3 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

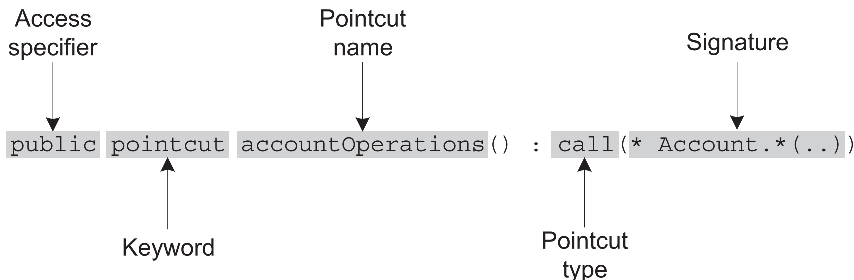

Figure 3.1 Defining a named pointcut. A named pointcut is defined using the `pointcut` keyword and has a name. The part after the colon defines the captured join points using the pointcut type and signature.

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
`Account`	Type of name `Account`.
`*Account`	Types with a name ending with `Account` such as `SavingsAccount` and `CheckingAccount`.
`java.*.Date`	Type `Date` in any of the direct subpackages of the `java` package, such as `java.util.Date` and `java.sql.Date`.
`java..*`	Any type inside the `java` package or all of its direct subpackages, such as `java.awt` and `java.util`, as well as indirect subpackages, such as `java.awt.event` and `java.util.logging`.
`javax..*Model+`	All the types in the `javax` package or its direct and indirect subpackages that have a name ending in `Model` and their subtypes. This signature would match `TableModel`, `TreeModel`, 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
`!Vector`	All types other than `Vector`.
`Vector || Hashtable`	`Vector` or `Hashtable` type.
`javax..*Model || javax.swing.text.Document`	All types in the `javax` package or its direct and indirect subpackages that have a name ending with `Model` or `javax.swing.text.Document`.
`java.util.RandomAccess+ && java.util.List+`	All types that implement both the specified interfaces. This signature, for example, will match `java.util.ArrayList` 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
`public void Collection.clear()`	The method `clear()` in the `Collection` class that has `public` access, returns `void`, and takes no arguments.
`public void Account.debit(float) throws InsufficientBalanceException`	The public method `debit()` in the `Account` class that returns `void`, takes a single `float` argument, and declares that it can throw `InsufficientBalanceException`.

Table 3.3 Examples of method signatures *(continued)*

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

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

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
`private float Account._balance`	Private field `_balance` of the `Account` class
`* Account.*`	All fields of the `Account` class regardless of an access modifier, type, or name

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

Signature Pattern	Matched Fields
`!public static * banking..*.*`	All nonpublic `static` fields of `banking` and its direct and indirect subpackages
`public !final *.*`	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 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	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(ConstructorSignature)`
Advice execution	`adviceexecution()`

