an expression as a parameter; this is not the case for `adviceexecution`. This joinpoint occurs when advice code, including advice code that will be associated with the joinpoint itself, is executed.

Consider the following pointcut descriptor and advice code:

```
pointcut aa(): adviceexecution();
Object before(): aa() { ... }
```

Before each execution of an advice code block, the `aa` pointcut launches the advice code that is given in the example. However, this advice code does not differ from any others—its execution triggers the occurrence of the pointcut, then the execution of the advice code, and so on. To avoid this endless loop, the `adviceexecution` type must be used with the filtering operators that the next section presents.

Filtering Operators

The joinpoints presented previously offer a rich syntax and can be used to define many different pointcut descriptors. You will learn in this section that they can also be combined with logical operators and that filtering operators allow you to restrict the set of caught joinpoints.

Logical Operations

Each defined pointcut descriptor can be compared to a Boolean function. For a given joinpoint, if the pointcut applies, the function returns `true`; if it does not apply, the function returns `false`.

With this logical reasoning in mind, the use of the Boolean operations AND, OR, and NOT is intuitive. They correspond to the conjunction, disjunction, and negation of the occurrence of a joinpoint, respectively. AspectJ supports these three operations with their Java syntax: `&&` for AND, `||` for OR, and `!` for NOT.

The following expression encompasses the executions of the computeAmount method and of the getPrice method:

```
execution( * Order.computeAmount(..) ) || execution( * Catalog.getPrice(..) )
```

The evaluation of a pointcut descriptor is computed for every existing joinpoint. A given joinpoint can be either the execution of computeAmount or the execution of getPrice but never both at the same time. Consequently, the use of && instead of || in the previous pointcut descriptor will not intercept any joinpoint.

The use of && instead of || in a pointcut descriptor is a frequent mistake. However, since the syntax of the expression is correct, the AspectJ compiler does not report an error. Only by using a tool such as the AJDT Eclipse plug-in for AspectJ (see Figure 3-1), which allows you to check the joinpoints that are associated with the pointcut, can you detect that no such points exist and that the pointcut descriptors are erroneous.

In general, combining different joinpoint types in a pointcut descriptor must be done with the || operator.

Pointcut descriptors can include tests. An expression containing if is followed by a Boolean expression and can be combined with any other pointcut expression.

For instance, the expression

```
if( thisJoinPoint.getArgs().length() == 1 )
```

returns true when the current joinpoint defines only one parameter, and it returns false in the other cases.

Filtering

A pointcut descriptor such as get(* aop..*.items) intercepts all the read operations on the items field. For the order-management application, two such joinpoints are included:

- The joinpoint in the addItem method where items is read in order to call the put method.
- The joinpoint in the computeAmount method where items is read in order to call get.

In certain situations, it can be useful to restrict this set to contain only one joinpoint. The withincode keyword, associated with a method name, can be used for this purpose. The method name can contain wildcards (*, .., and +).

The expression withincode(expr) will return true if the name of the method containing the joinpoint matches expr.

The expression

```
get( * aop..*.items ) && !withincode( * aop..*.computeAmount(..) )
```

designates all the read operations for the items field that are not defined in the computeAmount method.

The use of || instead of && in the previous expression does not lead to the expected result. Indeed, !withincode(* aop..*.computeAmount(..)) is true for any joinpoint that is not in the computeAmount method. The evaluation of the expression with the logical OR operator is thus true for these joinpoints, and the pointcut intercepts more joinpoints than expected. In sum, no truly useful situation requires the joint usage of || and withincode.

The second keyword, `within`, exists for filtering joinpoints. This keyword is associated with a class or interface name and can contain wildcards. The `within` keyword allows you to retain joinpoints that are defined in a given class or set of classes.

Two keywords, `this` and `target`, allow you to perform filtering depending on object references. You previously learned that `getThis` and `getTarget` return the current object and the target object of a call or execution joinpoint, respectively. For instance, for the `call` type, `getThis` returns the reference to the caller, and `getTarget` returns the reference to the called object. The `this` and `target` keywords play the same role for pointcut descriptors, and they apply filtering depending on the class of the current object or target object. Each is associated with a class or interface name and can contain wildcards.

The expression

```
call( * aop..*.addItem(..) ) && this(aop.aspectj.Order)
```

is `true` for all the calls made to the `addItem` method by the `Order` class. This expression excludes any calls to `addItem` made by other classes.

Control-Flow Filtering

The filtering operators presented previously are static. They do not depend on the dynamics of the program or the way the program is run but only on its structure. Therefore, all the previous joinpoints can be statically computed without the program needing to be run.

AspectJ defines two additional filtering operators, `cflow` and `cflowbelow`. These are referred to as *control-flow operators*. Intuitively, the control flow of a program encompasses all the methods that are visited during the program's execution.

To illustrate the way that `cflow` and `cflowbelow` work, take the example of a simple program that calls the `Foo.foo` and `Bar.bar` methods from the `main` method. The `Foo.foo` method also calls `Bar.bar`. This last method does not perform any calls.

The following pointcut descriptors intercept the calls to the `bar` method only if `bar` is called from `foo`, so calls to `bar` from `main` are ignored:

```
pointcut foopcd(): call( * Foo.foo(..) );
pointcut callToBarInFoo(): call( * Bar.bar(..) ) && cflow( foopcd() );
```

The `callToBarInFoo` pointcut descriptor specifies that calls to `bar` only in the control flow of the `foopcd` pointcut descriptor are considered. The `foopcd` pointcut descriptor designates all the calls to the `foo` method.

