4 Advanced AspectJ

This chapter covers

- Using reflection support to access join point information
- Using aspect precedence to coordinate multiple aspects
- Creating reusable aspects with aspect association
- Exception softening and privileged aspects

The core concepts presented earlier equipped you with basic AspectJ constructs so that you can begin to implement crosscutting functionality in your system. For complex applications involving the creation of reusable aspects and the use of multiple aspects, you will need advanced AspectJ concepts and constructs to provide you with additional options for design and implementation.

This chapter introduces more advanced features of AspectJ, such as aspect precedence and aspect association. Unlike the earlier chapters, where concepts build on top of one another, this chapter contains a collection of constructs that each stand alone.

4.1 Accessing join point information via reflection

Reflective support in AspectJ provides programmatic access to the static and dynamic information associated with the join points. For example, using reflection, you can access the name of the currently advised method as well as the argument objects to that method. The dynamic context that can be captured using reflective support is similar to that captured using `this()`, `target()`, and `args()` pointcuts—only the mechanism to obtain the information is different. The most common use of this reflective information is in aspects that implement logging and similar functionaliy. We have already used simple reflective support to write the `JoinPointTraceAspect` in chapter 2. In this section, we examine the details of reflective support.

NOTE While you can always use reflection to obtain the dynamic context, the preferred way is to use the `this()`, `target()`, and `args()` pointcuts. The reflective way of accessing information has poor performance, lacks static type checking, and is cumbersome to use. However, there are times when you need to use reflection because you need to access dynamic context and little information is available or required about the advised join points. For instance, you cannot easily use an `args()` pointcut to capture arguments for all logged methods, since each method may take a different number and type of arguments. Further, the logging aspect's advice doesn't need to care about the type of the argument objects because the only interaction of the logging aspect with those objects is to print them.

AspectJ provides reflective access by making three special objects available in each advice body: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`. These objects are much like the special variable `this` that

is available in each instance method in Java to provide access to the execution object. The information contained in these three objects is of two types: *dynamic* information and *static* information:

- Dynamic information consists of the kind of information that changes with each invocation of the same join points. For example, two different calls to the method `Account.debit()` will probably have different account objects and debit amounts.
- Static information is information that does not change between the multiple executions. For example, the name and source location of a method remain the same during different invocations of the method.

Each join point provides one object that contains dynamic information and two objects that contain static information about the join point and its enclosing join point. Let's examine the information in each of these special objects. We will examine the API to access the information from these objects in section 4.1.1:

- *`thisJoinPoint`*—This object of type `JoinPoint` contains the dynamic information of the advised join point. It gives access to the target object, the execution object, and the method arguments. It also provides access to the static information for the join point, using the `getStaticPart()` method. You use `thisJoinPoint` when you need dynamic information related to the join point. For example, if you want to log the execution object and method arguments, you would use the `thisJoinPoint` object.
- *`thisJoinPointStaticPart`*—This object of type `JoinPoint.StaticPart` contains the static information about the advised join point. It gives access to the source location, the kind (method-call, method-execution, field-set, field-get, and so forth), and the signature of the join point. You use `thisJoinPointStaticPart` when you need the structural context of the join point, such as its name, kind, source location, and so forth. For example, if you need to log the name of the methods that are executed, you would use the `thisJoinPointStaticPart` object.
- *`thisEnclosingJoinPointStaticPart`*—This object of type `JoinPoint.StaticPart` contains the static information about the enclosing join point, which is also refered to as the enclosing context. The enclosing context of a join point depends on the kind of join point. For example, for a method-call join point, the enclosing join point is the execution of the caller method, whereas for an exception-handler join point, the enclosing join point is the method that surrounds the catch block. You use the `thisEn-`

`closingJoinPointStaticPart` object when you need the context information of the join point's enclosing context. For example, while logging an exception, you can log the enclosing context information as well.

4.1.1 The reflective API

The reflective API in AspectJ is a set of interfaces that together form the programmatic access to the join point information. These interfaces provide access to dynamic information, static information, and various join point signatures. In this section, we examine these interfaces and their relationship with each other. Figure 4.1 shows the structural relationship between the interfaces of the reflective API in a UML class diagram.

