

# AspectJ

---

# IN ACTION

Practical Aspect-Oriented Programming

Ramnivas Laddad



 MANNING

For online information and ordering of this and other Manning books, go to [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department  
Manning Publications Co.

209 Bruce Park Avenue  
Greenwich, CT 06830

Fax: (203) 661-9018

email: [orders@manning.com](mailto:orders@manning.com)

©2003 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.



Manning Publications Co.  
209 Bruce Park Avenue  
Greenwich, CT 06830

Copyeditor: Liz Welch  
Typesetter: Denis Dalinnik  
Cover designer: Leslie Haines

ISBN 1-930110-93-6

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – VHG – 05 04 03 02

# *brief contents*

---

## **PART 1 UNDERSTANDING AOP AND ASPECTJ..... 1**

- 1 ■ Introduction to AOP 3
- 2 ■ Introducing AspectJ 32
- 3 ■ AspectJ: syntax basics 64
- 4 ■ Advanced AspectJ 100

## **PART 2 BASIC APPLICATIONS OF ASPECTJ.....143**

- 5 ■ Monitoring techniques: logging, tracing, and profiling 145
- 6 ■ Policy enforcement: system wide contracts 178
- 7 ■ Optimization: pooling and caching 202

## **PART 3 ADVANCED APPLICATIONS OF ASPECTJ.....243**

- 8 ■ Design patterns and idioms 245
- 9 ■ Implementing thread safety 286
- 10 ■ Authentication and authorization 323

- 11 ■ Transaction management 356
- 12 ■ Implementing business rules 391
- 13 ■ The next step 425
- A ■ The AspectJ compiler 438
- B ■ Understanding Ant integration 447
  - resources 455
  - index 461

# *AspectJ: syntax basics*

---

## ***This chapter covers***

- Pointcuts and advice
- Static crosscutting
- Simple examples that put it all together

In chapter 2, we presented a high-level view of the AspectJ programming language and introduced the concepts of aspects and join points. In this chapter, we continue with a more detailed discussion of the constructs of pointcuts and advice, their syntax, and their usages. We also examine a few simple programs that will help strengthen your understanding of the AspectJ constructs. Then we discuss static crosscutting. After reading this chapter, you should be able to start writing short programs in AspectJ.

Although the AspectJ syntax may feel somewhat complex in the beginning, once you understand the basic form, it's quite natural for a seasoned Java programmer: An aspect looks like a class, a pointcut looks like a method declaration, and an advice looks like a method implementation. Rest assured that the AspectJ syntax is actually a lot easier than it appears.

### 3.1 Pointcuts

---

Pointcuts capture, or identify, join points in the program flow. Once you capture the join points, you can specify weaving rules involving those join points—such as taking a certain action before or after the execution of the join points. In addition to matching join points, certain pointcuts can expose the context at the matched join point; the actions can then use that context to implement crosscutting functionality.

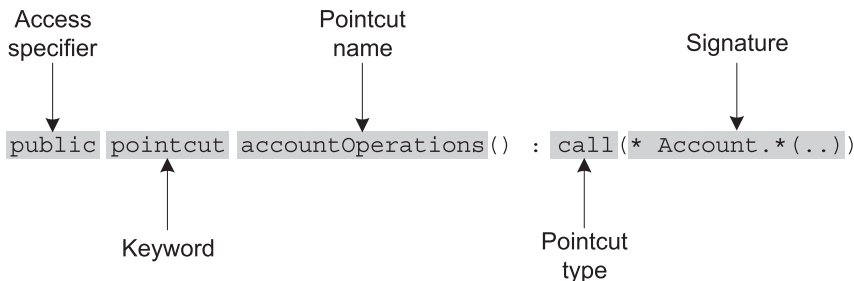
A pointcut designator identifies the pointcut either by name or by an expression. The terms *pointcut* and *pointcut designator* are often used interchangeably. You can declare a pointcut inside an aspect, a class, or an interface. As with data and methods, you can use an access specifier (public, private, and so forth) to restrict access to it.

In AspectJ, pointcuts can be either *anonymous* or *named*. Anonymous pointcuts, like anonymous classes, are defined at the place of their usage, such as a part of advice, or at the time of the definition of another pointcut. Named pointcuts are elements that can be referenced from multiple places, making them reusable.

Named pointcuts use the following syntax:

```
[access specifier] pointcut pointcut-name([args]) : pointcut-definition
```

Notice that the name of the pointcut is at the left of the colon and the pointcut definition is at the right. The pointcut definition is the syntax that identifies the join points where you want to insert some action. You can then specify what that action is in advice, and tie the action to the pointcut there. (We discuss advice in section 3.2.) Pointcuts are also used in static crosscutting to declare compile-time



**Figure 3.1** Defining a named pointcut. A named pointcut is defined using the pointcut keyword and has a name. The part after the colon defines the captured join points using the pointcut type and signature.

errors and warnings (discussed in section 3.3.3) as well as to soften exceptions thrown by captured join points (see section 4.4).

Let's look at an example of a pointcut named `accountOperations()` in figure 3.1 that will capture calls to all the methods in an `Account` class.

You can then use the named pointcut in advice as follows:

```
before() : accountOperations() {
    ... advice body
}
```

An anonymous pointcut, on the other hand, is a pointcut expression that is defined at the point of its usage. Since an anonymous pointcut cannot be referenced from any place other than where it is defined, you cannot reuse such a pointcut. Consequently, in practice, you should avoid using anonymous pointcuts when the pointcut code is complicated. Anonymous pointcuts can be specified as a part of advice, as follows:

```
advice-specification : pointcut-definition
```

For example, the previous example of a named pointcut and advice could all be replaced just by advice that includes an anonymous pointcut, like this:

```
before() : call(* Account.*(..)) {
    ... advice body
}
```

You can also use an anonymous pointcut as part of another pointcut. For example, the following pointcut uses an anonymous `within()` pointcut to limit the join points captured by calls to `accountOperations()` that are made from classes with `banking` as the root package:

```
pointcut internalAccountOperations()
    : accountOperations() && within(banking..*);
```

Anonymous pointcuts may be used in a similar manner as a part of static crosscutting.

Regardless of whether a pointcut is named or anonymous, its functionality is expressed in the pointcut definition, which contains the syntax that identifies the join points. In the following sections, we examine this syntax and learn how pointcuts are constructed.

---

**NOTE** There is a special form of named pointcut that omits the colon and the pointcut definition following it. Such a pointcut does not match any join point in the system. For example, the following pointcut will capture no join point:

```
pointcut threadSafeOperation();
```

We will discuss the use of this form in section 8.5.3.

---

### 3.1.1 Wildcards and pointcut operators

Given that crosscutting concerns, by definition, span multiple modules and apply to multiple join points in a system, the language must provide an economical way to capture the required join points. AspectJ utilizes a wildcard-based syntax to construct the pointcuts in order to capture join points that share common characteristics.

Three wildcard notations are available in AspectJ:

- `*` denotes any number of characters except the period.
- `..` denotes any number of characters including any number of periods.
- `+` denotes any subclass or subinterface of a given type.

Just like in Java, where unary and binary operators are used to form complex conditional expressions composed of simpler conditional expressions, AspectJ provides a unary negation operator (`!`) and two binary operators (`||` and `&&`) to form complex matching rules by combining simple pointcuts:

- *Unary operator*—AspectJ supports only one unary operation—`!` (the negation)—that allows the matching of all join points *except* those specified by the pointcut. For example, we used `!within(JoinPointTraceAspect)` in the tracing example in listing 2.9 to exclude all the join points occurring inside the `JoinPointTraceAspect`'s body.
- *Binary operators*—AspectJ offers `||` and `&&` to combine pointcuts. Combining two pointcuts with the `||` operator causes the selection of join points



that match either of the pointcuts, whereas combining them with the `&&` operator causes the selection of join points matching both the pointcuts.

The precedence between these operators is the same as in plain Java. AspectJ also allows the use of parentheses with the unary and binary operators to override the default operator precedence and make the code more legible.

### 3.1.2 Signature syntax

In Java, the classes, interfaces, methods, and fields all have signatures. You use these signatures in pointcuts to specify the places where you want to capture join points. For example, in the following pointcut, we are capturing all the calls to the `credit()` method of the `Account` class:

```
pointcut creditOperations() : call(void Account.credit(float));
```

When we specify patterns that will match these signatures in pointcuts, we refer to them as *signature patterns*. At times, a pointcut will specify a join point using one particular signature, but often it identifies join points specified by multiple signatures that are grouped together using matching patterns. In this section, we first examine three kinds of signature patterns in AspectJ—type, method, and field—and we then see how they are used in pointcut definitions in section 3.1.3.

Pointcuts that use the wildcards `*`, `..`, and `+` in order to capture join points that share common characteristics in their signatures are called *property-based pointcuts*. We have already seen an example of a signature that uses `*` and `..` in figure 3.1. Note that these wildcards have different usages in the type, method, and field signatures. We will point out these usages as we discuss the signatures and how they are matched.

#### **Type signature patterns**

The term *type* collectively refers to classes, interfaces, and primitive types. In AspectJ, *type* also refers to aspects. A type signature pattern in a pointcut specifies the join points in a type, or a set of types, at which you want to perform some crosscutting action. For a set of types, it can use wildcards, unary, and binary operators. The `*` wildcard is used in a type signature pattern to specify a part of the class, interface, or package name. The wildcard `..` is used to denote all direct and indirect subpackages. The `+` wildcard is used to denote a subtype (subclass or subinterface).

For example, the following signature matches `JComponent` and all its direct and indirect subclasses, such as `JTable`, `JTree`, `JButton`, and so on:

```
javax.swing.JComponent+
```

The `javax.swing.JComponent` portion matches the class `JComponent` in the `javax.swing` package. The `+` following it specifies that the signature will match all the subclasses of `javax.swing.JComponent` as well.

Let's look at a few examples. Note that when packages are not explicitly specified, the types are matched against the imported packages and the package to which the defining aspect or class belongs. Table 3.1 shows simple examples of matching type signatures.

**Table 3.1** Examples of type signatures

Signature Pattern	Matched Types
<code>Account</code>	Type of name <code>Account</code> .
<code>*Account</code>	Types with a name ending with <code>Account</code> such as <code>SavingsAccount</code> and <code>CheckingAccount</code> .
<code>java.*.Date</code>	Type <code>Date</code> in any of the direct subpackages of the <code>java</code> package, such as <code>java.util.Date</code> and <code>java.sql.Date</code> .
<code>java..*</code>	Any type inside the <code>java</code> package or all of its direct subpackages, such as <code>java.awt</code> and <code>java.util</code> , as well as indirect subpackages, such as <code>java.awt.event</code> and <code>java.util.logging</code> .
<code>javax..*Model+</code>	All the types in the <code>javax</code> package or its direct and indirect subpackages that have a name ending in <code>Model</code> and their subtypes. This signature would match <code>TableModel</code> , <code>TableModel</code> , and so forth, and all their subtypes.

In table 3.2, we combine type signatures with unary and binary operators.

**Table 3.2** Examples of a combined type signature using unary and binary operators

Signature Pattern	Matched Types
<code>!Vector</code>	All types other than <code>Vector</code> .
<code>Vector    Hashtable</code>	<code>Vector</code> or <code>Hashtable</code> type.
<code>javax..*Model    javax.swing.text.Document</code>	All types in the <code>javax</code> package or its direct and indirect subpackages that have a name ending with <code>Model</code> or <code>javax.swing.text.Document</code> .
<code>java.util.RandomAccess+ &amp;&amp; java.util.List+</code>	All types that implement both the specified interfaces. This signature, for example, will match <code>java.util.ArrayList</code> since it implements both the interfaces.

Although certain pointcut definitions use only a type signature pattern by itself to designate all join points in all types that match the pattern, type signature patterns

are also used within the method, constructor, and field signature patterns to further refine the selection of join points. In figure 3.1, the pointcut uses the `Account` type signature as a part of the method signature—`* Account.*(..)`. For example, if you want to identify all method call join points in a set of classes, you specify a pointcut that includes a signature pattern matching all of the type signatures of the classes, as well as the method call itself. Let's take a look at how that works.

### **Method and constructor signature patterns**

These kinds of signature patterns allow the pointcuts to identify call and execution join points in methods that match the signature patterns. Method and constructor signatures need to specify the name, the return type (for methods only), the declaring type, the argument types, and modifiers. For example, an `add()` method in a `Collection` interface that takes an `Object` argument and returns a `boolean` would have this signature:

```
public boolean Collection.add(Object)
```

The type signature patterns used in this example are `boolean`, `Collection`, and `Object`. The portion before the return value contains modifiers, such as the access specification (`public`, `private`, and so on), `static`, or `final`. These modifiers are optional, and the matching process will ignore the unspecified modifiers. For instance, unless the `final` modifier is specified, both `final` and `nonfinal` methods that match the rest of the signature will be selected. The modifiers can also be used with the negation operator to specify matching with all but the specified modifier. For example, `!final` will match all `nonfinal` methods.

When a type is used in the method signature for declaring classes, interfaces, return types, arguments, and declared exceptions, you can specify the type signature discussed in tables 3.1 and 3.2 in place of specifying exact types.

Please note that in method signatures, the wildcard `..` is used to denote any type and number of arguments taken by a method. Table 3.3 shows examples of matching method signatures.

**Table 3.3** Examples of method signatures

Signature Pattern	Matched Methods
<code>public void Collection.clear()</code>	The method <code>clear()</code> in the <code>Collection</code> class that has <code>public</code> access, returns <code>void</code> , and takes no arguments.
<code>public void Account.debit(float) throws InsufficientBalanceException</code>	The public method <code>debit()</code> in the <code>Account</code> class that returns <code>void</code> , takes a single <code>float</code> argument, and declares that it can throw <code>InsufficientBalanceException</code> .

**Table 3.3** Examples of method signatures (continued)

Signature Pattern	Matched Methods
<code>public void Account.set*(*)</code>	All public methods in the <code>Account</code> class with a name starting with <code>set</code> and taking a single argument of any type.
<code>public void Account.*()</code>	All public methods in the <code>Account</code> class that return <code>void</code> and take no arguments.
<code>public * Account.*()</code>	All public methods in the <code>Account</code> class that take no arguments and return any type.
<code>public * Account.*(..)</code>	All public methods in the <code>Account</code> class taking any number and type of arguments.
<code>* Account.*(..)</code>	All methods in the <code>Account</code> class. This will even match methods with <code>private</code> access.
<code>!public * Account.*(..)</code>	All methods with nonpublic access in the <code>Account</code> class. This will match the methods with <code>private</code> , <code>default</code> , and <code>protected</code> access.
<code>public static void Test.main(String[] args)</code>	The static <code>main()</code> method of a <code>Test</code> class with <code>public</code> access.
<code>* Account+.*(..)</code>	All methods in the <code>Account</code> class or its subclasses. This will match any new method introduced in <code>Account</code> 's subclasses.
<code>* java.io.Reader.read(..)</code>	Any <code>read()</code> method in the <code>Reader</code> class irrespective of type and number of arguments to the method. In this case, it will match <code>read()</code> , <code>read(char[])</code> , and <code>read(char[], int, int)</code> .
<code>* java.io.Reader.read(char[], ..)</code>	Any <code>read()</code> method in the <code>Reader</code> class irrespective of type and number of arguments to the method as long as the first argument type is <code>char[]</code> . In this case, it will match <code>read(char[])</code> and <code>read(char[], int, int)</code> , but not <code>read()</code> .
<code>* javax.*.add*Listener(Event- Listener+)</code>	Any method whose name starts with <code>add</code> and ends in <code>Listener</code> in the <code>javax</code> package or any of the direct and indirect subpackages that take one argument of type <code>EventListener</code> or its subtype. For example, it will match <code>TableModel.addTableModelListener(TableModelListener)</code> .
<code>* *.*(..) throws Remote- Exception</code>	Any method that declares it can throw <code>RemoteException</code> .