Intuitively, you can consider that the control flow of the program enters the `foopcd` pointcut when `foo` is called and exits the pointcut when the call returns. The expression `cflow(foopcd())` designates all the joinpoints located between this entry point and exit point.

More formally, the `cflow` operator is associated with a pointcut named `p`. All the joinpoints that occur between the moment the program encounters one of the joinpoints included in `p` and the moment the program exits this joinpoint are denoted by `cflow(p())`.

The `cflowbelow` operator is similar to `cflow` except that the joinpoints belonging to `p` are not returned by `cflowbelow`.

Pointcut Parameterization

In the previous sections, you learned about the keywords, operators, and symbols that can be used with AspectJ to define pointcuts. You learned that each pointcut descriptor is named and defined in an aspect, and that the advice code defines the treatment to be executed before and after the joinpoints that are denoted by their associated pointcuts.

In OOP languages, the methods can be parameterized. In AOP, the parameterization also applies to pointcut descriptors. The parameters contain the information that is passed from the pointcut to the utilizing advice code.

The parameters of a pointcut, like those of a method, are defined in parentheses after the name of the pointcut. Each parameter has a name and a type.

The following `toBeTraced` pointcut descriptor defines four parameters (`src`, `dst`, `ref`, and `qty`):

```
pointcut toBeTraced( Customer src, Order dst, String ref, int qty )
```

Pointcut parameters can pass three kinds of information: the source of the joinpoints that are included in the pointcut, the target of those joinpoints, and the parameters of those joinpoints. The goal is to expose the information from the joinpoint so the advice code can access that information.

The source and target of the joinpoints are accessed with a modified version of the `this` and `target` operators, which we previously presented. Instead of being associated with a class name, the operators here use an identifier. The arguments of the joinpoints are accessed with the `args` operator followed by a list of identifiers.

As an example, consider the following `toBeTraced2` pointcut descriptor:

```
pointcut toBeTraced2( Customer src, Order dst, String ref, int qty ) :
 call(* *.*(..)) && this(src) && target(dst) && args(ref,qty);
```

This example denotes all the calls (by the `call(* *.*(..))` subexpression, specifies that the source must be bound to the `src` parameter (by the `this(src)` subexpression), that the target must be bound to `dst` (by the `target(dst)` subexpression), and that the arguments must be bound to the `ref` and `qty` variables (by the `args(ref,qty)` subexpression).

Thanks to the type information that is given in the signature of the pointcut, you can deduce that the pointcut deals with calls to methods that the `Order` class defines, that take parameters of type `String` and `int`, and that are made from the `Customer` class.

You previously saw that the source, target, and arguments could be accessed through the joinpoint-introspection mechanism and the `thisJoinPoint` keyword. As you have seen in this section, pointcut parameterization is an alternative means to achieve this end—but in a typed way. Furthermore, parameterizing a pointcut generally brings more-efficient run-time performances than introspection. The only drawback of parameterization is that the types need to be available at compile time, which is not always the case when programming generic pointcuts.

Summary of Pointcut Descriptors

The AspectJ pointcut language provides a rich syntax and many keywords. Compared to JAC, JBoss AOP, and AspectWerkz, which are presented in the following chapters, AspectJ can define pointcut descriptors that are more precise—although more complex to learn.

To conclude the discussion of joinpoints, we present Table 3-2, which gives a summary of all the existing types of joinpoints in AspectJ. We classify them by their function into the following categories:

- Methods (`call` and `execution`)
- Fields (`get` and `set`)
- Exceptions (`handler`)
- Constructors (`initialization` and `preinitialization`)
- Static code blocks (`staticinitialization`)
- Advice code (`adviceexecution`)

Table 3-2. *AspectJ Joinpoint Types*

Type	Definition
`call(methexpr)`	A call to a method that matches `methexpr`
`execution(methexpr)`	An execution of a method that matches `methexpr`
`get(fieldexpr)`	A read operation on a field that matches `fieldexpr`
`set(fieldexpr)`	A write operation on a field that matches `fieldexpr`
`handler(exceptexpr)`	An execution of a `catch` block for an exception that matches `exceptexpr`
`initialization(constexpr)`	An execution of a constructor that matches `constexpr`
`preinitialization(constexpr)`	An execution of an inherited constructor that matches `constexpr`
`staticinitialization(classexpr)`	An execution of a static block in a class that matches `classexpr`
`adviceexecution()`	An execution of an advice code block

In addition to these joinpoint types, AspectJ pointcut descriptors can also include the operators that are illustrated in Table 3-3. These operators can be grouped into the following categories:

- Logical (`&&`, `||`, `!`, `if`)
- Joinpoint location in the code (`withincode` and `within`)
- Joinpoint source and target (`this` and `target`)
- Control-flow (`cflow` and `cflowbelow`)

Table 3-3. *AspectJ Operators in Pointcut Descriptors*

Keyword	Definition
&&	Logical AND
||	Logical OR
!	Logical NOT
`if(expr)`	Evaluation of the Boolean expression `expr`
`withincode(methexpr)`	`true` when the joinpoint is defined in a method with a signature that matches `methexpr`
`within(typeexpr)`	`true` when the joinpoint is defined in a class with a name that matches `typeexpr`
`this(typeexpr)`	`true` when the source-object type for the joinpoint matches `typeexpr`
`target(typeexpr)`	`true` when the target-object type for the joinpoint matches `typeexpr`
`cflow(pcd)`	`true` for any joinpoint located from the entry of the given pointcut descriptor (`pcd`) to the exit, inclusive
`cflowbelow(pcd)`	`true` for any joinpoint located from the entry of the given pointcut descriptor (`pcd`) to the exit, exclusive

Advice Code

You learned that pointcut descriptors define the areas where the instructions from an aspect need to be inserted. These instructions are defined in advice code blocks.