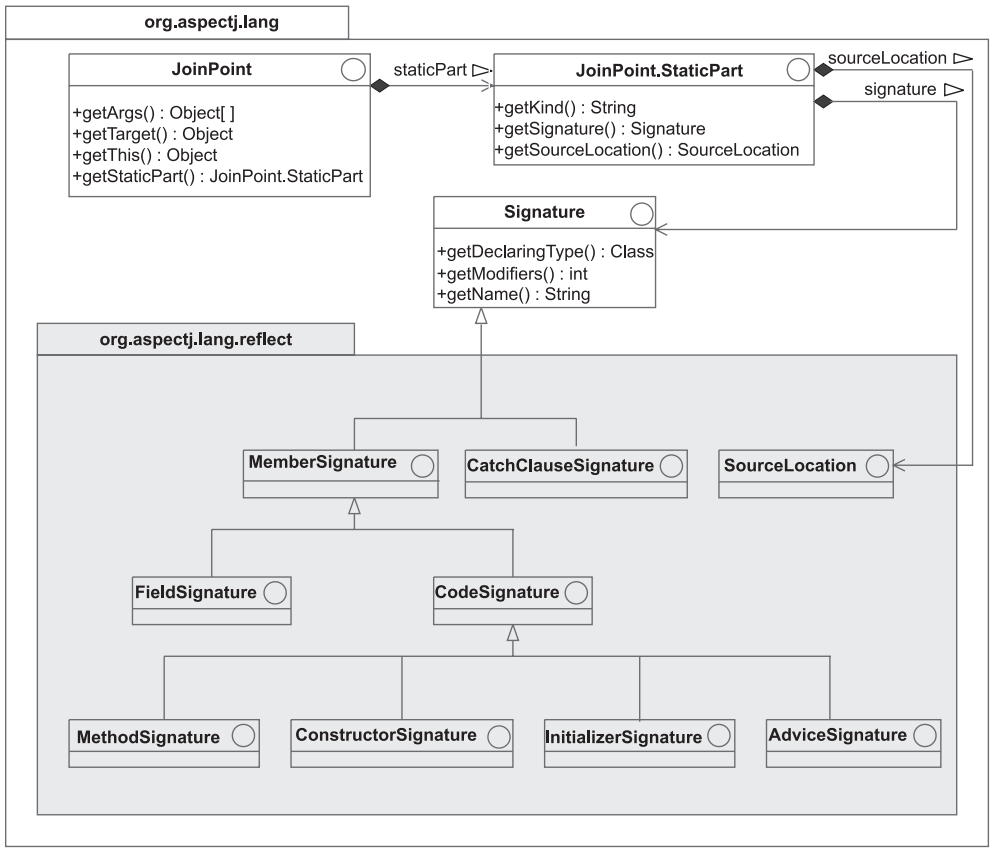


Figure 4.1 The structural relationship among various interfaces supporting reflection

The package `org.aspectj.lang` contains three interfaces and a subpackage. These modules provide support to access all of the join point's information. The `JoinPoint` interface models dynamic information associated with an advised join point. A `JoinPoint` object also contains an object of type `JoinPoint.StaticPart` that can be accessed through the method `getStaticPart()`. This object provides access to the join point's static information. This static information consists of the join point's "kind," signature, and source code location. A `JoinPoint.StaticPart` object is composed of a `String` object (that represents the "kind"), a `Signature` object, and a `SourceLocation` object. The `Signature` object provides access to the join point's signature, and the `SourceLocation` object provides access to the join point's source-code location. The subpackage `org.aspectj.lang.reflect` contains interfaces for various join point signatures connected through an inheritance relationship, as well as the `SourceLocation` interface.

NOTE The purpose of the API discussion in this section is to give an overview. For more detailed information, refer to the AspectJ API documentation.