A constructor signature is similar to a method signature, except for two differences. First, because constructors do not have a return value, there is no return value specification required or allowed. Second, because constructors do not

have names as regular methods do, `new` is substituted for the method name in a signature. Let's consider a few examples of constructor signatures in table 3.4.

**Table 3.4** Examples of constructor signatures

Signature Pattern	Matched Constructors
<code>public Account.new()</code>	A public constructor of the <code>Account</code> class taking no arguments.
<code>public Account.new(int)</code>	A public constructor of the <code>Account</code> class taking a single integer argument.
<code>public Account.new(..)</code>	All public constructors of the <code>Account</code> class taking any number and type of arguments.
<code>public Account+.new(..)</code>	Any public constructor of the <code>Account</code> class or its subclasses.
<code>public *Account.new(..)</code>	Any public constructor of classes with names ending with <code>Account</code> . This will match all the public constructors of the <code>SavingsAccount</code> and <code>CheckingAccount</code> classes.
<code>public Account.new(..) throws InvalidAccountNumberException</code>	Any public constructors of the <code>Account</code> class that declare they can throw <code>InvalidAccountNumberException</code> .

### **Field signature patterns**

Much like the method signature, the field signature allows you to designate a member field. You can then use the field signatures to capture join points corresponding to read or write access to the specified fields. A field signature must specify the field's type, the declaring type, and the modifiers. Just as in method and constructor signatures, you can use type signature patterns to specify the types. For example, this designates a public integer field `x` in the `Rectangle` class:

```
public int java.awt.Rectangle.x
```

Let's dive straight into a few examples in table 3.5.

**Table 3.5** Examples of field signatures

Signature Pattern	Matched Fields
<code>private float Account._balance</code>	Private field <code>_balance</code> of the <code>Account</code> class
<code>* Account.*</code>	All fields of the <code>Account</code> class regardless of an access modifier, type, or name

**Table 3.5** Examples of field signatures (continued)

Signature Pattern	Matched Fields
<code>!public static * banking...*</code>	All nonpublic <code>static</code> fields of <code>banking</code> and its direct and indirect subpackages
<code>public !final *.*</code>	Nonfinal public fields of any class

Now that you understand the syntax of the signatures, let's see how to put them together into pointcuts.

### 3.1.3 Implementing pointcuts

Let's recap: Pointcuts are program constructs that capture a set of exposed join points by matching certain characteristics. Although a pointcut can specify a single join point in a system, the power of pointcuts comes from the economical way they match a set of join points.

There are two ways that pointcut designators match join points in AspectJ. The first way captures join points based on the category to which they belong. Recall from the discussion in section 2.4.1 that join points can be grouped into categories that represent the kind of join points they are, such as method call join points, method execution join points, field get join points, exception handler join points, and so forth. The pointcuts that map directly to these categories or *kinds* of exposed join points are referred to as *kinded* pointcuts.

The second way that pointcut designators match join points is when they are used to capture join points based on matching the circumstances under which they occur, such as control flow, lexical scope, and conditional checks. These pointcuts capture join points in any category as long as they match the prescribed condition. Some of the pointcuts of this type also allow the collection of context at the captured join points. Let's take a more in-depth look at each of these types of pointcuts.

#### ***Kinded pointcuts***

Kinded pointcuts follow a specific syntax to capture each kind of exposed join point in AspectJ. Once you understand the categories of exposed join points, as discussed in section 2.4.1, you will find that understanding kinded pointcuts is simple—all you need is their syntax. Table 3.6 shows the syntax for each of the kinded pointcuts.

When you understand the pointcut syntax in table 3.6 and the signature syntax as described in section 3.1.2, you will be able to write kinded pointcuts that

**Table 3.6 Mapping of exposed join points to pointcut designators**

Join Point Category	Pointcut Syntax
Method execution	<code>execution(MethodSignature)</code>
Method call	<code>call(MethodSignature)</code>
Constructor execution	<code>execution(ConstructorSignature)</code>
Constructor call	<code>call(ConstructorSignature)</code>
Class initialization	<code>staticinitialization(TypeSignature)</code>
Field read access	<code>get(FieldSignature)</code>
Field write access	<code>set(FieldSignature)</code>
Exception handler execution	<code>handler(TypeSignature)</code>
Object initialization	<code>initialization(ConstructorSignature)</code>
Object pre-initialization	<code>preinitialization(ConstructorSignature)</code>
Advice execution	<code>adviceexecution()</code>

capture the weaving points in the system. Once you express the pointcuts in this fashion, you can use them as a part of dynamic crosscutting in the advice construct as well as in static crosscutting constructs. For example, to capture all public methods in the `Account` class, you use a `call()` pointcut along with one of the signatures in table 3.3 to encode the pointcut as follows:

```
call(public * Account.*())
```

Similarly, to capture all write accesses to a private `_balance` field of type `float` in the `Account` class, you would use a `set()` pointcut with the signature described in table 3.3 to encode the pointcut as follows:

```
set(private float Account._balance)
```

Let's take a quick look at an example of how a pointcut is used in static crosscutting. In the following snippet, we declare that calling the `Logger.log()` method will result in a compile-time warning. The pointcut `call(void Logger.log(..))` is a kinded pointcut of the method call category type. We will discuss the compile-time error and warning declaration in section 3.3.3:

```
declare warning : call(void Logger.log(..))
                : "Consider Logger.logp() instead";
```

Now that we've examined the kinded pointcuts, let's look at the other type of pointcut—the ones that capture join points based on specified conditions

regardless of the kind of join point it is. This type of pointcut offers a powerful way to capture certain complex weaving rules.

### **Control-flow based pointcuts**

These pointcuts capture join points based on the control flow of join points captured by another pointcut. The control flow of a join point defines the flow of the program instructions that occur as a result of the invocation of the join point. Think of control flow as similar to a call stack. For example, the `Account.debit()` method calls `Account.getBalance()` as a part of its execution; the call and the execution of `Account.getBalance()` is said to have occurred in the `Account.debit()` method's control flow, and therefore it has occurred in the control flow of the join point for the method. In a similar manner, it captures other methods, field access, and exception handler join points within the control flow of the method's join point.

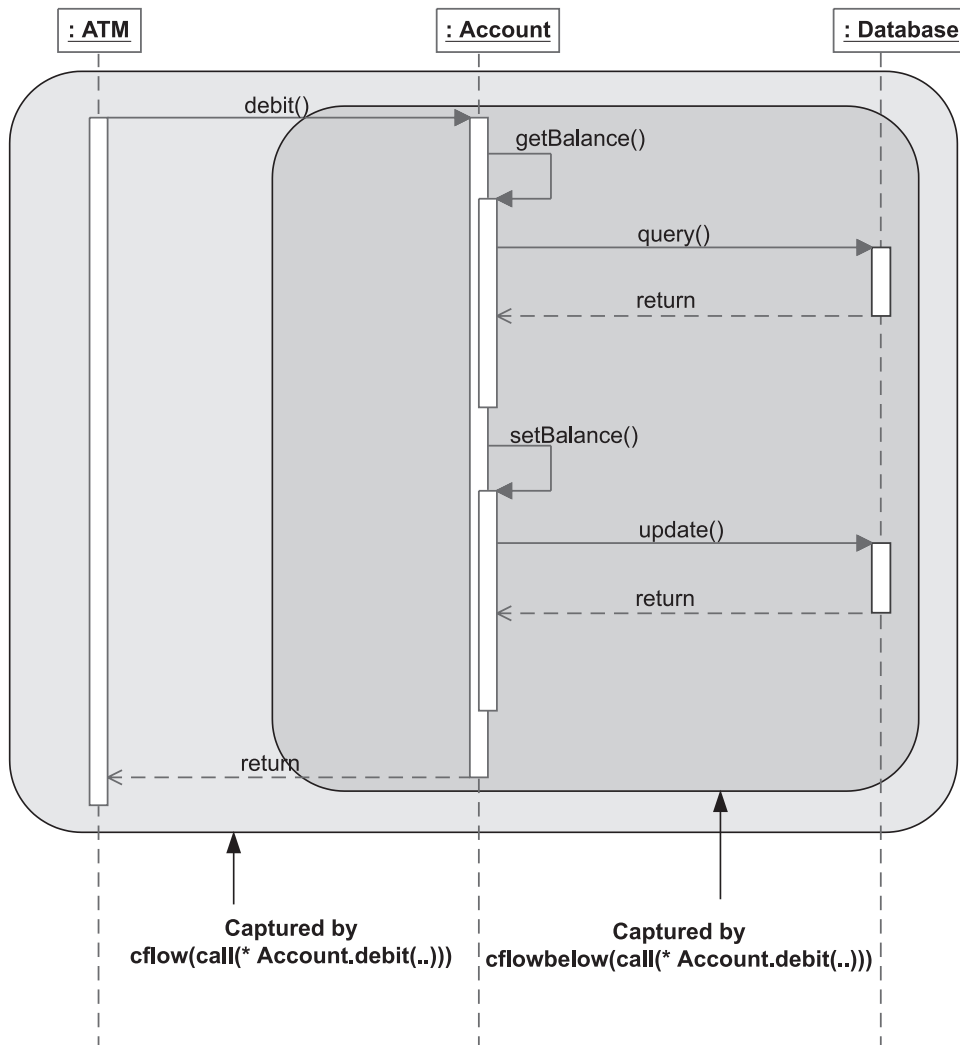
A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts. The first pointcut is expressed as `cflow(Pointcut)`, and it captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself. The second pointcut is expressed as `cflowbelow(Pointcut)`, and it excludes the join points in the specified pointcut. Table 3.7 shows some examples of the usage of control-flow based pointcuts.

**Table 3.7** Examples of control-flow based pointcuts

Pointcut	Description
<code>cflow(call(* Account.debit(..))</code>	All the join points in the control flow of any <code>debit()</code> method in <code>Account</code> that is called, including the call to the <code>debit()</code> method itself
<code>cflowbelow(call(* Account.debit(..))</code>	All the join points in the control flow of any <code>debit()</code> method in <code>Account</code> that is called, but excluding the call to the <code>debit()</code> method itself
<code>cflow(transactedOperations())</code>	All the join points in the control flow of the join points captured by the <code>transactedOperations()</code> pointcut
<code>cflowbelow(execution(Account.new(..))</code>	All the join points in the control flow of any of the <code>Account</code> 's constructor execution, excluding the constructor execution itself
<code>cflow(staticinitializer(BankingDatabase))</code>	All the join points in the control flow occurring during the class initialization of the <code>BankingDatabase</code> class



The sequence diagram in figure 3.2 shows the graphical representation of the `cflow()` and `cflowbelow()` pointcuts. Here, the area encompassing the captured join points is superimposed on a sequence diagram that shows an



**Figure 3.2** Control-flow based pointcuts capture every join point occurring in the control flow of join points matching the specified pointcut. The `cflow()` pointcut includes the matched join point itself, thus encompassing all join points occurring inside the outer box, whereas `cflowbelow()` excludes that join point and thus captures only join points inside the inner box.

`Account.debit()` method that is called by an ATM object. The difference between the matching performed by the `cflow()` and `cflowbelow()` pointcuts is also depicted.

One common usage of `cflowbelow()` is to select nonrecursive calls. For example, `transactedOperations() && !cflowbelow(transactedOperations())` will select the methods that are not already in the context of another method captured by the `transactedOperations()` pointcut.

### **Lexical-structure based pointcuts**

A lexical scope is a segment of source code. It refers to the scope of the code as it was written, as opposed to the scope of the code when it is being executed, which is the dynamic scope. Lexical-structure based pointcuts capture join points occurring inside a lexical scope of specified classes, aspects, and methods. There are two pointcuts in this category: `within()` and `withincode()`. The `within()` pointcuts take the form of `within(TypePattern)` and are used to capture all the join points within the body of the specified classes and aspects, as well as any nested classes. The `withincode()` pointcuts take the form of either `withincode(MethodSignature)` or `withincode(ConstructorSignature)` and are used to capture all the join points inside a lexical structure of a constructor or a method, including any local classes in them. Table 3.8 shows some examples of the usage of lexical-structure based pointcuts.

**Table 3.8** Examples of lexical-structure based pointcuts

Pointcut	Natural Language Description
<code>within(Account)</code>	Any join point inside the <code>Account</code> class's lexical scope
<code>within(Account+)</code>	Any join point inside the lexical scope of the <code>Account</code> class and its subclasses
<code>withincode(* Account.debit(..))</code>	Any join point inside the lexical scope of any <code>debit()</code> method of the <code>Account</code> class
<code>withincode(* *Account.getBalance(..))</code>	Any join point inside the lexical scope of the <code>getBalance()</code> method in classes whose name ends in <code>Account</code>

One common usage of the `within()` pointcut is to exclude the join points in the aspect itself. For example, the following pointcut excludes the join points corresponding to the calls to all `print` methods in the `java.io.PrintStream` class that occur inside the `TraceAspect` itself:

```
call(* java.io.PrintStream.print*(..) && !within(TraceAspect))
```

### Execution object pointcuts

These pointcuts match the join points based on the types of the objects at execution time. The pointcuts capture join points that match either the type `this`, which is the current object, or the `target` object, which is the object on which the method is being called. Accordingly, there are two execution object pointcut designators: `this()` and `target()`. In addition to matching the join points, these pointcuts are used to collect the context at the specified join point.

The `this()` pointcut takes the form `this(Type or ObjectIdentifier)`; it matches all join points that have a `this` object associated with them that is of the specified type or the specified *ObjectIdentifier*'s type. In other words, if you specify *Type*, it will match the join points where the expression `this instanceof <Type>` is true. The form of this pointcut that specifies *ObjectIdentifier* is used to collect the `this` object. If you need to match without collecting context, you will use the form that uses *Type*, but if you need to collect the context, you will use the form that uses *ObjectIdentifier*. We discuss context collection in section 3.2.6.