In object-oriented languages, the behavior of a class is defined in its methods. In contrast, in AOP, this behavior is defined in advice code. In an aspect, there may be several advice code blocks, with each one containing a set of instructions. In addition, each advice code block has a type and is associated with a pointcut descriptor.

The Code of an Advice Code Block

The code of an advice code block can contain any valid Java instruction, such as a method call, a variable assignment, an object creation (using `new`), a loop (using `for`, `while`, or `do/while`), a test (using `if`), and an exception-handling block (using `try/catch`).

Two additional instructions, `proceed` and `thisJoinPoint`, are also valid in AspectJ. These instructions are used exclusively for advice code; in any other context, they generate compiler errors. The `proceed` keyword executes the joinpoint and can be used only in "around" advice code. The `thisJoinPoint` keyword can be used for any kind of advice code. As mentioned previously, `thisJoinPoint` is a reference to an object that describes the current joinpoint.

The Different Types of Advice Code

AspectJ defines five types of advice code. Of these five, "before," "after," and "around" are the more-commonly encountered ones. The last two types, "after returning" and "after throwing," can be seen as refinements of the "after" type.

The "Before" Type

"Before" advice code is executed before the joinpoints that are included in the pointcut associated with the advice code.

The following example illustrates the usage of "before" advice code:

```
before(): toBeTraced() {
  System.out.println("... before the joinpoints included in toBeTraced ...");
}
```

The syntax for the definition of "before" advice code consists of the type of advice code (here, `before`), the name of the pointcut descriptor that is associated with the advice code (here, `toBeTraced`), and the code itself between curly brackets.

The second part, the name of the pointcut descriptor, is not mandatory. You can provide the code of the pointcut descriptor right after the type. In this case, the pointcut is said to be *anonymous*. An example of anonymous pointcut is the following:

```
before(): call( * Order.addItem(..) ) { ... }
```

However, the use of a named pointcut descriptor is preferred because this produces programs that are clearer and easier to maintain. Furthermore, when a pointcut descriptor is reused in several advice code blocks, the use of a name avoids useless, error-prone repetitions.

In the previous section, you learned that pointcut descriptors can accept parameters. Hence, when used, the associated advice code must also be parameterized.

The example in Listing 3-10 reuses the `toBeTraced2` pointcut descriptor with four parameters and associates the pointcut with "before" advice.

Listing 3-10. *Defining Parameterized Pointcuts*

```
 1 pointcut
 2   toBeTraced2( Customer src, Order dst, String ref, int qty ):
 3     call( * *.*(..) ) &&
 4     this(src) && target(dst) && args(ref,qty);
 5 before( Customer src, Order dst, String ref, int qty ):
 6   toBeTraced2(src,dst,ref,qty) {
 7     System.out.println(
 8       "... before the joinpoints included in toBeTraced2 ...");
 9     System.out.println( src + " " + dst + " " + ref + " " + qty );
10 }
```

The four parameters that are defined in the pointcut descriptor are also used for the definition of the advice code. (See line 6 in Listing 3-10). These parameters are then available for any instruction that is defined in the body of the advice code block.

The "After" Type

"After" advice is executed after each associated joinpoint. Its code uses the same syntax rules as "before" advice. The previous examples of the `toBeTraced` and `toBeTraced2` pointcut descriptors would be valid simply by replacing the `before` keyword with `after`.

The last two types of advice code that are defined by AspectJ are "after returning" and "after throwing." These types ensue from the idea that the execution of a joinpoint—for instance, a method-execution joinpoint—ends normally or with the raising of an exception. The former case is handled by the "after returning" type, whereas the latter case is handled by the "after throwing" type.

The "After Returning" Type

"After returning" advice code is executed after each normal execution of the joinpoints that are associated with the pointcut descriptor.

The following code illustrates the usage of the "after returning" type:

```
after() returning (double d): ... {
  System.out.println("The returned value is: "+d);
}
```

The value that is returned by the joinpoint can be accessed with the variable found in parentheses after the `after returning` keywords. The variable is either the exact type (here, `double`) of the value returned by the joinpoint or a valid supertype (for example, `Object` for all Java types, including primitive types).

The "After Throwing" Type

"After throwing" advice code is executed when a given joinpoint that is associated with a pointcut descriptor ends its execution by raising an exception.

The following code illustrates the usage of the "after throwing" type:

```
after() throwing (Exception e): ... {
 System.out.println("The raised exception is: "+e);
}
```

The exception that is raised by the joinpoint can be accessed with the variable defined in parentheses following the `after throwing` keywords. In the previous example, the variable is `e`. This variable can be used anywhere in the body of the advice code.

The "Around" Type

"Around" advice code is executed before and after each associated joinpoint. The `proceed` keyword executes the joinpoint, which is bound by the "before" and "after" parts of the advice code. Specifically, this keyword executes the joinpoint in the following way:

1. The "before" part is executed.
2. The joinpoint is executed when the `proceed` keyword is used.
3. The "after" part is executed.

In "around" advice code, the "before" or "after" part can be empty; then, the "around" advice code is equivalent to "before" or "after" advice code. The `proceed` keyword is optional in "around" advice code. If `proceed` is not used, the joinpoint is not executed. This behavior could

correspond to a security aspect in which calls from unauthorized users are rejected, for instance. The `proceed` keyword can be used several times in the same "around" advice code block. However, this situation is infrequent and corresponds to cases in which several attempts at executing the application are needed—for instance, after an unexpected error.