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

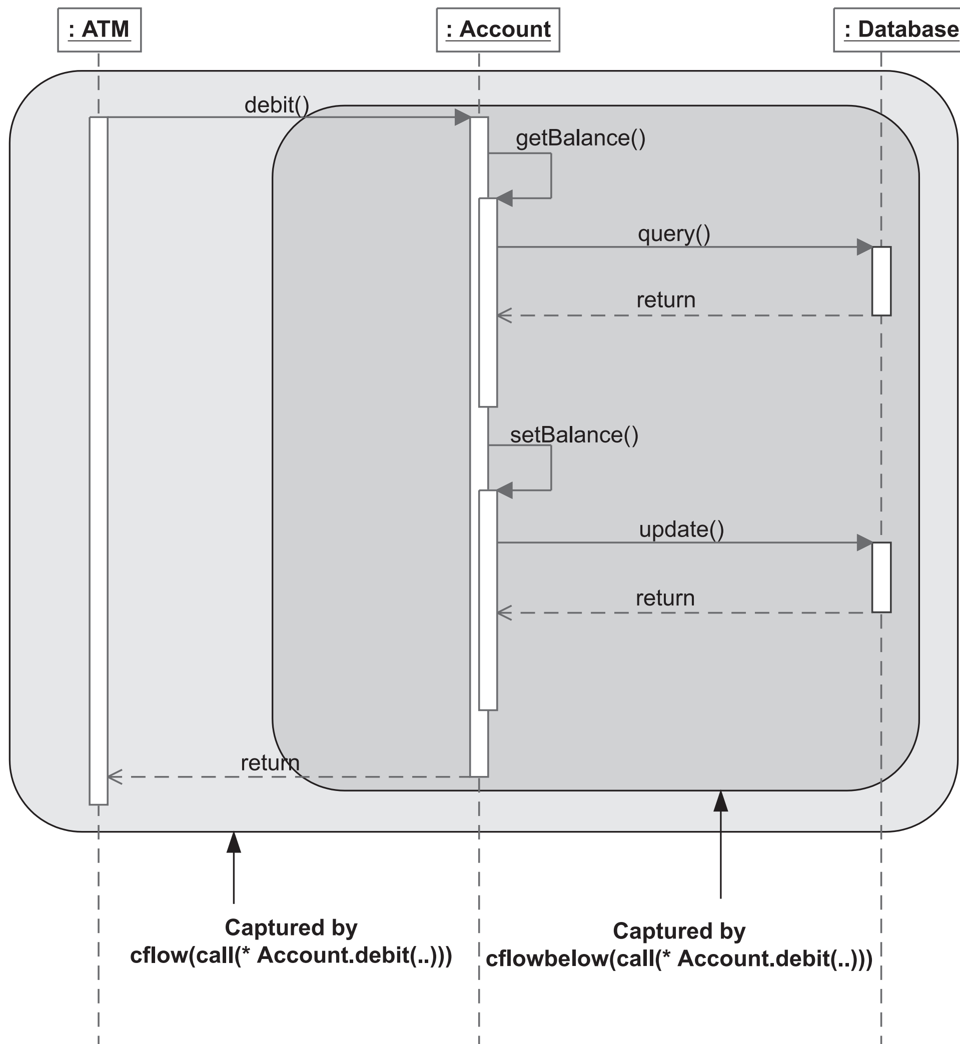

Figure 3.2 Control-flow based pointcuts capture every join point occurring in the control flow of join points matching the specified pointcut. The `cflow()` pointcut includes the matched join point itself, thus encompassing all join points occurring inside the outer box, whereas `cflowbelow()` excludes 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
`within(Account)`	Any join point inside the `Account` class's lexical scope
`within(Account+)`	Any join point inside the lexical scope of the `Account` class and its subclasses
`withincode(* Account.debit(..))`	Any join point inside the lexical scope of any `debit()` method of the `Account` class
`withincode(* *Account.getBalance(..))`	Any join point inside the lexical scope of the `getBalance()` method in classes whose name ends in `Account`

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

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 {

    ...                                                   Captured by
                                                          within(Account)
    public void debit(float amount)
        throws InsufficientBalanceException {             Captured by
        ...                                               this(Account)
    }

    private static class Helper {             Captured by
        ...                                   within(Account)
    }
}

public class SavingsAccount extends 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
`args(String,.., int)`	All the join points in all methods where the first argument is of type `String` and the last argument is of type `int`.
`args(Remote-Exception)`	All the join points with a single argument of type `RemoteException`. It would match a method taking a single `RemoteException` argument, a field write access setting a value of type `RemoteException`, or an exception handler of type `RemoteException`.

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
`if(System.currentTimeMillis() > triggerTime)`	All the join points occurring after the current time has crossed the `triggerTime` value.
`if(circle.getRadius() < 5)`	All the join points where the `circle`'s radius is smaller than 5. The `circle` 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.