The `target()` pointcut is similar to the `this()` pointcut, but uses the target of the join point instead of `this`. The `target()` pointcut is normally used with a method call join point, and the target object is the one on which the method is invoked. A `target()` pointcut takes the form `target(Type or ObjectIdentifier)`. Table 3.9 shows some examples of the usage of execution object pointcuts.

**Table 3.9** Examples of execution object pointcuts

Pointcut	Natural Language Description
<code>this(Account)</code>	All join points where <code>this</code> is <code>instanceof Account</code> . This will match all join points like methods calls and field assignments where the current execution object is <code>Account</code> , or its subclass, for example, <code>SavingsAccount</code> .
<code>target(Account)</code>	All the join points where the object on which the method called is <code>instanceof Account</code> . This will match all join points where the target object is <code>Account</code> , or its subclass, for example, <code>SavingsAccount</code> .

Note that unlike most other pointcuts that take the *TypePattern* argument, `this()` and `target()` pointcuts take *Type* as their argument. So, you cannot use the `*` or `..` wildcard while specifying a type. You don't need to use the `+` wildcard since subtypes that match are already captured by Java inheritance without `+`; adding `+` will not make any difference.

Because static methods do not have the `this` object associated with them, the `this()` pointcut will not match the execution of such a method. Similarly,

because static methods are not invoked on a object, the `target()` pointcut will not match calls to such a method.

There are a few important differences in the way matching is performed between `within()` and `this()`: The former will match when the object in the lexical scope matches the type specified in the pointcut, whereas the latter will match when the current execution object is of a type that is specified in the pointcut or its subclass. The code snippet that follows shows the difference between the two pointcuts. We have a `SavingsAccount` class that extends the `Account` class. The `Account` class also contains a nested class: `Helper`. The join points that will be captured by `within(Account)` and `this(Account)` are annotated.

```
public class Account {
    ...
    public void debit(float amount)
        throws InsufficientBalanceException {
        ...
    }

    private static class Helper {
        ...
    }
}

public class SavingsAccount extends Account {
    ...
}
```

**Captured by  
within(Account)**

**Captured by  
this(Account)**

**Captured by  
within(Account)**

**Captured by  
this(Account)**

In this example, `within(Account)` will match all join points inside the definition of the `Account` class, including any nested classes, but no join points inside its subclasses, such as `SavingsAccount`. On the other hand, `this(Account)` will match all join points inside the definition of the `Account` class as well as `SavingsAccount`, but will exclude any join points inside either class's nested classes. You can match all the join points in subclasses of a type while excluding the type itself by using the `this(Type) && !within(Type)` idiom. Another difference between the two pointcuts is their context collection capability: `within()` cannot collect any context, but `this()` can.

Also note that the two pointcuts `call(* Account.*(..))` and `call(* *.*(..)) && this(Account)` won't capture the same join points. The first one will pick up all the instance and static methods defined in the `Account` class and all the parent classes in the inheritance hierarchy, whereas the latter will pick up the same instance methods and any methods in the subclasses of the `Account` class, but none of the static methods.

### Argument pointcuts

These pointcuts capture join points based on the argument type of a join point. For method and constructor join points, the arguments are simply the method and constructor arguments. For exception handler join points, the handled exception object is considered an argument, whereas for field write access join points, the new value to be set is considered the argument for the join point. Argument-based pointcuts take the form of `args(TypePattern or ObjectIdentifier, ..)`.

Similar to execution object pointcuts, these pointcuts can be used to capture the context, but again more will be said about this in section 3.2.6. Table 3.10 shows some examples of the usage of argument pointcuts.

**Table 3.10** Examples of argument pointcuts

Pointcut	Natural Language Description
<code>args(String, .., int)</code>	All the join points in all methods where the first argument is of type <code>String</code> and the last argument is of type <code>int</code> .
<code>args(RemoteException)</code>	All the join points with a single argument of type <code>RemoteException</code> . It would match a method taking a single <code>RemoteException</code> argument, a field write access setting a value of type <code>RemoteException</code> , or an exception handler of type <code>RemoteException</code> .

### Conditional check pointcuts

This pointcut captures join points based on some conditional check at the join point. It takes the form of `if(BooleanExpression)`. Table 3.11 shows some examples of the usage of conditional check pointcuts.

**Table 3.11** Examples of conditional check pointcuts

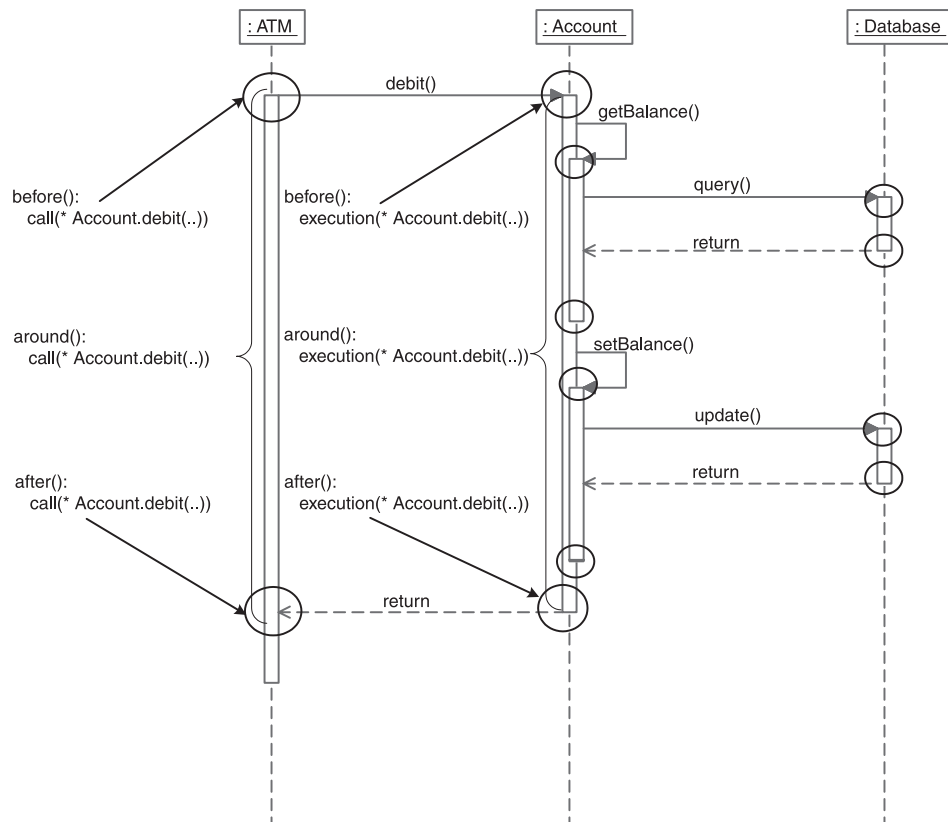
Pointcut	Natural Language Description
<code>if(System.currentTimeMillis() &gt; triggerTime)</code>	All the join points occurring after the current time has crossed the <code>triggerTime</code> value.
<code>if(circle.getRadius() &lt; 5)</code>	All the join points where the <code>circle</code> 's radius is smaller than 5. The <code>circle</code> object must be a context collected by the other parts of the pointcut. See section 3.2.6 for details about the context-collection mechanism.

We now have completed the overview of all the pointcuts supported in AspectJ. In the next section, we study the dynamic crosscutting concept of advice. Writing an advice entails first specifying a pointcut and then defining the action to be taken at the join points captured by the pointcut. Later, in section 3.3, we discuss using pointcuts for static crosscutting.

## 3.2 Advice

Advice is the action and decision part of the crosscutting puzzle. It helps you define “what to do.” Advice is a method-like construct that provides a way to express crosscutting action at the join points that are captured by a pointcut. The three kinds of advice are as follows:

- *Before advice* executes prior to the join point.
- *After advice* executes following the join point.
- *Around advice* surrounds the join point’s execution. This advice is special in that it has the ability to bypass execution, continue the original execution, or cause execution with an altered context.



**Figure 3.3** Various points in a program flow where you can advise the join point (not all possible points are shown). Each circle represents an opportunity for before or after advice. The passage between the matching circles on each lifeline represents an opportunity for around advice.

Join points exposed by AspectJ are the only points where you apply an advice. Figure 3.3 shows various join points in an execution sequence at which you can introduce a new behavior via advice.

### 3.2.1 Anatomy of advice

Let's look at the general syntactical structure of an advice. We will study the details of each kind of advice—before, after, and around—in subsequent sections. An advice can be broken into three parts: the advice declaration, the pointcut specification, and the advice body. Let's look at two examples of these three parts. Both examples will use the following named pointcut:

```
pointcut connectionOperation(Connection connection)
    : call(* Connection.*(..) throws SQLException)
    && target(connection);
```

This named pointcut consists of two anonymous pointcuts. The method `call` pointcut captures calls to any method of the `Connection` class that takes any argument and returns any type. The `target()` pointcut captures the `target` object of the method calls. Now let's look at an example of before and around advice using the named pointcut:

```
before(Connection connection): 1 Advice declaration
    connectionOperation (connection) { 2 Pointcut specification
        System.out.println("Performing operation on " + connection); 3
    }

Object around(Connection connection) throws SQLException 1
    : connectionOperation (connection) { 2
        System.out.println("Operation " + thisJoinPoint
            + " on " + connection
            + " started at "
            + System.currentTimeMillis());

        proceed(connection);

        System.out.println("Operation " + thisJoinPoint
            + " on " + connection
            + " completed at "
            + System.currentTimeMillis());
    } 3 Advice body
```

- 1 The part before the colon is the advice declaration, which specifies when the advice executes relative to the captured join point—before, after, or around it. The advice declaration also specifies the context information available to the advice body, such as the execution object and arguments, which the advice body

can use to perform its logic in the same way a method would use its parameters. It also specifies any checked exceptions thrown by the advice.

- ② The part after the colon is the pointcut; the advice executes whenever a join point matching the pointcut is encountered. In our case, we use the named pointcut, `connectionOperation()`, in the advice to log join points captured by the pointcut.
- ③ Just like a method body, the advice body contains the actions to execute and is within the `{}`. In the example, the before advice body prints the context collected by the pointcut, whereas the around advice prints the start and completion time of each connection operation. `thisJoinPoint` is a special variable available in each join point. We will study its details in the next chapter, section 4.1. In around advice, the `proceed()` statement is a special syntax to carry out the captured operation that we examine in section 3.2.4.

Let's take a closer look at each type of advice.

### 3.2.2 The before advice

The before advice executes before the execution of the captured join point. In the following code snippet, the advice performs authentication prior to the execution of any method in the `Account` class:

```
before() : call(* Account.*(..)) {  
    ... authenticate the user  
}
```

If you throw an exception in the before advice, the captured operation won't execute. For example, if the authentication logic in the previous advice throws an exception, the method in `Account` that is being advised won't execute. The before advice is typically used for performing pre-operation tasks, such as policy enforcement, logging, and authentication.

### 3.2.3 The after advice

The after advice executes after the execution of a join point. Since it is often important to distinguish between normal returns from a join point and those that throw an exception, AspectJ offers three variations of after advice: after returning normally, after returning by throwing an exception, and returning either way. The following code snippet shows the basic form for after advice that returns either way:

```
after() : call(* Account.*(..)) {  
    ... log the return from operation  
}
```



The previous advice will be executed after any call to any method in the `Account` class, regardless of how it returns—normally or by throwing an exception. Note that an after advice may be used not just with methods but with any other kind of join point. For example, you could advise a constructor invocation, field write-access, exception handler, and so forth.

It is often desirable to apply an advice only after a successful completion of captured join points. AspectJ offers “after returning” advice that is executed after the successful execution of join points. The following code shows the form for after returning advice:

```
after() returning : call(* Account.*(..)) {
    ... log the successful completion
}
```

This advice will be executed after the successful completion of a call to any method in the `Account` class. If a captured method throws an exception, the advice will not be executed. AspectJ offers a variation of the after returning advice that will capture the return value. It has the following syntax:

```
after() returning(<ReturnType returnObject>)
```

You can use this form of the after returning advice when you want to capture the object that is returned by the advised join point so that you can use its context in the advice. Note that unless you want to capture the context, you don’t need to supply the parentheses following `returning`. See section 3.2.6 for more details on collecting the return object as context.

Similar to after returning advice, AspectJ offers “after throwing” advice, except such advice is executed only when the advised join point throws an exception. This is the form for after advice that returns after throwing an exception:

```
after() throwing : call(* Account.*(..)) {
    ... log the failure
}
```

This advice will be executed after a call to any method in the `Account` class that throws an exception. If a method returns normally, the advice will not be executed. Similar to the variation in the after returning advice, AspectJ offers a variation of the after throwing advice that will capture the thrown exception object. The advice has the following syntax:

```
after() throwing (<ExceptionType exceptionObject>)
```

You can use this form of the after throwing advice when you want to capture the exception that is thrown by the advised method so that you can use it to make

decisions in the advice. See section 3.2.6 for more details on capturing the exception object.

### 3.2.4 The around advice

The around advice surrounds the join point. It has the ability to bypass the execution of the captured join point completely, or to execute the join point with the same or different arguments. It may also execute the captured join points multiple times, each with different arguments. Some typical uses of this advice are to perform additional execution before and after the advised join point, to bypass the original operation and perform some other logic in place of it, or to surround the operation with a try/catch block to perform an exception-handling policy.

If within the around advice you want to execute the operation that is at the join point, you must use a special keyword—`proceed()`—in the body of the advice. Unless you call `proceed()`, the captured join point will be bypassed. When using `proceed()`, you can pass the context collected by the advice, if any, as the arguments to the captured operation or you can pass completely different arguments. The important thing to remember is that you must pass the same number and types of arguments as collected by the advice. Since `proceed()` causes the execution of the captured operation, it returns the same value returned by the captured operation. For example, while in an advice to a method that returns a float value, invoking `proceed()` will return the same float value as the captured method. We will study the details of returning a value from an around advice in section 3.2.7.

In the following snippet, the around advice invokes `proceed()` with a try/catch block to handle exceptions. This snippet also captures the context of the operation's target object and argument. We discuss that part in section 3.2.6:

```
void around(Account account, float amount)
    throws InsufficientBalanceException :
    call(* Account.debit(float) throws InsufficientBalanceException)
    && target(account)
    && args(amount) {
    try {
        proceed(account, amount);
    } catch (InsufficientBalanceException ex) {
        ... overdraft protection logic
    }
}
```

In the previous advice, the advised join point is the call to the `Account.debit()` method that throws `InsufficientBalanceException`. We capture the `Account`

object and the amount using the `target()` and `args()` pointcuts. In the body of the advice, we surround the call to `proceed()` with a try/catch block, with the catch block performing overdraft protection logic. The result is that when the advice is executed, it in turn executes the captured method using `proceed()`. If an exception is thrown, the catch block executes the overdraft protection logic using the context that it captured in the `target()` and `args()` pointcuts.