Unlike "before" and "after" advice code, the return type of "around" advice code is associated with the return type of the joinpoints. If the "around" advice code is not a supertype of the return type that is defined for the joinpoints, the AspectJ compiler raises an error. When other return types (including the `void` return type) appear for the joinpoints in a given pointcut, the `Object` type must be used as the return type of the advice code. (In Java, `Object` is considered the supertype of all types.)

Listing 3-11 illustrates the use of "around" advice code.

Listing 3-11. *"Around" Advice-Code Example*

```
Object around(): ... {
 System.out.println("before");
 Object ret = proceed();
 System.out.println("before");
 return ret;
}
```

In Listing 3-11, the call to `proceed` returns a value that is stored in the `ret` variable. This is the value that is returned by the joinpoint. This value and, in fact, all other values, must be returned by the advice code (as is done here by the `return ret` instruction).

When the advice code is associated with a parameterized pointcut descriptor, all the parameters must be passed when `proceed` is called. This is illustrated by Listing 3-12.

Listing 3-12. *"Around" Advice Code with Parameters*

```
Object around( Customer src, Order dst, String ref, int qty ):
 toBeTraced2(src,dst,ref,qty) {
  System.out.println("before");
  Object ret = proceed(src,dst,ref,qty);
  System.out.println("after");
  return ret;
}
```

Advice Code and Exceptions

Advice code has the ability to raise an exception when needed. In such cases, the type of the exception must be specified in the signature of the advice code. For methods, the `throws` keyword must be used to specify the exception.

The following piece of code defines "around" advice code that possibly throws an exception:

```
  Object around() throws Exception: ... {
  /* ... */
  if( /*condition*/ )
    throw new Exception();
  /* ... */
}
```

When an exception is declared by advice code, the type of the exception must also be specified in the signature of the joinpoint. For instance, a method-execution joinpoint must list the exception in the `throws` clause. This is a limitation that obliges the application code to be aware of the exceptions thrown by the aspects. In practice, this limitation can be solved by specifying that the advice code raises an exception of type `RuntimeException`. In Java, run-time exceptions are unchecked; therefore, the signatures of the program methods can be left unchanged.

The Introduction Mechanism

In the previous sections, you learned that pointcut descriptors and advice code blocks allow an aspect to extend or modify the behavior of an application. Pointcut descriptors designate joinpoints (method calls, method executions, read operations, write operations, and so on) in the execution flow of a program, and advice code blocks add instructions before or after these joinpoints. In all cases, if no joinpoints are activated, the advice code will not be executed either.

The introduction mechanism is used in AspectJ to extend the structure of an application. The term *introduction* refers to the process of the aspect adding code elements to the application. AspectJ can introduce, or add, six categories of these elements: fields, methods, constructors, inherited classes, implemented interfaces, and exceptions. The following sections will explain these categories in detail.

Contrary to advice code, which extends the behavior of an application only when the joinpoints are executed, the introduction mechanism is unconditional—the extended code is always added.

AspectJ uses the term *intertype declaration* to refer to the introduction mechanism. The concept behind this is that an aspect, which is considered a type, declares elements (for example, fields, methods, inherited classes, and implemented interfaces) on behalf of other types—in other words, on behalf of the classes of the application.

No special keyword is defined in AspectJ for introduced elements, which are defined declaratively.

Fields, Methods, and Constructors

An aspect that introduces fields, methods, or constructors in a class performs a declaration on behalf of this class. This declaration follows the same rules as a regular Java declaration. The name of the field or method is preceded by the name of the class that the introduction is to be performed in. Constructors are designated with the `new` keyword.

The aspect in Listing 3-13 introduces the `date` field, two methods named `getDate` and `setDate`, and a constructor in the `Order` class.

Listing 3-13. *Intertype Declaration*

```
import java.util.Date;

public aspect AddDate {

  private Date Order.date;
  public Date Order.getDate() { return date; }
  public void Order.setDate(Date date) { this.date=date; }
```

```
  public Order.new(Date date) { this.date=date; }

  after(): initialization(Order.new(..)) {
    Order myOrder = (Order) thisJoinPoint.getTarget();
    myOrder.date = new Date();
  }
}
```

In the `AddDate` aspect in Listing 3-13, "after" advice code is defined after the instantiations of the `Order` class. This advice code collects the references of the instantiated orders and sets the current date to the introduced `date` field.

Although these introductions are simplistic, they should be dealt with cautiously. Accordingly, the elements defined in the application must not conflict with the elements introduced by the aspect. For instance, if a `date` field already exists in the `Order` class, a compiler error occurs.

AspectJ does not provide an option to check whether a field or method name already exists before it performs the introduction. The existence of conflicting elements is revealed only during compile time when the error occurs. Therefore, it is up to you to manually correct the program or the aspect when this occurs.

Inherited Classes and Implemented Interfaces

In addition to fields and methods, the introduction mechanism provided by AspectJ allows you to modify the inheritance and implementation hierarchies defined in an application. The `declare parents` keyword combination permits this.

The following aspect adds the `AddDateItf` interface to the `Order` class and creates an inheritance link between `AddDateImpl` and `Order`:

```
public aspect AddDate2 {
  declare parents: Order implements AddDateItf;
  declare parents: Order extends AddDateImpl;
}
```

The addition of a new interface is always possible since there are no constraints in Java on the number of interfaces implemented by a class. This is not the case for inheritance because only a single inheritance is supported in Java.