The org.aspectj.lang.JoinPoint interface

This interface provides access to the dynamic information associated with the currently advised join point. It specifies methods for obtaining the currently executing object, target object, and arguments, as well as the static information:

- The `getThis()` method gives access to the currently executing object, whereas the `getTarget()` method is used for obtaining the target object for a called join point. The `getThis()` method returns `null` for join points occurring in a static method, whereas `getTarget()` returns `null` for the calls to static methods.
- The `getArgs()` method gives access to arguments for the join point. For method and constructor join points, `getArgs()` simply returns an array of each element referring to each argument in the order they are supplied to the join point. Each primitive argument is wrapped in the corresponding wrapper type. For example, an `int` argument will be wrapped inside an `Integer` object. For field-set join points, the new value of the field is available in `getArgs()`. For field-get join points, `getArgs()` returns an empty array, since there is no argument for the operation. Similarly, for handler execution, `getArgs()` returns the exception object.

Besides providing access to the dynamic information, the `JoinPoint` interface offers direct access to the static information of the advised join point. There are two ways to obtain the static information through the `thisJoinPoint` variable of type `JoinPoint`:

- By using direct methods (`getKind()`, `getSignature()`, and `getSourceLocation()`) with the `thisJoinPoint` object. The next section discusses these methods since they are also defined in the `JoinPoint.StaticPart` interface, where they perform identical tasks.
- Through the object obtained with `getStaticPart()`, which contains the same information as `thisJoinPointStaticPart`.

The org.aspectj.lang.JoinPoint.StaticPart interface

This interface allows the API to access the static information associated with the currently advised join point. It specifies methods to obtain the kind of join point, the join point signature, and the source location information corresponding to code for the join point:

- The method `getKind()` returns the kind of join point. The method returns a string such as "method-call", "method-execution", or "field-set" that indicates the kind of the advised join point. The `JoinPoint` interface defines one constant for each of the available kinds of join points.
- The method `getSignature()` returns a `Signature` object for the executing join point. Depending on the nature of the join point, it can be an instance of one of the subinterfaces shown in figure 4.1. While the base `Signature` interface allows access to common information such as the name, the declaring type, and so forth, you will have to cast the object obtained through `getSignature()` to a subinterface if you need finer information (the type of method argument, its return type, its exception, and so on).
- The method `getSourceLocation()`, which returns a `SourceLocation` object, allows access to the source location information corresponding to the join point. The `SourceLocation` interface contains a method for accessing the source filename, line number, and so forth.

Each of the `JoinPoint`, `JoinPoint.StaticPart`, and `Signature` interfaces specifies three methods for obtaining string representation of the object with varied descriptiveness: `toString()` (which suffices for most debug logging needs), `toLongString()`, and `toShortString()`.

NOTE The `thisJoinPoint` object is allocated every time an advice is executed to capture the current dynamic context, whereas the `thisJoinPointStaticPart` is allocated only once per join point during the execution of a program. Therefore, using dynamic information is expensive compare to static information. You should be aware of this fact while designing aspects such as logging.

Also note that the static information obtained through both `thisJoinPoint.getStaticPart()` and `thisJoinPointStaticPart` is the same. In many situations, such as low-overhead logging and profiling, you need to gather only static, and not dynamic, information about the join point. In those cases, you should use the `thisJoinPointStaticPart` object directly instead of the object obtained through `thisJoinPoint.getStaticPart()`. The first method does not require allocation of a separate object (`thisJoinPoint`) for each join point execution and thus is a lot more efficient.

4.1.2 Using reflective APIs

To demonstrate the use of reflective APIs, let's modify the simple tracing aspect that we wrote in chapter 2. If you recall, `JoinPointTraceAspect` used simple reflective support to print the information for all the join points as the code in the classes executed. We will use the same unmodified classes, `Account`, `InsufficientBalanceException`, and `SavingsAccount`, from listings 2.5, 2.6, and 2.7. The abstract `Account` class provides the methods for debiting, crediting, and querying the account balance. The `SavingsAccount` class extends `Account` to a savings account. We will modify the versions of the `Test` class (from listing 2.8) and `JoinPointTraceAspect` (from listing 2.9) so that our new example will use the reflection API to log detailed messages that show information not only about the methods invoked, but also about the objects involved in each method invocation.

To limit the output, we first remove the call to the `debit()` method in the `Test` program as shown in listing 4.1.