### 3.2.5 Comparing advice with methods

As you can see, the advice declaration part looks much like a method signature. Although it does not have a name, it takes arguments and may declare that it can throw exceptions. The arguments form the context that the advice body can use to perform its logic, just like in a method. The before and after advice cannot return anything, while the around advice does and therefore has a return type. The pointcut specification part uses named or anonymous pointcuts to capture the join points to be advised. The body of advice looks just like a method body except for the special keyword `proceed()` that is available in the around advice.

By now, you might be thinking that advice looks an awful lot like methods. Let's contrast the two here. Like methods, advice:

- Follows access control rules to access members from other types and aspects
- Declares that it can throw checked exceptions
- Can refer to the aspect instance using `this`

Unlike methods, however, advice:

- Does not have a name
- Cannot be called directly (it's the system's job to execute it)
- Does not have an access specifier (this makes sense because you cannot directly call advice anyway)
- Has access to a few special variables besides `this` that carry information about the captured join point: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinpointStaticPart` (we examine these variables in chapter 4)

One way to think of advice is that it overrides the captured join points, and in fact, the exception declaration rules for advice actually do follow the Java specification for overridden methods. Like overridden methods, advice:

- Cannot declare that it may throw a checked exception that is not already declared by the captured join point. For example, when your aspect is

implementing persistence, you are not allowed to declare that the advice may throw `SQLException` unless the method that was captured by the join point already declares that it throws it.

- May omit a few exceptions declared by the captured join points.
- May declare that it can throw more specific exceptions than those declared by the captured join points.

Chapter 4 discusses the issue of dealing with additional checked exceptions in more depth and shows a pattern for addressing the common situations.

### 3.2.6 Passing context from a join point to advice

Advice implementations often require access to data at the join point. For example, to log certain operations, advice needs information about the method and arguments of the operation. This information is called *context*. Pointcuts, therefore, need to expose the context at the point of execution so it can be passed to the advice implementation. AspectJ provides the `this()`, `target()`, and `args()` pointcuts to collect the context. You'll recall that there are two ways to specify each of these pointcuts: by using the type of the objects or by using *ObjectIdentifier*, which simply is the name of the object. When context needs to be passed to the advice, you use the form of the pointcuts that use *ObjectIdentifier*.

In a pointcut, the object identifiers for the collected objects must be specified in the first part of the advice—the part before the colon—in much the same way you would specify method arguments. For example, in figure 3.4, the anonymous pointcut in the before advice collects all the arguments to the method executions associated with it.

```
before (Account account, float amount) :
    call (void Account.credit(float))
    && target (~account)
    && args (amount) {
        System.out.println("Crediting " + amount
            + " to " + account);
    }
```

Passing argument value    Passing target object

**Figure 3.4** Passing an executing object and an argument context from the join point to the advice body. The target object in this case is captured using the `target()` pointcut, whereas the argument value is captured using the `args()` pointcut. The current execution object can be captured in the same way using `this()` instead of `target()`.

Figure 3.4 shows the context being passed between an anonymous pointcut and the advice. The `target()` pointcut collects the objects on which the `credit()` method is being invoked, whereas the `args()` pointcut captures the argument to the method. The part of the advice before the colon specifies the type and name for each of the captured arguments. The body of the advice uses the collected context in the same way that the body of a method would use the parameters passed to it. The object identifiers in the previous code snippet are `account` and `amount`.

When you use named pointcuts, those pointcuts themselves must collect the context and pass it to the advice. Figure 3.5 shows the collection of the same information as in figure 3.4, but uses named pointcuts to capture the context and make it available to the advice.

The code in figure 3.5 is functionally identical to that in 3.4, but unlike figure 3.4, we use a named pointcut. The pointcut `creditOperation()`, besides matching join points, collects the context so that the advice can use it. We collect the target object and the argument to the `credit()` operation. Note that the pointcut itself declares the type and name of each collected element, much like a method call. In the advice to this pointcut, the first part before the colon is unchanged from figure 3.4. The pointcut definition simply uses the earlier defined pointcut. Note how the names of the arguments in the first part of the advice match those in the pointcut definition.

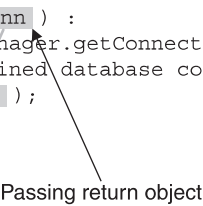
Let's look at some more examples of passing context. In figure 3.6, an after returning advice captures the return value of a method.

```
pointcut creditOperation(Account account, float amount) :
    call (void Account.credit(float)
    && target ( account )
    && args ( amount );

before (Account account, float amount) :
    creditOperation(account, amount) {
    System.out.println("Crediting " + amount
    + " to " + account );
}
```

**Figure 3.5** Passing an executing object and an argument captured by a named pointcut. This code snippet is functionally equivalent to figure 3.4, but achieves it using a named pointcut. For the advice to access the join point's context, the pointcut itself must collect the context, as opposed to the advice collecting the context when using anonymous pointcuts.

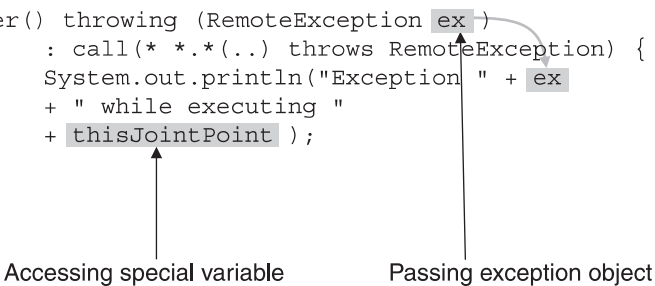
```
after() returning (Connection conn) :
    call(Connection DriverManager.getConnection(..)) {
        System.out.println("Obtained database connection: "
            + conn);
    }
```



Passing return object

**Figure 3.6** Passing a return object context to an advice body. The return object is captured in `returning()` by specifying the type and object ID.

```
after() throwing (RemoteException ex)
    : call(* *.*(..) throws RemoteException) {
    System.out.println("Exception " + ex
        + " while executing "
        + thisJoinPoint);
    }
```



Accessing special variable

Passing exception object

**Figure 3.7** Passing a thrown exception to an advice body. The exception object is captured in `throwing()` by specifying the type and object ID. The special variables such as `thisJoinPoint` are accessed in a similar manner to `this` inside an instance method.

In figure 3.6, we capture the return value of `DriverManager.getConnection()` by specifying the type and the name of the return object in the `returning()` part of the advice specification. We can use the return object in the advice body just like any other collected context. In this example, the advice simply prints the return value.

In figure 3.7, we capture the exception object thrown by any method that declares that it can throw `RemoteException` by specifying the type and name of the exception to the `throwing()` part of the advice specification. Much like the return value and any other context, we can use this exception object in the advice body.

Note that `thisJoinPoint` is a special type of variable that carries join point context information. We will look at these types of variables in detail in chapter 4.

### 3.2.7 Returning a value from around advice

Each around advice must declare a return value (which could be `void`). It is typical to declare the return type to match the return type of the join points that are being advised. For example, if a set of methods that are each returning an integer were advised, you would declare the advice to return an integer. For a field-read join point, you would match the advice's return type to the accessed field's type.

Invoking `proceed()` returns the value returned by the join point. Unless you need to manipulate the returned value, around advice will simply return the value that was returned by the `proceed()` statement within it. If you do not invoke `proceed()`, you will still have to return a value appropriate for the advice's logic.

There are cases when an around advice applies to join points with different return types. For example, if you advise all the methods needing transaction support, the return values of all those methods are likely to be different. To resolve such situations, the around advice may declare its return value as `Object`. In those cases, if around returns a primitive type after it calls `proceed()`, the primitive type is wrapped in its corresponding wrapper type and performs the opposite, unwrapping after returning from the advice. For instance, if a join point returns an integer and the advice declares that it will return `Object`, the integer value will be wrapped in an `Integer` object and it will be returned from the advice. When such a value is assigned, the object is first unwrapped to an integer. Similarly, if a join point returns a non-primitive type, appropriate typecasts are performed before the return value is assigned. The scheme of returning the `Object` type works even when a captured join point returns a `void` type.

### 3.2.8 An example using around advice: failure handling

Let's look at an example that uses around advice to handle system failures. In a distributed environment, dealing with a network failure is often an important task. If the network is down, clients often reattempt operations. In the following example, we examine how an aspect with around advice can implement the functionality to handle a network failure.

In listing 3.1, we simulate the network and other failures by simply making the method throw an exception randomly.

**Listing 3.1** RemoteService.java

```
import java.rmi.RemoteException;

public class RemoteService {
    public static int getReply() throws RemoteException {
        if(Math.random() > 0.25) {
            throw new RemoteException("Simulated failure occurred");
        }
        System.out.println("Replying");
        return 5;
    }
}
```

The `getReply()` method simulates the service offered. By checking against a randomly generated number, it simulates a failure resulting in an exception (statistically, the method will fail approximately 75 percent of the time—a really high failure rate!). When it does not fail, it prints a message and returns 5.

Next let's write a simple client (listing 3.2) that invokes the only method in `RemoteService`.

**Listing 3.2 RemoteClient.java**

```
public class RemoteClient {
    public static void main(String[] args) throws Exception {
        int retVal = RemoteService.getReply();
        System.out.println("Reply is " + retVal);
    }
}
```

Now let's write an aspect to handle failures by reattempting the operation three times before giving up and propagating the failure to the caller (listing 3.3).

**Listing 3.3 FailureHandlingAspect.java**

```
import java.rmi.RemoteException;

public aspect FailureHandlingAspect {
    final int MAX_RETRIES = 3;

    Object around() throws RemoteException
        : call(* RemoteService.get*(..) throws RemoteException) {
        int retry = 0;
        while(true){
            try{
                return proceed();
            } catch(RemoteException ex){
                System.out.println("Encountered " + ex);
                if (++retry > MAX_RETRIES) {
                    throw ex;
                }
                System.out.println("\tRetrying...");
            }
        }
    }
}
```

- 1 We declare that the `around` advice will return `Object` to accommodate the potential different return value types in the captured join points. We also declare that



it may throw `RemoteException` to allow the propagating of any exception thrown by the execution of captured join points.

- ② The pointcut part of the advice uses an anonymous pointcut to capture all the getter methods in `RemoteService` that throw `RemoteException`.
- ③ We simply return the value returned by the invocation of `proceed()`. Although the join point is returning an integer, AspectJ will take care of wrapping and unwrapping the logic.

When we compile and run the program, we get output similar to the following:

```
> ajc RemoteService.java RemoteClient.java FailureHandlingAspect.java
> java RemoteClient
Encountered java.rmi.RemoteException: Simulated failure occurred
    Retrying...
Encountered java.rmi.RemoteException: Simulated failure occurred
    Retrying...
Replaying
Reply is 5
```

The output shows a few failures, retries, and eventual success. (Your output may be a little different due to the randomness introduced.) It also shows the correct assignment to the `retVal` member in the `RemoteClient` class, even though the advice returned the `Object` type.

### 3.2.9 Context collection example: caching

The goal of this example is to understand how to collect context in arguments, execution objects, and return values. First, we write a method for a simple factorial computation, and then we write an aspect to cache the computed value for later use. We want to insert a result into the cache for values passed on to only nonrecursive calls (to limit the amount of caching). Before any calls to the `factorial()` method, including the recursive ones, we check the cache and print the value if a precomputed value is found. Otherwise, we proceed with the normal computation flow. Let's start with creating the factorial computation in listing 3.4.

**Listing 3.4** `TestFactorial.java`: factorial computation

```
import java.util.*;

public class TestFactorial {
    public static void main(String[] args) {
        System.out.println("Result: " + factorial(5) + "\n");
        System.out.println("Result: " + factorial(10) + "\n");
        System.out.println("Result: " + factorial(15) + "\n");
    }
}
```

```

        System.out.println("Result: " + factorial(15) + "\n");
    }

    public static long factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n-1);
        }
    }
}

```

Now let's write the aspect to optimize the factorial computation by caching the computed value for later use, as shown in listing 3.5.

**Listing 3.5** OptimizeFactorialAspect.java: aspect for caching results

```

import java.util.*;

public aspect OptimizeFactorialAspect {
    pointcut factorialOperation(int n) :
        call(long *.factorial(int)) && args(n);

    pointcut topLevelFactorialOperation(int n) :
        factorialOperation(n)
        && !cflowbelow(factorialOperation(int));

    private Map _factorialCache = new HashMap();

    before(int n) : topLevelFactorialOperation(n) {
        System.out.println("Seeking factorial for " + n);
    }

    long around(int n) : factorialOperation(n) {
        Object cachedValue = _factorialCache.get(new Integer(n));
        if (cachedValue != null) {
            System.out.println("Found cached value for " + n
                + ": " + cachedValue);
            return ((Long) cachedValue).longValue();
        }
        return proceed(n);
    }

    after(int n) returning(long result) :
        topLevelFactorialOperation(n) {
        _factorialCache.put(new Integer(n), new Long(result));
    }
}

```

❶ Capturing context using args()

❷ Capturing context from another pointcut

❸ Using pointcut's context

❹ Returning primitive from around advice

❺ Passing along context to proceed()

❻ Capturing return value

- ❶ The `factorialOperation()` pointcut captures all calls to the `factorial()` method. It also collects the argument to the method.
- ❷ The `topLevelFactorialOperation()` pointcut captures all nonrecursive calls to the `factorial()` method. It captures the context available in any `factorialOperation()` pointcut it uses. See figure 3.5 for a graphical representation of capturing context using named pointcuts.
- ❸ The `before` advice logs the nonrecursive `factorial()` method invocation. In the log message, it uses the collected context.
- ❹ The `around` advice to any `factorial()` method invocation also uses the context. It declares that it will return a `long` matching the return type of the advised join point.
- ❺ The `around` advice passes the captured context to `proceed()`. Recall that the number and type of arguments to `proceed()` must match the advice itself.
- ❻ The `after` returning advice collects the return value by specifying its type and identifier in the `returning()` part. It then uses the return value as well as the context collected from the join point to update the cache.

When we compile and run the code, we get the following output:

```
> ajc TestFactorial.java OptimizeFactorialAspect.java
> java TestFactorial
Seeking factorial for 5
Result: 120

Seeking factorial for 10
Found cached value for 5: 120
Result: 3628800

Seeking factorial for 15
Found cached value for 10: 3628800
Result: 1307674368000

Seeking factorial for 15
Found cached value for 15: 1307674368000
Result: 1307674368000
```

As soon as a cached value is found, the factorial computation uses that value instead of continuing with the recursive computation. For example, while computing a factorial for 15, the computation uses a pre-cached factorial value for 10.

---

**NOTE** It seems that you could simply modify the `Test.factorial()` method to insert code for caching optimization, especially since only one method needs to be modified. However, such an implementation will tangle the optimization logic with factorial computation logic. With conventional

refactoring techniques, you can limit the inserted code to a few lines. Using an aspect, you refactor the caching completely out of the core factorial computation code. You can now modify the caching strategy without even touching the `factorial()` method.