In the previous example, if the `Order` class was already a subclass of the `Document` class, the modification of the inheritance link by the `AddDate2` aspect would have produced a compiler error.

The names of the classes to be modified can contain wildcards. Logical expressions created with the Boolean OR operator (`||`) can be written to modify several classes with a single declaration.

Exceptions

The last element that can be introduced performs the function of catching the exceptions that are raised by an application.

The exceptions raised by the application are wrapped in a special exception called `org.aspectj.lang.SoftException`.

The `declare soft` keyword combination is provided to implement this introduction mechanism. A type and a pointcut descriptor must be provided. The type designates the type of exceptions to be caught, and the pointcut descriptor designates the joinpoints where the exceptions are to be caught.

For instance, the following line of code

```
declare soft: IOException+: call( * InputStream.*(..) )
```

declares that the subclasses of a certain exception—the `IOException` exception that is thrown when a method defined in the `InputStream` class is called—must be wrapped in the `org.aspectj.lang.SoftException` exception.

Advanced Features

The previous sections introduced the many features offered by AspectJ for programming aspects. As with other languages, advanced functionalities—apart from the mainstream features—that complement the language exist. In AspectJ, these are the concepts of the abstract aspect, aspect inheritance, aspect instantiation, aspect ordering, and the privileged aspect.

The Abstract Aspect

The aim of the *abstract aspect* in AspectJ is to define an aspect that has some undefined elements (pointcuts or methods). These elements are then said to be *abstract*. This concept is similar to that of an abstract class. The precise definition of these elements will be given in a subaspect.

Abstract aspects allow you to factor definitions that are shared by several aspects. As with abstract classes, abstract aspects cannot be instantiated.

The `abstract` keyword is used to define abstract aspects. It can be written before the `aspect` or `pointcut` keyword (for defining pointcut descriptors) and before the method that is abstract in the aspect. The following section illustrates the precise definition of an abstract aspect.

Aspect Inheritance

Inheritance can be used with aspects like it is used with classes. The goal is to extend an aspect without rewriting it completely. Only single inheritance is supported.

As with classes, the `extends` keyword provides the inheritance feature. In the following line of code, the `TraceAspect2` aspect extends the `TraceAspect` aspect:

```
public aspect TraceApect2 extends TraceAspect { ... }
```

However, aspect inheritance does not follow the same exact rules as class inheritance. An aspect can extend only abstract aspects, whereas a class can extend both abstract and nonabstract classes.

A pointcut can be redefined in a subaspect. If a pointcut is redefined, the redefined version will always be used. Pointcut redefinition with aspect inheritance follows the same rules as method redefinition with class inheritance.

Conversely, advice code cannot be redefined. Inherited advice code is thus always available in a subaspect.

The code in Listing 3-14 defines the abstract `TraceAspect` aspect with the abstract `toBeTraced` pointcut descriptor. `TraceAspect2` extends `TraceAspect` and defines the `toBeTraced` pointcut descriptor.

Listing 3-14. *The Definition of an Abstract Aspect*

```
public abstract aspect TraceAspect {
  abstract pointcut toBeTraced();
  before(): toBeTraced() { ... }
}

public aspect TraceAspect2 extends TraceAspect {
  pointcut toBeTraced(): call(* Order.*(..));
}
```

Aspect Instantiation

By default, a unique instance of an aspect is created when the application is launched. The aspect is then said to be a *singleton*. The same aspect instance is shared by all the application objects.

In special cases, it can be useful to create several instances of a given aspect. Different application objects are then aspectized by the different instances. For example, this feature can be used to dedicate different pieces of data to each part of the application.

The three following cases can occur:

- The aspect is a singleton, and only one instance of the aspect exists at run time. This is the default case.
- The aspect is instantiated several times, and the instances are associated with different application objects.
- The aspect is instantiated several times, and the instances are associated with different control-flow sequences of the application.

No keyword exists for the first case because this is the default behavior when the `aspect` keyword is used.

For the second case, two keywords are available: `perthis` and `pertarget`. The aspects can then be written as follows:

```
aspect <name> perthis( <poincut> ) { ... }
aspect <name> pertarget( <poincut> ) { ... }
```

When `perthis` is used, an instance of the aspect is created for every object that is an executing object of the given pointcut. The other objects (those that are not executing objects of the pointcut) are not aspectized. When `pertarget` is used, an instance of the aspect is created for every object that is the target object of the given pointcut.

For the third case, two keywords are available: `percflow` and `percflowbelow`. The aspects can then be written as follows:

```
aspect <name> percflow( <poincut> ) { ... }
aspect <name> perclfowbelow( <poincut> ) { ... }
```

When `percflow` is used, an aspect instance is created each time the application enters the control-flow sequence that is designated by the pointcut. As for the `cflow` operator in pointcut definitions, the joinpoints belongs to the control flow. When `percflowbelow` is used, an aspect instance is created each time the application enters the control-flow sequence that is designated by the pointcut, but the joinpoints are not included in the control flow.

The static `aspectOf` method is defined for each aspect and returns the aspect instance that is currently in use. For example, in the singleton `TraceAspect` aspect, the call `TraceAspect.aspectOf()` returns the reference to the singleton. For the `perthis` and `pertarget` types, a parameter must be passed when `aspectOf` is called; this parameter gives the source or the target object that is associated with the requested instance. For the `percflow` and `percflowbelow` types, no parameters are needed. The method returns the aspect instance or `null`, depending on whether the current run is included in the control-flow sequences that are defined for the aspects.