Listing 4.1 Test.java

```
public class Test {
    public static void main(String[] args) {
        SavingsAccount account = new SavingsAccount(12456);
        account.credit(100);
    }
}
```

Now let's modify the `JoinPointTraceAspect` aspect, as shown in listing 4.2. Instead of printing the string representation of `thisJoinPoint` that showed only the signature and kind of the join point, we want it to print the join point's dynamic information, which is the `this` object, the target object, and the arguments at the join point, as well as its static information, which consists of the signature, the kind, and the source location of the join point in the before advice. We will also limit the trace join points for the purpose of limiting the output.

Listing 4.2 JoinPointTraceAspect.java

```
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

public aspect JoinPointTraceAspect {
    private int _indent = -1;

    pointcut tracePoints() :                                    Defining trace join points (1)
        !within(JoinPointTraceAspect)
        && !call(*.new(..)) && !execution(*.new(..))
        && !initialization(*.new(..)) && !staticinitialization(*);

    before() : tracePoints() {
        _indent++;
        println("========= " + thisJoinPoint + " ===========");
        println("Dynamic join point information:");
        printDynamicJoinPointInfo(thisJoinPoint);
        println("Static join point information:");
        printStaticJoinPointInfo(thisJoinPointStaticPart);
        println("Enclosing join point information:");
        printStaticJoinPointInfo(thisEnclosingJoinPointStaticPart);
    }                                                           Obtaining reflective access objects (2)

    after() : tracePoints() {
        _indent--;
    }

    private void printDynamicJoinPointInfo(JoinPoint joinPoint) {
        println("This: " + joinPoint.getThis() +
                " Target: " + joinPoint.getTarget());
        StringBuffer argStr = new StringBuffer("Args: ");
        Object[] args = joinPoint.getArgs();
        for (int length = args.length, i = 0; i < length; ++i) {
            argStr.append(" [" + i + "] = " + args[i]);
        }                                                       Printing dynamic information (3)
        println(argStr);
    }
```

```
    private void printStaticJoinPointInfo(
        JoinPoint.StaticPart joinPointStaticPart) {
        println("Signature: " + joinPointStaticPart.getSignature()
                + " Kind: " + joinPointStaticPart.getKind());
        SourceLocation sl = joinPointStaticPart.getSourceLocation();
        println("Source location: " +
                sl.getFileName() + ":" + sl.getLine());
    }

    private void println(Object message) {
        for (int i = 0, spaces = _indent * 2; i < spaces; ++i) {
            System.out.print(" ");
        }
        System.out.println(message);
    }
}
```

Printing static information ❹

❶ The `tracePoints()` pointcut excludes the join points inside the aspect itself by using the `!within()` pointcut. Without the pointcut, the method calls within the advice in the aspect will get advised. When the advice executes for the first method, it will encounter a method call inside itself, and the advice will be called again. This will begin the infinite recursion. To limit the trace output, we also exclude the join points for the call and execution of constructors as well as object and class initialization.

❷ The advice body passes the reflective objects to the helper methods to print information contained in them.

❸ The `printDynamicJoinPointInfo()` method prints the dynamic information passed in the argument object. We first print the current execution object and the method target object by using `getThis()` and `getTarget()`, respectively. Note that `getThis()` will return `null` for the static method execution, whereas `getTarget()` will return `null` for the static method call. The `getArgs()` method returns an object array with each primitive argument wrapped in a corresponding type. For example, our float argument is wrapped in a `Float` object.

❹ The `printStaticJoinPointInfo()` method prints static information passed in the argument object. We print the signature of the join point and the kind of join point. We also print the source location information obtained through `getSourceLocation()`, returning a `SourceLocation` object that contains such information as the source file, the declaring class, and the line number.