### 3.3 Static crosscutting

In AOP, we often find that in addition to affecting dynamic behavior using advice, it is necessary for aspects to affect the static structure in a crosscutting manner. While dynamic crosscutting modifies the execution behavior of the program, static crosscutting modifies the static structure of the types—the classes, interfaces, and other aspects—and their compile-time behavior. There are four broad classifications of static crosscutting: member introduction, type-hierarchy modification, compile-time error and warning declaration, and exception softening. In this section, we study the first three kinds. Understanding exception softening requires additional design considerations for effective use, and we will visit that along with other similar topics in chapter 4.

#### 3.3.1 Member introduction

Aspects often need to introduce data members and methods into the aspected classes. For example, in a banking system, implementing a minimum balance rule may require additional data members corresponding to a minimum balance and a method for computing the available balance. AspectJ provides a mechanism called *introduction* to introduce such members into the specified classes and interfaces in a crosscutting manner.

The code snippet in listing 3.6 introduces the `_minimumBalance` field and the `getAvailableBalance()` method to the `Account` class. The after advice sets the minimum balance in `SavingsAccount` to 25.

Listing 3.6 `MinimumBalanceRuleAspect.java`

```
public aspect MinimumBalanceRuleAspect {
    private float Account._minimumBalance;
}

public float Account.getAvailableBalance() {
    return getBalance() - _minimumBalance;
}

after(Account account) :
    execution(SavingsAccount.new(..) && this(account) {
        account._minimumBalance = 25;
    }
```

Introducing a data member

Introducing a method

Using the introduced data member

```

before(Account account, float amount)
    throws InsufficientBalanceException :
    execution(* Account.debit()
    && this(account) && args(amount) {
    if (account.getAvailableBalance() < amount) {
        throw new InsufficientBalanceException(
            "Insufficient available balance");
    }
}

```

← Using the introduced method

In the aspect in listing 3.6, we introduce a member `_minimumBalance` of type `float` into the `Account` class. Note that introduced members can be marked with an access specifier, as we have marked `_minimumBalance` with `private` access. The access rules are interpreted with respect to the aspect doing the introduction. For example, the members marked `private` are accessible only from the introducing aspect.

You can also introduce data members and methods with implementation into *interfaces*; this will provide a default behavior to the implementing classes. As long as the introduced behavior suffices for your implementation needs, this prevents the duplication of code in each class, since the introduction of the data members and methods effectively adds the behavior to each implementing class. In chapter 8, we will look more closely at doing this.

### 3.3.2 Modifying the class hierarchy

A crosscutting implementation often needs to affect a set of classes or interfaces that share a common base type so that certain advice and aspects will work only through the API offered by the base type. The advice and aspects will then be dependent only on the base type instead of application-specific classes and interfaces. For example, a cache-management aspect may declare certain classes to implement the `Cacheable` interface. The advice in the aspect then can work only through the `Cacheable` interface. The result of such an arrangement is the decoupling of the aspect from the application-specific class, thus making the aspect more reusable. With AspectJ, you can modify the inheritance hierarchy of existing classes to declare a superclass and interfaces of an existing class or interface as long as it does not violate Java inheritance rules. The forms for such a declaration are:

```
declare parents : [ChildTypePattern] implements [InterfaceList];
```

and

```
declare parents : [ChildTypePattern] extends [Class or InterfaceList];
```

For example, the following aspect declares that all classes and interfaces in the `entities` package that have the `banking` package as the root are to implement the `Identifiable` interface:

```
aspect AccountTrackingAspect {
    declare parents : banking..entities.* implements Identifiable;

    ... tracking advices
}
```

The declaration of parents must follow the regular Java object hierarchy rules. For example, you cannot declare a class to be the parent of an interface. Similarly, you cannot declare parents in such a way that it will result in multiple inheritance.

### 3.3.3 Introducing compile-time errors and warning

AspectJ provides a static crosscutting mechanism to declare compile-time errors and warnings based on certain usage patterns. With this mechanism, you can implement behavior similar to the `#error` and `#warning` preprocessor directives supported by some C/C++ preprocessors, and you can also implement even more complex and powerful directives.

The `declare error` construct provides a way to declare a compile-time error when the compiler detects the presence of a join point matching a given pointcut. The compiler then issues an error, prints the given message for each detected usage, and aborts the compilation process:

```
declare error : <pointcut> : <message>;
```

Similarly, the `declare warning` construct provides a way to declare a compile-time warning, but does not abort the compilation process:

```
declare warning : <pointcut> : <message>;
```

Note that since these declarations affects compile-time behavior, you must use only *statically* determinable pointcuts in the declarations. In other words, the pointcuts that use dynamic context to select the matching join points—`this()`, `target()`, `args()`, `if()`, `cflow()`, and `cflowbelow()`—cannot be used for such a declaration.

A typical use of these constructs is to enforce rules, such as prohibiting calls to certain unsupported methods, or issuing a warning about such calls. The following code example causes the AspectJ compiler to produce a compile-time error if the join point matching the `callToUnsafeCode()` pointcut is found anywhere in the code that is being compiled:

```
declare error : callToUnsafeCode()  
: "This third-party code is known to result in crash";
```

The following code is similar, except it produces a compile-time warning instead of an error:

```
declare warning : callToBlockingOperations()  
: "Please ensure you are not calling this from AWT thread";
```

We have more examples of how to use compile-time errors and warnings for policy enforcement in chapter 6.

### 3.4 Tips and tricks

---

Here are some things to keep in mind as you are learning AspectJ. These simple tips will make your aspects simpler and more efficient:

- *Understand the difference between the AspectJ compiler and a Java compiler*—One of the most common misconceptions that first-time users have is that an AspectJ compiler works just like a Java compiler. However, unlike a Java compiler, which can compile either individual files or a set of files together without any significant difference, the AspectJ compiler must compile all of the related classes and aspects at the same time. This means that you need to pass all the source files to the compiler together. The latest compiler version has additional options for weaving these files into JAR files. With those options, you also need to pass all JAR files together into a single invocation of the compiler. See appendix A for more details.
- *Use a consistent naming convention*—To get the maximum benefit from a wildcard-pointcut, it is important that you follow a naming convention consistently. For example, if you follow the convention of naming all the methods changing the state of an object to start with `set`, then you can capture all the state-change methods using `set*`. A consistent package structure with the right granularity will help capture all the classes inside a package tree.
- *Use after returning when appropriate*—When designing the after advice, consider using after returning instead of after, as long as you don't need to capture an exception-throwing case. The implementation for the after advice without returning needs to use a try/catch block. There is a cost associated with such a try/catch block that you can avoid by using an after returning advice.
- *Don't be misled by &&*—The natural language reading of pointcuts using `&&` often misleads developers who are new to AspectJ. For example, the point-

cut `publicMethods() && privateMethods()` won't match any method even though the natural reading would suggest “public *and* private methods.” This is because a method can have either private access or public access, but not both. The solution is simple: use `||` instead to match public *or* private methods.

Chapter 8 presents a set of idioms that will help you avoid potential troubles as you begin using AspectJ.

### 3.5 Summary

---

AspectJ introduces AOP programming to Java by adding constructs to support dynamic and static crosscutting. Dynamic crosscutting modifies the behavior of the modules, while static crosscutting modifies the structure of the modules. Dynamic crosscutting consists of pointcut and advice constructs. AspectJ exposes the join points in a system through pointcuts. The support of wildcard matching in pointcuts offers a powerful yet simple way to capture join points without knowing the full details. The advice constructs provide a way to express actions at the desired join points. Static crosscutting, which can be used alone or in support of dynamic crosscutting, includes the constructs of member introduction, type hierarchy modification, and compile-time declarations. The overall result is a simple and programmer-friendly language supporting AOP in Java. At this point, if you haven't already done so, you may want to download and install the AspectJ compiler and tools. Appendix A explains where to find the compiler and how to install it.

Together, this chapter and the previous one should get you started on AspectJ, but for complex programs, you will need to learn a few more concepts, such as exception softening and aspect association. We present these concepts and more in the next chapter.



# AspectJ IN ACTION



AspectJ  
v1.1

## Practical Aspect-Oriented Programming

Ramnivas Laddad

**M**odularizing code into objects cannot be fully achieved in pure OOP. In practice some objects must deal with aspects that are not their main business. A method to modularize aspects—and benefit from a clean maintainable result—is called aspect-oriented programming. AspectJ is an open-source Java extension and compiler designed for AOP development. Now integrated with Eclipse, NetBeans, JBuilder, and other IDEs, AspectJ v1.1 is ready for the real world.

It is time to move from AOP theory and toy examples to AOP practice and real applications. With this unique book you can make that move. It teaches you AOP concepts, the AspectJ language, and how to develop industrial-strength systems. It shows you examples which you can reuse. It unleashes the true power of AOP through unique *patterns* of AOP design. When you are done, you will be eager—and able—to build new systems, and enhance your existing ones, with the help of AOP.

### What's Inside

- What is aspect-oriented programming?
- How AspectJ works with JAAS, Jess, log4j, Ant, JTA, POJOs
- Best practices and design patterns **NEW**
- How to implement
  - policy enforcement
  - resource pooling and caching
  - thread-safety
  - authentication and authorization
  - transaction management
  - business rules **NEW**

**Ramnivas Laddad** is an AOP and AspectJ authority. With his writings, he has contributed to the general awareness of AOP and has contributed to features now incorporated in AspectJ Version 1.1. Ramnivas lives in Sunnyvale, California.

“Speaks directly to me as a developer”

—Alan Cameron Wills  
fastnloose

“... real solutions to tough problems ...”

—Chris Bartling, Identix, Inc.

“I started reading at 11 PM and couldn't stop ... It's a must read for anyone interested in the future of programming.”

—Arno Schmidmeier  
AspectSoft

“... The only resource that presents AOP concepts and real-world examples in an approachable, readable way.”

—Jean Baltus  
Metafro-Infosys

[www.manning.com/laddad](http://www.manning.com/laddad)

**AUTHOR  
ONLINE**

Author responds to reader questions



Ebook edition available



9 781930 110939

5 4 4 9 5

ISBN 1-930110-93-6

 **MANNING**

\$44.95 US/\$67.95 Canada

# AspectJ

---

# IN ACTION

Practical Aspect-Oriented Programming

Ramnivas Laddad



 MANNING

For online information and ordering of this and other Manning books, go to [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department  
Manning Publications Co.

209 Bruce Park Avenue  
Greenwich, CT 06830

Fax: (203) 661-9018

email: [orders@manning.com](mailto:orders@manning.com)

©2003 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.



Manning Publications Co.  
209 Bruce Park Avenue  
Greenwich, CT 06830

Copyeditor: Liz Welch  
Typesetter: Denis Dalinnik  
Cover designer: Leslie Haines

ISBN 1-930110-93-6

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – VHG – 05 04 03 02

# *brief contents*

---

## **PART 1 UNDERSTANDING AOP AND ASPECTJ..... 1**

- 1 ■ Introduction to AOP 3
- 2 ■ Introducing AspectJ 32
- 3 ■ AspectJ: syntax basics 64
- 4 ■ Advanced AspectJ 100

## **PART 2 BASIC APPLICATIONS OF ASPECTJ.....143**

- 5 ■ Monitoring techniques: logging, tracing, and profiling 145
- 6 ■ Policy enforcement: system wide contracts 178
- 7 ■ Optimization: pooling and caching 202

## **PART 3 ADVANCED APPLICATIONS OF ASPECTJ.....243**

- 8 ■ Design patterns and idioms 245
- 9 ■ Implementing thread safety 286
- 10 ■ Authentication and authorization 323

- 11 ■ Transaction management 356
- 12 ■ Implementing business rules 391
- 13 ■ The next step 425
- A ■ The AspectJ compiler 438
- B ■ Understanding Ant integration 447
  - resources 455
  - index 461

# 10

## *Authentication and authorization*

---

### ***This chapter covers***

- Using JAAS to implement authentication and authorization
- Using AspectJ to modularize JAAS-based authentication
- Using AspectJ to modularize JAAS-based authorization

An important consideration for modern software systems, security consists of many components, including authentication, authorization, auditing, protection against web site attacks, and cryptography. In this chapter, we focus on two of these: authentication and authorization. Together these security components manage system access by evaluating users' identities and credentials.

This chapter introduces an AspectJ-based solution using the Java Authentication and Authorization Service (JAAS), one of the newest ways to implement authentication and authorization in Java applications. You'll see how AspectJ-based solutions work in cooperation—and not in competition—with existing technologies. Using AspectJ helps you to modularize your implementation, which leads to better response to requirement changes, while at the same time greatly reducing the amount of code you have to write.

To get a clear understanding of the core problem and how you'd use JAAS to address it, we also examine the conventional solution for implementing authentication and authorization. Developing the conventional solution serves two purposes: it introduces the basic mechanism offered by JAAS and it demonstrates its shortcomings. Later when we present the AspectJ-based solution, this knowledge will come in handy.

## **10.1 Problem overview**

---

*Authentication* is a process that verifies that you are who you say you are. *Authorization*, on the other hand, is a process that establishes whether an authenticated user has sufficient permissions to access certain resources. Both components are so closely related that it is difficult to talk about one without the other—authorization cannot be accomplished without first performing authentication, and authentication alone is rarely sufficient to determine access to resources.

Since authentication and authorization are so important—and continue to become even more so given our highly connected world—we must learn to deal with the various ways of implementing such control. Modern APIs like JAAS (which is now a standard part of J2SE 1.4) abstract the underlying mechanisms and allow you to separate the access control configuration from the code. The application-level developer doesn't have to be aware of the underlying mechanism and won't need to make any invasive changes when it changes. In parallel to these APIs, efforts such as the Security Assertion Markup Language (SAML) and the Extensible Access Control Markup Language (XACML) aim to standardize the configuration specification language. The overall goal of these APIs and standardization efforts is to reduce complexity and provide agile implementations.

Conventional programming methods, even when using APIs such as JAAS, require you to modify multiple modules individually to equip them with authentication and authorization code. For instance, to implement access control in a banking system, you must add calls to JAAS methods to all the business methods. As the business logic is spread over multiple modules, so too is the implementation of the access control logic.

Unlike the bare OOP solution, an EJB framework handles authorization in a much more modular way, separating the security attributes in the deployment descriptor. As we mentioned in chapter 1, the very existence of EJB is proof that we need to modularize such concerns. When EJB or a similar framework is not a choice, as in a UI program, the solution often lacks the desired modularization. With AspectJ, you now have a much better solution for all such situations.

---

**NOTE** Even with the EJB framework, you may face situations that need a custom solution for authentication and authorization. Consider, for example, data-driven authorization where the authorization check not only considers the identity of the user and the functionality being accessed, but also the data involved. Current EJB frameworks do not offer a good solution to these problems that demand flexibility.