Aspect Ordering

When two or more aspects apply to the same joinpoint, the execution order of these aspects must be determined.

AspectJ allows you to define explicit ordering rules. This is called *explicit ordering*. If the rules are undefined, the compiler automatically orders the aspects. This is called *implicit ordering*.

Explicit Ordering

The `declare precedence` keyword combination allows you to declare the execution order of different aspects.

The following code illustrates the usage of this keyword combination:

```
aspect GlobalOrder {
  declare precedence: Authentication, Trace;
}
aspect Authentication { ... }
aspect Trace { ... }
```

In the previous code, the `Authentication` aspect is always executed before `Trace` is. Wildcards can be used in the aspect names that are associated with `declare precedence`.

Despite the location of the definition, the order is valid for the whole program. The `declare precedence` keyword combination can be used several times in a program. If the given orders are inconsistent, the AspectJ compiler raises an error.

Implicit Ordering

When no order is defined, or when the order is defined for some aspects but not for others, AspectJ applies the following rules:

- The subaspects are applied before the inherited aspects.
- No order is guaranteed for aspects that are not linked by an inheritance relationship.
- If several advice code blocks apply to the same joinpoint for a given aspect, the following rules apply:
  - "After" advice code blocks are executed last.
  - Advice code blocks are executed in the order in which they are defined in the aspect.

These rules can lead to inconsistencies which are raised by the AspectJ compiler.

It is recommended that the aspect order be defined explicitly and as often as possible with the `declare precedence` keyword combination. When you write an aspect, it is favorable to first write all the "before" advice code and then the "after" advice code.

The Privileged Aspect

For accessing fields or methods, the same rules apply to aspects as Java classes. For example, aspects cannot read or write a private or protected field. The purpose of this particular rule is to guarantee the integrity of the program and to avoid the accidental and erroneous altering of objects.

Nevertheless, some cases require a bypass of this limitation. AspectJ provides for these cases the concept of a privileged aspect:

```
privileged aspect <name> { ... }
```

A privileged aspect can access all the fields and methods defined in a class—regardless of their access modifiers. This feature must be used cautiously as it may corrupt the normal behavior of the program.

Declaring Warnings and Errors

AspectJ offers a mechanism that raises compile-time warnings or errors whenever a given pointcut expression is matched by a program. In this way, you can be notified if your program defines unwanted code elements.

For example, the aspect in Listing 3-15 raises a warning if the `Remote` interface is implemented in the `bank.ejb` package.

Listing 3-15. *Declaring Warnings with an Aspect*

```
public aspect Foo {
  declare warning:
    execution(* Remote+.*(..)) && within(bank.ejb.*):
      " Remote may interfere with EJBs in bank.ejb";
}
```

A message can be associated with each raised warning.

The `declare error` keyword combination works similarly.

Load-Time Weaving

By default, AspectJ is a compile-time weaver. Given a set of `.java` source files and a set of aspects, the `ajc` command-line tool produces a set of `.class` files in which the aspects are woven to the classes.

Since version 1.2, AspectJ can weave code at load time. The source code of the program is no longer required, and AspectJ can weave any class that can be obtained with the class-loading mechanism of the Java language.

New Features in AspectJ 5

During the writing of this book, AspectJ 1.2.1, which was released on November 5, 2004, was the latest stable version of AspectJ. However, a newer version, numbered 1.5.0 and officially called AspectJ 5, is under preparation. The first major developments of AspectJ 1.5.0M1 were made available to the developing community in December 2004. The main purpose of this evolution was to incorporate the changes brought to the Java language by Java 5.

The features described in the remainder of this section can be found in the developer release of AspectJ 1.5.0, which is available at the time of the writing of this book. By the time AspectJ 5 is final, these features may have slightly changed according to user's feedback, design choices, or error corrections.

Most of the changes brought by AspectJ 5 deal with *annotations* (also known as *metadata*). First, an aspect can deal with an annotated Java program. The annotations defined for classes or methods can be taken into account when defining a pointcut. Also, annotations can be introduced into a Java program. Second, annotations have an impact on the syntax of the AspectJ language itself. Instead of using dedicated keywords such as `aspect` and `pointcut`, you can write annotated Java classes that will be understood by the AspectJ weaver as aspects. These two categories of features are presented in the remainder of this section.

Working with Annotations in Aspects

The handling of annotations in Java programs brings several changes to the AspectJ language. The changes concern the definition of an aspect, the pointcut-definition language, and the introduction of annotations.

Annotations and Pointcut Definitions

You learned in the previous sections that a pointcut can capture elements of a Java program, such as methods and fields, and link them together in order to aspectize them. In this process, the writing of a pointcut relies heavily on the elements that define a field or method, such as the name, type, or signature. It then seems natural to incorporate annotations for fields or methods in the writing of a pointcut.

With AspectJ 5, the pattern language for a pointcut descriptor is extended with the at-sign symbol (`@`) followed by an annotation name. For instance, the following expression

```
execution( @Transaction * *.*(..) )
```

designates all the joinpoints that are executions of a method annotated with `@Transaction`, regardless of the method's return type, declaring class, name, or signature.

Several annotations can be specified, in which case all of them are required for the element to be included in the pointcut.

The Boolean OR operator (`||`) can be used between annotations to signify that at least one of the annotations must be present. For instance, `@(Foo || Bar)` means that either `@Foo` or `@Bar` is required as an annotation.