When we run the program, we see the following output. You can see how `getThis()`, `getTarget()`, and `getArgs()` behave for different kinds of join points:

```
> ajc *.java
> java Test
========= execution(void Test.main(String[])) ===========
Dynamic join point information:
This: null Target: null
Args:  [0] = [Ljava.lang.String;@1eed786
Static join point information:
Signature: void Test.main(String[]) Kind: method-execution
Source location: Test.java:3
Enclosing join point information:
Signature: void Test.main(String[]) Kind: method-execution
Source location: Test.java:3
  ========= preinitialization(SavingsAccount(int)) ===========
  Dynamic join point information:
  This: null Target: null
  Args:  [0] = 12456
  Static join point information:
  Signature: SavingsAccount(int) Kind: preinitialization
  Source location: SavingsAccount.java:5
  Enclosing join point information:
  Signature: SavingsAccount(int) Kind: preinitialization
  Source location: SavingsAccount.java:5
  ========= preinitialization(Account(int)) ===========
  Dynamic join point information:
  This: null Target: null
  Args:  [0] = 12456
  Static join point information:
  Signature: Account(int) Kind: preinitialization
  Source location: Account.java:7
  Enclosing join point information:
  Signature: Account(int) Kind: preinitialization
  Source location: Account.java:7
  ========= set(int Account._accountNumber) ===========
  Dynamic join point information:
  This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
  Args:  [0] = 12456
  Static join point information:
  Signature: int Account._accountNumber Kind: field-set
  Source location: Account.java:8
  Enclosing join point information:
  Signature: Account(int) Kind: constructor-execution
  Source location: Account.java:8
  ========= call(void Account.credit(float)) ===========
  Dynamic join point information:
  This: null Target: SavingsAccount@1ad086a
  Args:  [0] = 100.0
  Static join point information:
  Signature: void Account.credit(float) Kind: method-call
  Source location: Test.java:4
  Enclosing join point information:
  Signature: void Test.main(String[]) Kind: method-execution
```

```
Source location: Test.java:3
  ========= execution(void Account.credit(float)) ===========
  Dynamic join point information:
  This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
  Args:  [0] = 100.0
  Static join point information:
  Signature: void Account.credit(float) Kind: method-execution
  Source location: Account.java:12
  Enclosing join point information:
  Signature: void Account.credit(float) Kind: method-execution
  Source location: Account.java:12
    ========= call(float Account.getBalance()) ===========
    Dynamic join point information:
    This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
    Args:
    Static join point information:
    Signature: float Account.getBalance() Kind: method-call
    Source location: Account.java:12
    Enclosing join point information:
    Signature: void Account.credit(float) Kind: method-execution
    Source location: Account.java:12
      ========= execution(float Account.getBalance()) ===========
      Dynamic join point information:
      This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
      Args:
      Static join point information:
      Signature: float Account.getBalance() Kind: method-execution
      Source location: Account.java:26
      Enclosing join point information:
      Signature: float Account.getBalance() Kind: method-execution
      Source location: Account.java:26
        ========= get(float Account._balance) ===========
        Dynamic join point information:
        This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
        Args:
        Static join point information:
        Signature: float Account._balance Kind: field-get
        Source location: Account.java:26
        Enclosing join point information:
        Signature: float Account.getBalance() Kind: method-execution
        Source location: Account.java:26
    ========= call(void Account.setBalance(float)) ===========
    Dynamic join point information:
    This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
    Args:  [0] = 100.0
    Static join point information:
    Signature: void Account.setBalance(float) Kind: method-call
    Source location: Account.java:12
    Enclosing join point information:
    Signature: void Account.credit(float) Kind: method-execution
    Source location: Account.java:12
```

```
========= execution(void Account.setBalance(float)) ===========
Dynamic join point information:
This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
Args:  [0] = 100.0
Static join point information:
Signature: void Account.setBalance(float) Kind: method-execution
Source location: Account.java:30
Enclosing join point information:
Signature: void Account.setBalance(float) Kind: method-execution
Source location: Account.java:30
  ========= set(float Account._balance) ===========
  Dynamic join point information:
  This: SavingsAccount@1ad086a Target: SavingsAccount@1ad086a
  Args:  [0] = 100.0
  Static join point information:
  Signature: float Account._balance Kind: field-set
  Source location: Account.java:30
  Enclosing join point information:
  Signature: void Account.setBalance(float) Kind: method-execution
  Source location: Account.java:30
```

Notice in the output that `getThis()` returns `null` for method calls from the `main()` method. This is because it will return `null` for join points in a static method, as we mentioned in ❸ of the discussion of listing 4.2.