## 10.2 A simple banking example

---

To illustrate the problem and provide a test bed, let's write a simple banking system. We'll examine only the parts of the system that illustrate issues involved in conventional and AspectJ-based solutions to authentication and authorization implementation. The banking example here differs from the one in chapter 2 in a few ways: We refactor the classes to create interfaces, we put all the classes and interfaces in the `banking` package, and we introduce a new class. We will continue to build on this system in the next two chapters.

Listing 10.1 shows the `Account` interface. (As you can see, we have omitted some of the methods that you would expect to see in an `Account` interface.) Later we'll create a simple implementation of this interface. The exception `InsufficientBalanceException` that we'll use to identify an insufficient balance is implemented in listing 10.2.

**Listing 10.1** `Account.java`

```
package banking;

public interface Account {
    public int getAccountNumber();
}
```



```
    public void credit(float amount);

    public void debit(float amount)
        throws InsufficientBalanceException;

    public float getBalance();
}
```

**Listing 10.2** `InsufficientBalanceException.java`

```
package banking;

public class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
```

Now, let's look at a simple, bare-bones implementation of the Account interface. Later, we'll pose the problem of authorizing all of its methods, using both conventional and AspectJ-based solutions. Listing 10.3 shows a simple implementation of the Account interface that models a banking account.

**Listing 10.3** `AccountSimpleImpl.java`

```
package banking;

public class AccountSimpleImpl implements Account {
    private int _accountNumber;
    private float _balance;

    public AccountSimpleImpl(int accountNumber) {
        _accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return _accountNumber;
    }

    public void credit(float amount) {
        _balance = _balance + amount;
    }

    public void debit(float amount)
        throws InsufficientBalanceException {
        if (_balance < amount) {
            throw new InsufficientBalanceException(
                "Total balance not sufficient");
        }
    }
}
```

```
        } else {
            _balance = _balance - amount;
        }
    }

    public float getBalance() {
        return _balance;
    }
}
```

The code for `AccountSimpleImpl` is straightforward. To examine how our solution works across multiple modules and with nested methods that need authorization, let's introduce another class, `InterAccountTransferSystem` (listing 10.4), which simply contains one method for transferring funds from one account to another.

#### Listing 10.4 `InterAccountTransferSystem.java`

```
package banking;

public class InterAccountTransferSystem {
    public static void transfer(Account from, Account to,
                               float amount)
        throws InsufficientBalanceException {
        to.credit(amount);
        from.debit(amount);
    }
}
```

Finally, to test our solution we'll write a simple `Test` class. In the sections that follow, we will use this class as a basis for adding authentication and authorization in the conventional way; later in the chapter, we will use the class to test the AspectJ-based solution. Listing 10.5 shows the implementation of the `Test` class.

#### Listing 10.5 `Test.java`: version with no authentication or authorization

```
package banking;

public class Test {
    public static void main(String[] args) throws Exception {
        Account account1 = new AccountSimpleImpl(1);
        Account account2 = new AccountSimpleImpl(2);

        account1.credit(300);
        account1.debit(200);
    }
}
```

```
        InterAccountTransferSystem.transfer(account1, account2, 100);  
        InterAccountTransferSystem.transfer(account1, account2, 100);  
    }  
}
```

Because of the way the operations are arranged, the last operation should throw an `InsufficientBalanceException`. We will ensure that our solutions satisfy the requirement of throwing this exception (as opposed to some other type of exception or no exception at all) when the business logic detects insufficient funds in the debiting account.

Next, let's implement a basic logging aspect (listing 10.6) to help us understand the activities taking place.

**Listing 10.6 AuthLogging.java: logging banking operations**

```
package banking;  
  
import org.aspectj.lang.*;  
  
import logging.*;  
  
public aspect AuthLogging extends IndentedLogging {  
    declare precedence: AuthLogging, *;  
  
    public pointcut accountActivities()  
        : execution(public * Account.*(..))  
        || execution(public * InterAccountTransferSystem.*(..));  
  
    public pointcut loggedOperations()  
        : accountActivities();  
  
    before() : loggedOperations() {  
        Signature sig = thisJoinPointStaticPart.getSignature();  
        System.out.println("<" + sig.getName() + ">");  
    }  
}
```

The base aspect, `IndentedLogging`, was discussed in section 5.5.2. It provides the support for indenting the log statements according to their call depth. We need to define the `loggedOperation()` pointcut that was declared in the base `IndentedLogging` aspect. Later, we will add authentication and authorization logging to it as we develop the solution. We won't log more details about the activities (such as account number and amount involved), since the correctness of the core implementation is not the focus of this chapter.

When we compile the basic banking application and the logging aspect, and then run the test program, we see output similar to this:

```
> ajc banking\*.java logging\*.java
> java banking.Test
<credit>
<debit>
<transfer>
  <credit>
  <debit>
<transfer>
  <credit>
  <debit>
Exception in thread "main" banking.InsufficientBalanceException:
  ➡ Total balance not sufficient
...more call stack
```

The output shows the interaction when no authentication or authorization is in place. This interaction log will serve as the basis for comparison when we add authentication and authorization.

Coverage of the JAAS mechanism is brief since our purpose is to demonstrate the AOP solution. We encourage you to read a good JAAS book or tutorial so that you will understand the more complex issues that we do not deal with here; then you can extend the AspectJ-based solution to them as well. Please note that although we use a JAAS-based example to explain the AspectJ-based solution, you can also use the solution as a template for other kinds of access control systems.

## 10.3 Authentication: the conventional way

---

In this section, we add authentication functionality to our basic banking system. We employ the upfront login approach—asking for the username and password at the beginning of the program. Because of its complexity, we won't look at an example of just-in-time authentication (in which authentication does not occur until the user accesses the system functionality that requires user identity verification) in this section, since the point we are demonstrating is basically the same.

### 10.3.1 Implementing the solution

The authentication functionality in JAAS consists of the following:

- A `LoginContext` object
- Callback handlers that present the login challenge to the user
- A login configuration file that enables you to modify the configuration without changing the source code

The callback handler provides a mechanism for acquiring authentication information. It asks users to provide their name and password either on the console, in a login dialog box, or through some other means. In our case, we use a simple `TextCallbackHandler` that is part of Sun's JRE 1.4 distribution. If you are using another JRE, this class may not be available, and you will have to either find an equivalent or write one of your own. `TextCallbackHandler`, when invoked, simply asks for the username and password and supplies the information to the authentication system invoking it. Since the username and password are visible to the user, you are unlikely to use this callback handler in a real system, but it serves as a simple, illustrative mechanism for our purposes.

---

**NOTE** We use the term *user* to mean anyone and anything accessing the system. It includes human as well as nonhuman users—people and other parts of the system. For example, in a business-to-business transaction, a machine is likely to represent the identity of a business accessing the service.

---

The login configuration file sets up the class that is used as the authentication module. We use a very simple authentication module, `sample.module.SampleLoginModule`, provided as a part of the JAAS tutorial (see <http://java.sun.com/j2se/1.4/docs/guide/security/jaas/tutorials/GeneralAcnAndAzr.html>). The classes from the `sample` package we use are described in the tutorial. Employing this simple scheme allows us to focus on using AOP instead of the details of JAAS. The following login configuration file (`sample_jaas.config`) associates the `Sample` configuration with the `sample.module.SampleLoginModule` class:

```
Sample {  
    sample.module.SampleLoginModule required debug=true;  
};
```

The `LoginContext` object needs two parameters: a configuration name and a callback handler. The configuration name (`Sample`), in conjunction with the configuration file, determines the login module used by the system.

Let's change the `Test` class to implement authentication with JAAS in the conventional way, as shown in listing 10.7.

#### Listing 10.7 Test.java: with authentication functionality

```
package banking;  
  
import javax.security.auth.login.LoginContext;  
  
import com.sun.security.auth.callback.TextCallbackHandler;
```

```
public class Test {
    public static void main(String[] args) throws Exception {
        LoginContext lc
            = new LoginContext("Sample",
                               new TextCallbackHandler());
        lc.login();

        Account account1 = new AccountSimpleImpl(1);
        Account account2 = new AccountSimpleImpl(2);

        account1.credit(300);
        account1.debit(200);

        InterAccountTransferSystem.transfer(account1, account2, 100);
        InterAccountTransferSystem.transfer(account1, account2, 100);
    }
}
```

We enable authentication in our banking system by performing login before executing any core code. First, we create a `LoginContext` object, supplying it with the name of the configuration we wish to use and the callback handler that will request the username and password. Next, we invoke the `login()` method on the `LoginContext` object. If the username and password pass the authentication test, the method simply returns normally. If, however, the username and password fail to match, it throws a checked exception of type `LoginException`. Once the authentication is passed successfully, we continue with the main program functionality.

Since we have chosen to implement upfront login authentication, this arrangement will satisfy that requirement. If, however, you want just-in-time authentication, you will need to add similar authentication coding in every such operation. Just-in-time authentication is useful when the system contains several parts that do not require authenticating the user. Pre-authenticating users may be less than desirable in such cases.

### 10.3.2 Testing the solution

To examine the interaction, let's improve the logging aspect for capturing the authentication join points. We will change the pointcuts to log the login join points, as shown in listing 10.8. In the section that follows, we will use the same logging aspect when we test our AspectJ-based solution.

**Listing 10.8 AuthLogging.java: with authentication logging implemented**

```

package banking;

import org.aspectj.lang.*;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;

import logging.*;

public aspect AuthLogging extends IndentedLogging {
    declare precedence: AuthLogging, *;

    public pointcut accountActivities()
        : execution(public * Account.*(..))
        || execution(public * InterAccountTransferSystem.*(..));

    public pointcut authenticationActivities()
        : call(* LoginContext.login(..));

    public pointcut loggedOperations()
        : accountActivities()
        || authenticationActivities();

    before() : loggedOperations() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        System.out.println("<" + sig.getName() + ">");
    }
}

```

When we run the program, it asks for a username and password. If the user can be authenticated, it proceeds with the remaining part of the program. Otherwise, it throws a `LoginException`:

```

> ajc banking/*.java logging/*.java
  ➤ sample/module/*.java sample/principal/*.java
> java -Djava.security.auth.login.config=sample_jaas.config
  ➤ banking.Test
<login>
user name: testUser
password: testPassword
           [SampleLoginModule] user entered user name: testUser
           [SampleLoginModule] user entered password: testPassword
           [SampleLoginModule] authentication succeeded
           [SampleLoginModule] added SamplePrincipal to Subject

<credit>
<debit>
<transfer>

```





```

before() : authOperations() {
    if(_authenticatedSubject != null) {
        return;
    }

    try {
        authenticate();
    } catch (LoginException ex) {
        throw new AuthenticationException(ex);
    }
}

private void authenticate() throws LoginException {
    LoginContext lc = new LoginContext("Sample",
                                     new TextCallbackHandler());
    lc.login();
    _authenticatedSubject = lc.getSubject();
}

public static class AuthenticationException
    extends RuntimeException {
    public AuthenticationException(Exception cause) {
        super(cause);
    }
}
}

```

**3 Authentication advice**

**4 Authentication logic**

**5 Authentication exception**

- ❶ The aspect stores the authenticated subject in an instance variable. By storing the authenticated subject and checking for it prior to invoking the login logic, we avoid asking for a login every time a method that needs authentication is called. After a successful login operation, we can obtain this member from the LoginContext object.

In our implementation, we will use the whole process as the login scope. Once a user is logged in, he will never have to log in again during the lifetime of the program. Depending on your system's specific requirements, you may want to move this member to an appropriate place. For example, if you are writing a servlet, you may want to keep this member in the session object. We also assume that a user, once logged in, never logs out. If this is not true in your system, you need to set this member to null when the current user logs out.

- ❷ The abstract pointcut is meant to be defined in subspects capturing all the operations needing authentication.
- ❸ The before advice to the authOperations() pointcut ensures that our code performs authentication logic only if this is the first time during the program's lifetime that a method that needs authentication is being executed. If it is the first

time, `_authenticatedSubject` will be null, and the `authenticate()` method will be invoked to perform the core authentication logic. When subsequent join points that need authentication are executed, because the `_authenticatedSubject` is already not null the login process won't be carried out.

Since the `LoginException` is a checked exception, the before advice cannot throw it. Throwing such exceptions would result in compiler errors. We could have simply softened this exception using the `declare soft` construct. However, following the exception introduction pattern discussed in chapter 8, we instead define a concern-specific runtime exception that identifies the cause of the exception, should a caller wish to handle the exception.

- ④ The core authentication operation is performed in this method. If the login fails, it throws a `LoginException` that aborts the program. If the login succeeds, it obtains the subject from the login context and sets it to the instance variable `_authenticatedSubject`.
- ⑤ `AuthenticationException` is simply a `RuntimeException` that wraps the original exception.

Adding authentication functionality to banking is now a simple matter of writing an aspect, as shown in listing 10.10, that extends `AbstractAuthAspect` and defines the `authOperations()` pointcut. In our example, we define the pointcut to capture calls to all methods in the `Account` and `InterAccountTransferSystem` classes.

**Listing 10.10** `BankingAuthAspect.java`: authenticating banking operations

```
package banking;

import auth.AbstractAuthAspect;

public aspect BankingAuthAspect extends AbstractAuthAspect {
    public pointcut authOperations()
        : execution(public * banking.Account.*(..))
        || execution(public * banking.InterAccountTransferSystem.*(..));
}
```

Although we have used just-in-time authentication in this example, you can easily implement up-front authentication by simply adding a pointcut corresponding to the method that represents “up-front” for you, such as the `main()` method in the console application or the frame initialization in a UI application. For example, defining the `authOperations()` pointcut as follows will perform authentication as soon as the `main()` method begins to execute:

```
public pointcut authOperations()
    : execution(void banking.Test.main(String[]));
```

With such a pointcut, the authentication advice will kick in as soon as the program starts entering the `main()` method. Further, when you choose up-front authentication, you can write an additional advice that tests for authentication status before executing a method that needs authenticated access. This advice could simply throw a runtime exception, because accessing this method without prior authentication is a violation.

### 10.4.2 Testing the solution

We now have the system equipped with authentication. When we compile the new aspects with the classes and interfaces in section 10.2, along with the logging aspect in listing 10.8, and run the test program, it prompts for a username and password, as in the conventional solution developed earlier:

```
> ajc banking\*.java auth\*.java logging\*.java
  ➤ sample\module\*.java sample\principal\*.java
> java -Djava.security.auth.login.config=sample_jaas.config
  ➤ banking.Test
<credit>
  <login>
user name: testUser
password: testPassword
           [SampleLoginModule] user entered user name: testUser
           [SampleLoginModule] user entered password: testPassword
           [SampleLoginModule] authentication succeeded
           [SampleLoginModule] added SamplePrincipal to Subject

<debit>
<transfer>
  <credit>
  <debit>
<transfer>
  <credit>
  <debit>
Exception in thread "main" banking.InsufficientBalanceException:
  ➤ Total balance not sufficient
  ... the rest of call stack
```

As expected, this output is identical to that shown in section 10.3. We now have a system with authentication modularized in one reusable abstract aspect and one system-specific concrete aspect.