The Boolean NOT operator (`!`) is used in front of an annotation. For instance, `!@Foo` means that all elements that are not annotated with `@Foo` are included in the definition.

Joinpoints can be filtered according to the annotations available to the source and target objects of a method call. The `@this` and `@target` operators are defined for this purpose. For instance, the following expression

```
call( * *.*(..) ) && @target( @Transaction )
```

designates all the calls to a method that is annotated with `@Transaction`.

At the time this book was written, it was not possible to match joinpoints based on their annotation values. This facility may be supported in future releases of AspectJ.

Introducing Annotations

In AspectJ 5, the `declare` keyword is associated with four new forms that allow you to introduce annotations in Java programs.

Introducing annotations consists of defining a pattern that refers to program elements (classes, interfaces, fields, methods, or constructors) and giving the annotations that must be introduced for all the program elements that match the pattern.

For instance, the following instruction

```
declare @type: bank.* : @EJBean;
```

introduces the annotation `@EJBean` for all the types (classes and interfaces) that are defined in the `bank` package.

Similarly, annotations can be introduced for methods. The keyword combination is then `declare @method` and the pattern refers to method names. For example, the following instruction

```
declare @method: bank.*.deposit(..): @Transaction(value="required");
```

introduces the annotation `@Transaction(value="required")` for all the `deposit` methods that are defined in the `bank` package.

The `declare @field` and `declare @constructor` keywords follow the same principle.

Defining Aspects with Annotations

Using the previously presented features, you can write aspects that deal with a Java program containing annotations, or you can write aspects that introduce annotations.

The annotations that are presented in this section are very different. Their purpose is to replace the current syntax of the AspectJ language and to allow you to write annotated Java classes. These classes are understood by the AspectJ weaver not as regular Java classes but as aspects. This approach creates an entirely new development style in which there are no more syntax extensions, but instead there are a set of annotations available for AOP.

The first annotation is `@Aspect`. A Java class annotated with `@Aspect` is understood as an aspect. The following code snippet defines the `TraceAspect4` aspect:

```
@Aspect
public class TraceAspect4 {
  // ...
}
```

You learned in the "Advanced Features" section earlier in this chapter that aspects can be qualified, either with an instantiation clause (using `perthis`, `pertarget`, `percflow`, or `percflowbelow`) or with the `privileged` keyword. These qualifiers still exist with the `@Aspect` annotation style. For example, the following annotation

```
@Aspect(
  instantiationModel=AspectInstantiationModel.PERTARGET,
  perClausePattern="aPointcutDefinition()",
  isPrivileged=true )
```

defines a privileged aspect with a `PERTARGET` instantiation model.

Pointcut Definition

With the annotation-based development style, a pointcut is a method with an empty body that is annotated with `@Pointcut`. This annotation takes the pointcut expression as a parameter. For example, the following code snippet defines the pointcut `toBeTraced`:

```
@Pointcut( "call(* aop.aspectj.Order.*(..))" )
void toBeTraced() {}
```

With this new style, the parameters that would have been defined for a pointcut are now defined as parameters of the annotated method.

Advice-Code Definition

Advice code blocks are already blocks of instructions, so it is quite natural that advice code blocks are now methods. Five new annotations are available, one for each different type of advice code: `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, and `@Around`.

In all five cases, the annotation takes the associated pointcut expression as a parameter.

"Before" Advice Code

The following code snippet illustrates the definition of "before" advice:

```
@Before( "toBeTraced()" )
public void beforeTrace() { ... }
```

Note that with this new development style, advice code blocks now share the same name as the method.

If the "before" advice code needs to access the joinpoint, a parameter of type `JoinPoint` is added to the method signature, as shown here:

```
@Before( "toBeTraced()" )
public void beforeTrace( JoinPoint jp ) { ... }
```

If the pointcut expression defines parameters, these parameters will also be made available as method parameters. The following code lines, for example, define "before" advice code for method calls in which the `src` and `dst` parameters are bound to the source and target, respectively, of the call:

```
@Before( "toBeTraced() && this(src) && target(dst)" )
public void beforeTrace( Object src, Object dst ) { ... }
```

This principle of appending the joinpoint and the pointcut parameters to the signature of the advice-code method also applies to other types of advice code.

"Around" Advice Code

"Around" advice code is similar to "before" advice code. The only issue here is raised by the annotation development style, which is linked to `proceed`. If you do not change the old syntax, the call to `proceed` will lead to a compiler error. (Keep in mind that aspects are now regular Java classes.)

To solve this problem, AspectJ 5 defines `proceed` as a method of the `ProceedingJoinPoint` interface, which extends the existing `JoinPoint` interface and allows the joinpoint to be visible as a parameter of the "around" advice-code method.

The following lines of code illustrate the definition of "around" advice code:

```
@Around( "toBeTraced()" )
public Object trace( ProceedingJoinPoint jp ) {
  // ... Before code
  Object ret = jp.proceed();
  // ... After code
  return ret;
}
```

"After" Advice Code

The definition of "after" advice code does not differ from the definition of "before" advice code except that the annotation is `@After`.

The situation is slightly different for `@AfterReturning` and `@AfterThrowing`. The returned value or the thrown exception must be made visible in the advice-code method.

In the case of `@AfterReturning`, a new parameter named `returning` is added to the annotation. Its value identifies the method parameter that will contain the returned value. The following lines define "after returning" advice code:

```
@AfterReturning( value="toBeTraced()", returning="ret" )
public void afterTrace( Object ret ) { ... }
```

The value returned by the intercepted joinpoints is assigned to the `ret` parameter.