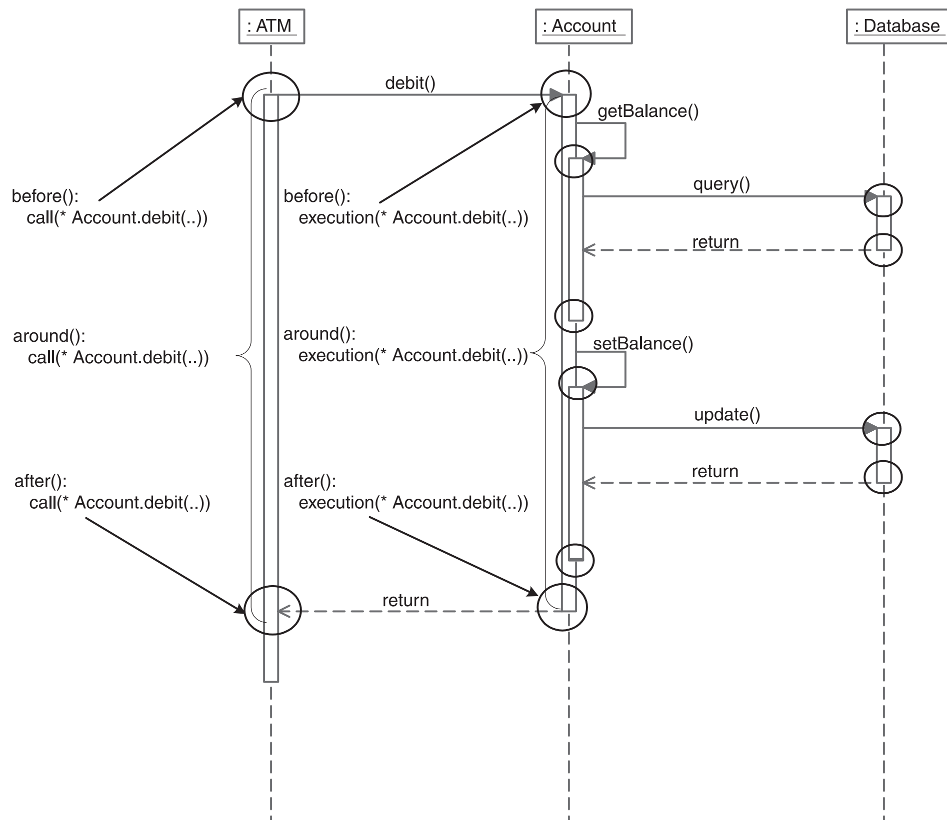

Figure 3.3 Various points in a program flow where you can advise the join point (not all possible points are shown). Each circle represents an opportunity for before or after advice. The passage between the matching circles on each lifeline represents an opportunity for around advice.

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):                          ❶ Advice declaration
   connectionOperation (connection) {                   ❷ Pointcut specification
   System.out.println("Performing operation on " + connection);      ❸
}

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

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

❶ 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.

❷ 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.

❸ 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 );
}
```


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 "
       + thisJointPoint );
}
```

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                    ❶ Method part of advice
        : call(* RemoteService.get*(..) throws RemoteException) {   ❷ Pointcut (anonymous) part of advice
        int retry = 0;
        while(true){
            try{
                return proceed();                              ❸ Execution of captured join point
            } catch(RemoteException ex){
                System.out.println("Encountered " + ex);
                if (++retry > MAX_RETRIES) {
                    throw ex;
                }
                System.out.println("\tRetrying...");
            }
        }
    }
}
```

❶ 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.

❷ The pointcut part of the advice uses an anonymous pointcut to capture all the getter methods in `RemoteService` that throw `RemoteException`.

❸ 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...
Replying
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);            // 1 Capturing context using args()

    pointcut topLevelFactorialOperation(int n) :
        factorialOperation(n)
        && !cflowbelow(factorialOperation(int));           // 2 Capturing context from another pointcut

    private Map _factorialCache = new HashMap();

    before(int n) : topLevelFactorialOperation(n) {        // 3 Using pointcut's context
        System.out.println("Seeking factorial for " + n);
    }

    long around(int n) : factorialOperation(n) {           // 4 Returning primitive from around advice
        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);                                 // 5 Passing along context to proceed()
    }

    after(int n) returning(long result)                    // 6 Capturing return value
        : topLevelFactorialOperation(n) {
        _factorialCache.put(new Integer(n), new Long(result));
    }
}
```

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;            <- Introducing a data member

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

    after(Account account) :
        execution(SavingsAccount.new(..)) && this(account) {
        account._minimumBalance = 25;        <- Using the introduced data member
    }
```

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

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 complier 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.