## 10.5 Authorization: the conventional way

The authorization process determines whether the user has sufficient credentials to access certain functions within the system. Let's consider a banking system

where the authorization rule specifies that only users with managerial credentials may waive certain fees. We need to perform the following operations:

- 1 Authentication is a prerequisite to authorization; unless we are certain that users are who they claim to be, there is no point in checking their credentials. Therefore, we first need to verify that users have been authenticated, and if they have not, we need to do so.
- 2 Then we need to retrieve users' credentials. You can do this in various ways depending on the authorization scheme you use. For example, the authorization system could check a policy file to extract the credentials associated with the authorized person.
- 3 Last, we need to verify whether those credentials are sufficient to access the fee-waiving operation. For example, if a person has only the teller credential and not the managerial credential, fee-waiving operations won't be available to that user.

### 10.5.1 Understanding JAAS-based authorization

While the exact way you use JAAS will depend on your system's access control requirements, a typical way to use it to perform authorization requires that you follow these steps:

- 1 *Perform authentication*—The system first needs to authenticate the user using a login or any suitable mechanism. Then it must obtain a verified subject from the authentication subsystem. The `Subject` class encapsulates information about a single entity, such as its identification and credentials. All subsequent operations that require authorization must check that this subject has sufficient credentials to access the operations.
- 2 *Create an action object*—JAAS requires that each method that needs an authorization check be encapsulated in an *action object*. This object must implement either `PrivilegedAction` or `PrivilegedExceptionAction`. Both interfaces contain just one method: `run()`. The only difference is that the `run()` method has no exception declaration in the former interface, whereas in the latter, it declares that it may throw an exception of type `Exception`. In either case, the `run()` method needs to execute the intended operation.
- 3 *Execute the action object*—The action object we just created needs to be executed on behalf of the authenticated subject using static methods in

the Subject class: `Subject.doAsPrivileged(Subject, PrivilegedAction, AccessControlContext)` or `Subject.doAsPrivileged(Subject, PrivilegedExceptionAction, AccessControlContext)`. In cases where `doAsPrivileged()` is called with a `PrivilegedExceptionAction` parameter, if the `run()` method throws a checked exception, it will wrap it inside `PrivilegedActionException` before throwing it.

- 4 *Check access*—The methods that need to ensure authorized access must check the subject’s credentials by calling the `AccessController.checkPermission()` method and passing it a permission object that contains the required permissions. If the user doesn’t have sufficient permissions, this method throws an unchecked `AccessControlException` exception.
- 5 *Create a system-level access control policy*—At the system level, you write a policy file that grants to a set of subjects permissions to certain operations. The `AccessController.checkPermission()` method indirectly uses this policy file to grant access only to those operations that are allowed by the accessing subject’s credentials and permissions.

### 10.5.2 Developing the solution

Now that we’ve looked at the changes needed in the system to implement authorization, let’s look at the modifications we need to make in the banking example. In listing 10.11, we define a simple permission class, `BankingPermission`. The name string passed in its constructor defines the permissions. We will later map these strings in a security policy file to allow only certain users to access certain functionality.

**Listing 10.11** `BankingPermission.java`: permission class for banking system authorization

```
package banking;

import java.security.*;

public final class BankingPermission extends BasicPermission {
    public BankingPermission(String name) {
        super(name);
    }

    public BankingPermission(String name, String actions) {
        super(name, actions);
    }
}
```

The class `BankingPermission` defines two constructors to match those in the base `BasicPermission` class. The `actions` parameter in the second constructor is unused and exists only to instantiate the permission object from a policy file. To learn more, refer to the JDK documentation.

Now let's modify the `AccountSimpleImpl` class to check permission in each of its public methods. Each change is simply a call to `AccessController.checkPermission()` with a `BankingPermission` object as an argument. Each `BankingPermission` needs a name argument to specify the kind of permission sought. We employ a simple scheme that uses the method name itself as the permission string. Listing 10.12 shows the implementation of `AccountSimpleImpl` where each method checks the permission before executing its core logic.

**Listing 10.12** `AccountSimpleImpl.java`: the conventional way

```
package banking;

import java.security.AccessController;

public class AccountSimpleImpl implements Account {
    private int _accountNumber;
    private float _balance;

    public AccountSimpleImpl(int accountNumber) {
        _accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        AccessController.checkPermission(
            new BankingPermission("getAccountNumber"));
        ...
    }

    public void credit(float amount) {
        AccessController.checkPermission(
            new BankingPermission("credit"));
        ...
    }

    public void debit(float amount)
        throws InsufficientBalanceException {
        AccessController.checkPermission(
            new BankingPermission("debit"));
        ...
    }
}
```



```
        public Object run() throws Exception {
            account1.debit(200);
            return null;
        }}, null);
    } catch (PrivilegedActionException ex) {
        Throwable cause = ex.getCause();
        if (cause instanceof InsufficientBalanceException) {
            throw (InsufficientBalanceException)ex.getCause();
        }
    }
}

try {
    Subject
        .doAsPrivileged(authenticatedSubject,
            new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    InterAccountTransferSystem
                        .transfer(account1, account2,
                            100);
                    return null;
                }}, null);
    } catch (PrivilegedActionException ex) {
        Throwable cause = ex.getCause();
        if (cause instanceof InsufficientBalanceException) {
            throw (InsufficientBalanceException)ex.getCause();
        }
    }
}

try {
    Subject
        .doAsPrivileged(authenticatedSubject,
            new PrivilegedExceptionAction() {
                public Object run() throws Exception {
                    InterAccountTransferSystem
                        .transfer(account1, account2,
                            100);
                    return null;
                }}, null);
    } catch (PrivilegedActionException ex) {
        Throwable cause = ex.getCause();
        if (cause instanceof InsufficientBalanceException) {
            throw (InsufficientBalanceException)ex.getCause();
        }
    }
}
}
```

---

Clearly, we've had to use too much code. For each operation needing access control, we create an anonymous class extending either `PrivilegedExceptionAction`



or `PrivilegedAction`, based on whether the operation can throw a checked exception. The `run()` method of each anonymous class simply calls the operation under consideration.

We put the calls to the methods that are routed through a `PrivilegedExceptionAction` object in a try/catch block. In the catch block, we check to see if the cause for the exception is an `InsufficientBalanceException`. If so, we throw that exception because the caller of the business method would expect it to be `InsufficientBalanceException` and not `PrivilegedExceptionAction`. Please refer to the JDK documentation for `PrivilegedExceptionAction` for more details on how the checked exceptions are handled differently than the runtime exceptions.

While we use anonymous classes here, we could have used named classes as well. Each named class would require a constructor taking all the parameters of the method. It would then store those parameters as instance variables. Later, while implementing the `run()` method, it would pass the stored instance variables to the method.

We could have also combined all the operations into one action by creating a single `PrivilegedExceptionAction` and routing all the actions through it. However, we did not do so in order to better mimic the real system, where not all the operations that need authorization will be in one or two places. Further, combining several methods into one action requires that you consider exception-handling carefully. By routing the methods individually through the `PrivilegedExceptionAction` class, you can handle an exception thrown by each method separately and make the appropriate decisions. With the combined method, you will need to handle the exceptions thrown by a set of methods together. While such an arrangement may not always be a problem, you need to consider it anyway.

### 10.5.3 Testing the solution

Let's see if the solution works. To do so, we add authorization logging to the `AuthLogging` aspect, as shown in listing 10.14.

**Listing 10.14** `AuthLogging.java`: adding authorization logging

```
package banking;

import org.aspectj.lang.*;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;

import logging.*;

public aspect AuthLogging extends IndentedLogging {
```

```
declare precedence: AuthLogging, *;

public pointcut accountActivities()
    : call(void Account.credit(..))
    || call(void Account.debit(..))
    || call(* Account.getBalance(..))
    || call(void InterAccountTransferSystem.transfer(..));

public pointcut authenticationActivities()
    : call(* LoginContext.login(..));

public pointcut authorizationActivities()
    : call(* Subject.doAsPrivileged(..));

public pointcut loggedOperations()
    : accountActivities()
    || authenticationActivities()
    || authorizationActivities();

before() : loggedOperations() {
    Signature sig = thisJoinPointStaticPart.getSignature();
    System.out.println("<" + sig.getName() + ">");
}
}
```

The aspect in listing 10.14 modified the one in listing 10.8 to add a new pointcut, `authorizationActivities()`, and include that pointcut in the `loggedOperation()` pointcut.

In the `BankingPermission` class (listing 10.11), the constructor took an argument name that was a string defining the permissions for the system. We said that we would later map name to a security policy file to allow only certain users to access certain functionality. Let's define that security policy file now. We want to permit `testUser` to be able to carry out all the operations in the banking system. Listing 10.15 shows the policy file that grants `testUser` the permissions to access all the operations (`credit`, `debit`, `getBalance`, and `transfer`).

#### Listing 10.15 security.policy: the policy file for authorization

```
grant Principal sample.principal.SamplePrincipal "testUser" {
    permission banking.BankingPermission "credit";
    permission banking.BankingPermission "debit";
    permission banking.BankingPermission "getBalance";
    permission banking.BankingPermission "transfer";
};
```

When we compile and run the test program, it not only asks for a name and password, but also executes all the operations that have been authorized through `Subject.doAsPrivileged()`:

```
> ajc banking\*.java logging\*.java
  ➤ sample\module\*.java sample\principal\*.java
> java -Djava.security.auth.login.config=sample_jaas.config
  ➤ -Djava.security.policy=security.policy banking.Test
<login>
user name: testUser
password: testPassword
           [SampleLoginModule] user entered user name: testUser
           [SampleLoginModule] user entered password: testPassword
           [SampleLoginModule] authentication succeeded
           [SampleLoginModule] added SamplePrincipal to Subject

<doAsPrivileged>
  <credit>
<doAsPrivileged>
  <debit>
<doAsPrivileged>
  <transfer>
    <credit>
    <debit>
<doAsPrivileged>
  <transfer>
    <credit>
    <debit>
Exception in thread "main" banking.InsufficientBalanceException:
  ➤ Total balance not sufficient
... the rest of call stack
```

The output shows that each method that needs authorization is called in the context of the `doAsPrivileged()` method. We will compare this output to one using AspectJ-based authorization in section 10.6; we expect them to be identical.

If you want to learn more about JAAS, modify the security policy file to see the effect of different permissions. This will allow you to see how JAAS prevents certain users from accessing a set of operations while allowing others to access those operations.

Now extend this problem to a real system and try to answer the following question: Which operations in your system need to be authenticated/authorized? The answer will not be easy to come by. You will have to examine all the modules and create a list of operations that perform access control checks. This task is laborious and error-prone.

### 10.5.4 Issues with the conventional solution

Let's summarize the problems posed by the conventional object-oriented solution:

- *Scattering of decisions*—The decision for operations to be checked against permissions is scattered throughout the system, and therefore any modifications to it will cause invasive changes.
- *Difficulty of determining access-controlled operations*—Consider the same problem of deciding if an operation needs to perform authorization checks from the business component developer's point of view. Since deciding whether an operation needs authorization depends on the system using the components, it is even harder to identify these operations in components than in system-specific classes.
- *The need to write a class for each access-controlled operation*—For each simple operation, you must write a named or anonymous class carrying out the desired operation.
- *Incoherent system behavior*—The implementation for authorizing a method is separated into two parts: the *callee* and the *caller*. The callee side uses `AccessController.checkPermission()` to check the permissions (as in listing 10.12), whereas the caller side uses `Subject.doAsPrivileged()` to execute the operation on a subject's behalf. Failure to check permissions on the callee side may allow unauthorized subjects to access your system. On the caller side, if you forget to use `Subject.doAsPrivileged()`, your operation will fail even if the user accessing the operation has the proper set of permissions. If you don't find and fix the problem during a code review or a testing phase, it will pop up after the deployment, potentially causing a major loss of business functionality.
- *Difficult evolution*—Any change in authorization operations means making changes in every place the call is made. Any such change will require that the entire test be run through again, increasing the cost of the change.

This list demonstrates the sheer amount of code you will need to write. However, the amount of code is not the biggest problem. Just examine the tangling of the authorization code—it simply overwhelms the core logic. The conventional methods force you to stuff the system-level authorization concern into every part of the system. A utility wrapper can reduce the amount of code, but the fundamental problem of tangling remains unsolved.

## 10.6 Authorization: the AspectJ way

---

In extending the AspectJ solution to address authorization, we use the worker object creation pattern described in chapter 8. As with authentication, AspectJ enables you to add authorization to the system without changing the core implementation. In this section, we develop a reusable aspect that enables you to add authorization to your system by simply writing a few lines for a subaspect.

### 10.6.1 Developing the solution

To recap, using JAAS to implement authorization involves routing the authorized call through a class that implements either `PrivilegedExceptionAction` or `PrivilegedAction`, depending on whether the operation throws checked exceptions. As you saw in section 10.5, the conventional solution requires the coding of both classes implementing `PrivilegedAction` and their invocations. The worker object creation pattern takes the pain out of this process. Without this pattern, we would have to implement classes for each operation that needs authorization. We could still use AspectJ to provide around advice to intercept each of the operations *separately* and to create and execute the corresponding, hand-written action objects through `Subject.doAsPrivileged(Subject, PrivilegedAction, AccessControlContext)`, or `Subject.doAsPrivileged(Subject, PrivilegedExceptionAction, AccessControlContext)`. Now, with the use of a worker object creation pattern, instead of writing a class for each operation that needs authorization, we simply write an aspect that advises all corresponding join points of such operations to auto-create worker classes and execute them through `Subject.doAsPrivileged()`.

The result is a real savings in the amount of code we have to write, since the concern is modularized within just one aspect. Listing 10.16 shows the base aspect that implements the authorization concern in addition to authentication.

**Listing 10.16** `AbstractAuthAspect.java`: adding authorization capabilities

```
package auth;

import org.aspectj.lang.JoinPoint;

import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.*;

import com.sun.security.auth.callback.TextCallbackHandler;

public abstract aspect AbstractAuthAspect {
```

```

private Subject _authenticatedSubject;

public abstract pointcut authOperations(); ← ❶ Pointcut for operations that need authorization

before() : authOperations() {
    if(_authenticatedSubject != null) {
        return;
    }

    try {
        authenticate();
    } catch (LoginException ex) {
        throw new AuthenticationException(ex);
    }
}

public abstract Permission getPermission(
    JoinPoint.StaticPart joinPointStaticPart); ❷ Method that obtains the needed permissions

Object around()
: authOperations() && !cflowbelow(authOperations()) { ❸ Around advice that creates and executes the worker object
    try {
        return Subject
            .doAsPrivileged(_authenticatedSubject,
                new PrivilegedExceptionAction() {
                    public Object run() throws Exception {
                        return proceed();
                    }
                }, null);
    } catch (PrivilegedActionException ex) {
        throw new AuthorizationException(ex.getException());
    }
}

before() : authOperations() {
    AccessController.checkPermission(
        getPermission(thisJoinPointStaticPart));
}

private void authenticate() throws LoginException {
    LoginContext lc = new LoginContext("Sample",
        new TextCallbackHandler());

    lc.login();
    _authenticatedSubject = lc.getSubject();
}

public static class AuthenticationException
    extends RuntimeException {
    public AuthenticationException(Exception cause) {
        super(cause);
    }
}

```

```
public static class AuthorizationException
    extends RuntimeException {
    public AuthorizationException(Exception cause) {
        super(cause);
    }
}
```

5 Authorization exception

This aspect routes every call that needs authorization through an anonymous class implementing the `PrivilegedExceptionAction` interface. By inserting `proceed()` in the implemented `run()` method, we take care of wrapping all operations that require any type and number of arguments, as well as any type of return value. This pattern saves us from writing a class for each operation that needs authorization.