The principle is the same for "after throwing" advice code, except that the annotation defines a parameter named `throwing` for holding the thrown exception.

Declare Statements

The `declare` statement is available in various forms in AspectJ:

- `declare parents ... implements` and `declare parents ... extends`: These are the two most current forms, which allow you to introduce a new interface and a new superclass, respectively.
- `declare error` and `declare warning`: These forms raise errors and warnings.
- `declare @...`: Annotations can be introduced with this form.
- `declare precedence`: Aspect ordering can be defined with this form.
- `declare soft`: This form can be used to soften exceptions.

At the time of the writing this book, all the previous forms of `declare` statements, except `declare parents ... extends` and `declare soft`, were available with the new annotation-based development style. The two exceptions may be added in a future release.

The aspect in Listing 3-16 illustrates the definition of the following:

- A precedence rule between all the aspects that have a name starting with `Authentication` and the `Trace` aspect
- The declaration to raise a warning whenever the `Remote` interface is implemented in the `bank.ejb` package
- The introduction of the annotation `@EJBean` for the types defined in the `bank.ejb` package
- The introduction of the annotation `@Transaction(value="required")` for the `deposit` methods defined in the `bank.ejb` package

Listing 3-16. *AnnotationBased Declare Statements*

```
@Aspect
@DeclarePrecedence( "Authentication*,Trace" )
public class GlobalOrder {

  @DeclareWarning( "execution(* Remote+.*(..)) && within(bank.ejb.*)" )
  final static String message = "Remote may interfere with EJBs in bank.ejb";

  @DeclareAnnotation( "bank.*" )
  @EJBean
  Object beans;

  @DeclareAnnotation( "bank.*.deposit(..)" )
  @Transaction(value="required")
  void depositMethods() {}
}
```

Aspect Instantiation in AspectJ 5

AspectJ 5 supports the five aspect-instantiation modes—the default singleton mode, `perthis`, `pertarget`, `percflow`, and `percflowbelow`—that are available in the previous version. (See the "Aspect Instantiation" section earlier in this chapter for further details.)

AspectJ 5 introduces a sixth mode: `PERTYPEWITHIN`. With this mode, a new aspect instance is created for each new type that is designated in the pointcut expression associated with `PERTYPEWITHIN`.

The following aspect illustrates the `PERTYPEWITHIN` mode:

```
@Aspect(
  instantiationModel=AspectInstantiationModel.PERTYPEWITHIN,
  perClausePattern="aop.aspectj.*")
public class FooBarAspect { ... }
```

This aspect will be instantiated for each different class that is defined in the `aop.aspectj` package.

Other Java 5 Features

Autoboxing does not impose any changes on the AspectJ language. The fact that primitive types (`int`, `float`, `double`, and so on) are equivalent to their class-based ones (`Integer`, `Float`, `Double`, and so on) simply means that when the AspectJ compiler computes the joinpoints that match a given pointcut, the compiler does not distinguish between a primitive type and its class-based counterpart.

Variable-length argument lists can be used when you write a pointcut descriptor. For instance, the following pointcut descriptor

```
call( * *.*( double, Object... ) )
```

designates all the calls to all the methods that take at least a `double` argument and a variable-length list of `Object` arguments as parameters.

Summary

This chapter presented the syntax of the AspectJ language. AspectJ extends the Java language with keywords for writing aspects, pointcuts, advice code, and intertype declarations. An aspect-oriented application in AspectJ contains Java classes that implement the core logic of the application and AspectJ aspects that implement the crosscutting functionalities.

The classes and the aspects are woven together to produce the final application. Most of the time, the weaving occurs at compile time. AspectJ provides a compiler (named `ajc`) for that. With newer versions of AspectJ, the weaving can also be done at load time—when the application is loaded into the Java virtual machine.

The pointcut language provided by AspectJ allows you to define where an aspect applies in an application. A pointcut descriptor denotes a set of joinpoints. Several types of joinpoints are supported by AspectJ. The two that are most frequently used are the `call` and `execution` joinpoints. These are the points in the execution flow of a program where a method is called or executed, respectively. Several other types of joinpoints exist in AspectJ: `get` and `set` for read and write operations, `handler` for exceptions, `initialization` and `preinitialization` for

constructors, `staticinitialization` for static code blocks, and `adviceexecution` for executions of advice code.

Advice code defines the modifications that are brought to an application by an aspect. Each advice code block is associated with a pointcut that defines where these modifications apply. Three main types of advice code are provided by AspectJ: "before," "after," and "around." Advice code applies either before the executions of its associated joinpoints, after the executions, or both. In the latter case, the advice code is called "around" advice code. Two other types of advice code are provided by AspectJ: "after returning," and "after throwing." The former applies for to that return normally, whereas the latter applies to joinpoints that end their executions by raising an exception. The behavior of an aspect is provided by the advice code blocks that are attached to the aspect. Any Java instructions are valid in those blocks. AspectJ provides two additional instructions: `thisJoinPoint` and `proceed`. The former provides information (method name, class name, parameters, and so on) about the current joinpoint. The latter, which can be used exclusively with "around" advice code, allows you to execute the joinpoint.

The mechanism known by the term *intertype declaration* provides a way for AspectJ to extend an application with additional features, such as fields, methods, constructors, interfaces, and superclasses.

Finally, AspectJ 5 takes advantage of the many new features of the Java 5 language, and it introduces a whole new development style that is based on annotations. Instead of using keywords such as `aspect` and `pointcut`, you can write aspects as annotated Java classes.