Let's examine the aspect in more detail:

- 1 The `authOperations()` abstract pointcut is identical to the one in the authentication solution we presented earlier. When we define the pointcut in the subspect, we will list all the operations that need authentication, which are the same as the ones that need authorization. Later, toward the end of chapter, we show you a simple modification you can use if you have to separate the list for operations that need authentication from those that need authorization.
- 2 This abstract method allows the subspects to define the permission needed for the captured operation. It passes the static information about the captured join point to the `getPermission()` method in case the permission depends on a class and method for the operation.
- 3 This around advice first creates a worker object for the captured operation and then executes it using `Subject.doAsPrivileged()` on behalf of the authenticated subject. By using the `&&` operator to combine the `authOperations()` pointcut with `!cflowbelow(authOperations())`, we ensure that the worker object is created only for the top-level operations that need authorization. Note that we do not need to separately route an operation if it is already in the control flow of another routed operation.
- 4 This before advice determines whether the caller of the method has sufficient permissions. Note we did not put the logic to check permissions in the preceding around advice. This is because we first need to create the worker object and pass it to `Subject.doAsPrivileged()`; only then can we check for the permissions called by the worker object.
- 5 `AuthorizationException` is simply a `RuntimeException` that wraps the original exception.

Notice how the two before advice and an around advice to the `authOperations()` pointcut are lexically arranged. (Please refer to section 4.2.4 for more information about how lexical ordering of advice in an aspect affects their precedence.) This arrangement is critical for the correct functioning of this aspect. With this arrangement the advice is executed as follows:

- 1 The first before advice is executed prior to executing the join point. This advice performs the authentication, if needed, and obtains an authenticated subject after authenticating.
- 2 The around advice is executed next. It creates a wrapper worker object and invokes it using `Subject.doAsPrivileged()`. This results in calling the original captured join point when the advice body encounters `proceed()`.
- 3 The second before advice is executed just prior to proceeding with the execution of the captured join point. Essentially, think of the before advice as being called right before the `proceed()` method in the around advice. This advice uses `AccessController.checkPermission()` to check the permission needed.

In summary, by controlling the precedence, we ensure that authentication occurs before authorization; we verify the identity of the subject before we check the permissions for that subject.

To enable authorization in our banking system, we must modify `BankingAuthAspect` to implement the abstract `getPermission()` method. This is all we have to change in order to enable authorization—the reusable base aspect takes care of all the complexities. Listing 10.17 shows `BankingAuthAspect`, which enables authorization in our example banking system.

**Listing 10.17** `BankingAuthAspect.java`: adding authorization capabilities

```
package banking;

import org.aspectj.lang.JoinPoint;

import java.security.Permission;

import auth.AbstractAuthAspect;

public aspect BankingAuthAspect extends AbstractAuthAspect {
    public pointcut authOperations()
        : execution(public * banking.Account.*(..))
        || execution(public * banking.InterAccountTransferSystem.*(..));
```



```

    public Permission getPermission(
        JoinPoint.StaticPart joinPointStaticPart) {
        return new BankingPermission(
            joinPointStaticPart.getSignature().getName());
    }
}

```

In this concrete aspect, we add a definition for the `getPermission()` method. In our implementation, we return a new `BankingPermission` class with the name of the method obtained from the join point's static information as the permission identification string. This permission scheme is identical to the one we used for the conventional solution in listing 10.15.

### 10.6.2 Testing the solution

When we compile all the classes and aspects and run the test program, we see output similar to the following:

```

> ajc banking\*.java auth\*.java logging\*.java
↳ sample\module\*.java sample\principal\*.java
> java -Djava.security.auth.login.config=sample_jaas.config
↳ -Djava.security.policy=security.policy banking.Test
<credit>
  <login>
user name: testUser
password: testPassword
           [SampleLoginModule] user entered user name: testUser
           [SampleLoginModule] user entered password: testPassword
           [SampleLoginModule] authentication succeeded
           [SampleLoginModule] added SamplePrincipal to Subject
  <doAsPrivileged>
<debit>
  <doAsPrivileged>
<transfer>
  <doAsPrivileged>
    <credit>
    <debit>
<transfer>
  <doAsPrivileged>
    <credit>
    <debit>
Exception in thread "main"
↳ auth.AbstractAuthAspect$AuthorizationException:
↳ banking.InsufficientBalanceException: Total balance not sufficient

```

Note that the output is nearly identical to that in section 10.5.4. However, there are a few differences. The first difference is that the login occurs in a different

place due to the just-in-time policy. Second, the log for each operation occurs before the log for the `doPrivileged()` method that routed the operation. This is because the logging aspect has a higher precedence, and its before advice is applied before the around advice in `AbstractAuthAspect`. Refer to chapter 4, section 4.2, for details on aspect precedence rules. Also note that the type of exception thrown by the last `transfer()` call is not the expected `InsufficientBalanceException`. This behavior is due to the fact that any exception thrown by the `PrivilegedExceptionAction.run()` method is wrapped in an `AuthorizationException`. Since we cannot throw a checked exception of a type other than that declared by the method itself, we wrap the exception in a runtime exception `AbstractAuthAspect.AuthorizationException`.

We can remedy the situation by simply adding one more aspect, modeled after the exception introduction pattern in chapter 8, to the system. This aspect's job is to catch the `AbstractAuthAspect.AuthorizationException` thrown by any method that could throw an `InsufficientBalanceException` and check the cause of the thrown exception. If the cause's type is `InsufficientBalanceException`, it then throws the cause exception instead of `AuthorizationException`. Listing 10.18 shows the implementation of this logic in an aspect.

**Listing 10.18** `PreserveCheckedException.java`: aspect preserving checked exceptions

```
package banking;

import auth.AbstractAuthAspect;

public aspect PreserveCheckedException {
    after() throwing(AbstractAuthAspect.AuthorizationException ex)
        throws InsufficientBalanceException
        : call(* banking..*.*(..)
            throws InsufficientBalanceException) {
        Throwable cause = ex.getCause();
        if (cause instanceof InsufficientBalanceException) {
            throw (InsufficientBalanceException) cause;
        }
        throw ex;
    }
}
```

---

In this case, the only exception that we need to preserve is `InsufficientBalanceException`. Now when we compile all the classes and aspects, we see that the checked exception is preserved:

```

> ajc banking\*.java auth\*.java logging\*.java
  ↳ sample\module\*.java sample\principal\*.java
> java -Djava.security.auth.login.config=sample_jaas.config
  ↳ -Djava.security.policy=security.policy banking.Test
<credit>
  <login>
user name: testUser
password: testPassword
           [SampleLoginModule] user entered user name: testUser
           [SampleLoginModule] user entered password: testPassword
           [SampleLoginModule] authentication succeeded
           [SampleLoginModule] added SamplePrincipal to Subject
  <doAsPrivileged>
<debit>
  <doAsPrivileged>
<transfer>
  <doAsPrivileged>
    <credit>
    <debit>
<transfer>
  <doAsPrivileged>
    <credit>
    <debit>
Exception in thread "main" banking.InsufficientBalanceException:
Total balance not sufficient
... the rest of call stack

```

We now have an aspect-oriented solution to authentication and authorization for the banking system. The most beneficial characteristics of this solution are:

- You can add functionality without touching even a single core source file.
- The specifications are captured in a single aspect.
- The base aspect that implements most of the functionality is reusable.

You now should be able to write a simple subspect of this reusable aspect to get a comprehensive access-controlled system.

Now that we have a modularized implementation of authorization concerns, we can quickly react to any changes in the authorization requirements. For example, consider data-driven authorization in a banking system where the credentials needed for performing the fee-waiving operations depend on the amount involved. We can implement this requirement easily by capturing the join points corresponding to the fee-waiving operations and collecting the waived amount as a context. We then advise such join points to check the credentials based on the amount. Consider another requirement: providing the opportunity for re-login with a different identity upon determining that the credentials with the current identity are not sufficient to perform an operation. We can easily imple-

ment this functionality by modifying the authorization advice to present the user with a login opportunity upon authorization failure. In a nutshell, the ease of implementation brought forth by AspectJ-based authorization makes it practical to implement useful variations of the core functionality.

## 10.7 Fine-tuning the solution

---

In this section, we examine a few finer points that you may want to consider when customizing the access control solution for your system.

### 10.7.1 Using multiple subaspects

In most common situations, the list of operations that need authentication and authorization is a system-wide consideration, similar to the solution in this chapter. However, suppose each subsystem must control its list of operations. In this case, you need multiple subaspects, one for each subsystem, each specifying operations in the associated subsystem. For example, the following aspect extends `AbstractAuthAspect` to authenticate all the public operations in the `com.mycompany.secretprocessing` package:

```
public aspect SecretProcessingAuthenticationAspect {
    extends AbstractAuthAspect {
        public pointcut authOperations() :
            execution(public * com.mycompany.secretprocessing.*(..));
    }
}
```

Using this scheme, you can include multiple subaspects in a system, each specifying a list of join points needing authentication and authorization. Then the advice in the base aspect applies to join points captured by the pointcut in each subaspect. This is similar to the participant pattern, in which each class controls the subaspect that defines the pointcuts for the class. However, in this case the subaspect defines the pointcuts for a subsystem, which results in greater flexibility and ease of maintenance for the owners of the subsystem.

Remember that if you use multiple subaspects, the system will create an instance of each of the concrete subaspects that share the common base aspect. If you store the authenticated subject as an instance variable of the base aspect, as we did in the solution in this chapter, the user will be forced to log in multiple times—upon reaching the first join point captured by the pointcut in each concrete subaspect. You will need to store the authenticated subject in a different way. For instance, if your authentication has program scope, you may want to keep the authenticated subject as a static variable inside the `AbstractAuthAspect`.

### 10.7.2 Separating authentication and authorization

In the chapter's solution, we used a single pointcut to capture both authorization and authentication join points. While this scheme is fine in most cases, there are situations when you need to separate these join points. For example, consider a requirement for up-front login. You need the method corresponding to the main entry in the program to be authenticated but not necessarily authorized. Satisfying such a requirement is quite simple. First you need two pointcuts: one for authentication and another for authorization. Then you must modify the aspect we developed to separate out the authentication advice to apply to the authentication pointcut, and you will have to modify the authorization advice in a similar way.

What happens if your authorization join point is encountered prior to an authentication one? The solution depends on your system's requirements. One solution is to fall back to just-in-time authentication, thus performing authentication prior to the execution of the first method that needs to check authorization (if the user was never authenticated). The easiest way to achieve this would be to include an authorization pointcut in an authentication pointcut as well:

```
pointcut authenticatedOperations()  
    : primaryAuthenticatedOperations() || authorizedOperations();
```

The other possibility is to simply throw an exception if an authorization join point is reached before the user is authenticated. Checking to see if the `_authenticatedSubject` is null in the authorization advice may be the easiest option. Both the choices can be implemented easily, and the choice you make depends on your system requirements.

## 10.8 Summary

---

The JAAS API provides a standard way to introduce authentication and authorization into your system without requiring application developers to know the complex implementation details. The conventional JAAS-based solution suffers from code bloat and poses the problem of having no single place to list or enforce authentication and authorization decisions. On a large system, this makes it almost impossible to figure out which operations are being authorized. Further, it separates the implementation on the caller side from the callee side. Failing to add an authentication check on the caller side leads to making resources unavailable to otherwise qualified users. Failing to add an authorization check on the callee side, on the other hand, results in potential unauthorized access to the operations, compromising the system's integrity.

The beauty of an AspectJ solution for authentication and authorization lies in modularizing the access control implementation into a few modules, separate from the core system logic. You still use JAAS to perform the core part of authentication and authorization, but you no longer need to have calls to its API all over the system. By simply including a few aspects and specifying operations that require access control, you complete the implementation. If you have to add or remove operations under access control, you just change the list of operations needing such control—no change is required to the core parts of the system. AOP and AspectJ make authentication and authorization not only easy to implement but also easy to evolve.

By combining such aspects along with those in the rest of the book, you could create an EJB-lite framework and benefit from improved control over the services you need.

# AspectJ IN ACTION


 AspectJ  
v1.1

## Practical Aspect-Oriented Programming

Ramnivas Laddad

**M**odularizing code into objects cannot be fully achieved in pure OOP. In practice some objects must deal with aspects that are not their main business. A method to modularize aspects—and benefit from a clean maintainable result—is called aspect-oriented programming. AspectJ is an open-source Java extension and compiler designed for AOP development. Now integrated with Eclipse, NetBeans, JBuilder, and other IDEs, AspectJ v1.1 is ready for the real world.

It is time to move from AOP theory and toy examples to AOP practice and real applications. With this unique book you can make that move. It teaches you AOP concepts, the AspectJ language, and how to develop industrial-strength systems. It shows you examples which you can reuse. It unleashes the true power of AOP through unique *patterns* of AOP design. When you are done, you will be eager—and able—to build new systems, and enhance your existing ones, with the help of AOP.

### What's Inside

- What is aspect-oriented programming?
- How AspectJ works with JAAS, Jess, log4j, Ant, JTA, POJOs
- Best practices and design patterns **NEW**
- How to implement
  - policy enforcement
  - resource pooling and caching
  - thread-safety
  - authentication and authorization
  - transaction management
  - business rules **NEW**

**Ramnivas Laddad** is an AOP and AspectJ authority. With his writings, he has contributed to the general awareness of AOP and has contributed to features now incorporated in AspectJ Version 1.1. Ramnivas lives in Sunnyvale, California.

“Speaks directly to me as a developer”

—Alan Cameron Wills  
fastnloose

“... real solutions to tough problems ...”

—Chris Bartling, Identix, Inc.

“I started reading at 11 PM and couldn't stop ... It's a must read for anyone interested in the future of programming.”

—Arno Schmidmeier  
AspectSoft

“... The only resource that presents AOP concepts and real-world examples in an approachable, readable way.”

—Jean Baltus  
Metafro-Infosys

[www.manning.com/laddad](http://www.manning.com/laddad)

**AUTHOR  
ONLINE**

Author responds to reader questions



Ebook edition available



9 781930 110939

5 4 4 9 5

ISBN 1-930110-93-6