In a similar manner, you can build a quick logging functionality to get insight into the program flow of your system. The use of dynamic information can enhance your understanding of the system execution by logging the object and parameter with each join point along with the static information. In chapter 5, we provide a more detailed description of logging. In chapter 10, we use reflective information for creating authorization permission objects.

4.2 Aspect precedence

When a system includes multiple aspects, it's possible that advice in more than one aspect applies to a join point. In such situations, it may be important to control the order in which the advice is applied. To understand the need for controlling the advice execution order, let's look at the example in listing 4.3. Consider a class representing a home, with the methods of entering and exiting the home.

Listing 4.3 Home.java

```
public class Home {
    public void enter() {
        System.out.println("Entering");
    }
```

```
    public void exit() {
        System.out.println("Exiting");
    }
}
```

Now let's create a security aspect (listing 4.4) consisting of advice for engaging the security system in the home when you exit and disengaging it when you enter.

Listing 4.4 HomeSecurityAspect.java

```
public aspect HomeSecurityAspect {
    before() : call(void Home.exit()) {
        System.out.println("Engaging");
    }

    after() : call(void Home.enter()) {
        System.out.println("Disengaging");
    }
}
```

Another aspect (listing 4.5) handles conserving energy by switching the lights off before you leave the home and switching them on after you enter.

Listing 4.5 SaveEnergyAspect.java

```
public aspect SaveEnergyAspect {
    before() : call(void Home.exit()) {
        System.out.println("Switching off lights");
    }

    after() : call(void Home.enter()) {
        System.out.println("Switching on lights");
    }
}
```

Now let's create a simple test (listing 4.6) to see the effects of multiple advice on a join point.

Listing 4.6 TestHome.java: a simple test to see the effect of multiple advice on a join point

```
public class TestHome {
    public static void main(String[] args) {
        Home home = new Home();

        home.exit();
```

```
        System.out.println();

        home.enter();
    }
}
```

Now when we compile these files together and execute the `Test` program, we see the following output:[1]

```
> ajc Home.java TestHome.java
  ➥ HomeSecurityAspect.java SaveEnergyAspect.java
> java TestHome
Switching off lights
Engaging
Exiting

Entering
Disengaging
Switching on lights
```

The exhibited behavior may not be desirable, considering that switching lights off prior to securing the home may make you fumble in the dark. Also, trying to disarm the security system without the lights on upon entry may cause similar troubles, and any delay in disarming the system may result in calling security. So the preferred sequence when entering the home is *enter-switch on lights-disarm*, and while exiting, *arm-switch off lights-exit*. From the implementation perspective, we would like:

1. The before advice in `SaveEnergyAspect` to run before the `HomeSecurityAspect` before advice
2. The after advice in `SaveEnergyAspect` to run after the `HomeSecurityAspect` after advice

In the next sections, we will study the rules and ways to control precedence. Later we will apply this information to the previous problem to show how you can achieve the correct advice ordering.

[1] It is possible to get output that is different from that shown here, depending on several factors, including the version of the AspectJ compiler you are using. The actual output may match the desired output. Such matching, however, is purely accidental, since the precedence is arbitrarily determined unless you specify the advice precedence.

4.2.1 Ordering of advice

As you have just seen, with multiple aspects present in a system, pieces of advice in the different aspects can often apply to a single join point. When this happens, AspectJ uses the following precedence rules to determine the order in which the advice is applied. Later, we will see how to control precedence:

- The aspect with higher precedence executes its before advice on a join point *before* the one with lower precedence.
- The aspect with higher precedence executes its after advice on a join point *after* the one with lower precedence.
- The around advice in the higher-precedence aspect encloses the around advice in the lower-precedence aspect. This kind of arrangement allows the higher-precedence aspect to control whether the lower-precedence advice will run at all by controlling the call to `proceed()`. In fact, if the higher-precedence aspect does not call `proceed()` in its advice body, not only will the lower-precedence aspects not execute, but the advised join point also will not be executed.

Figure 4.2 illustrates the precedence rules.

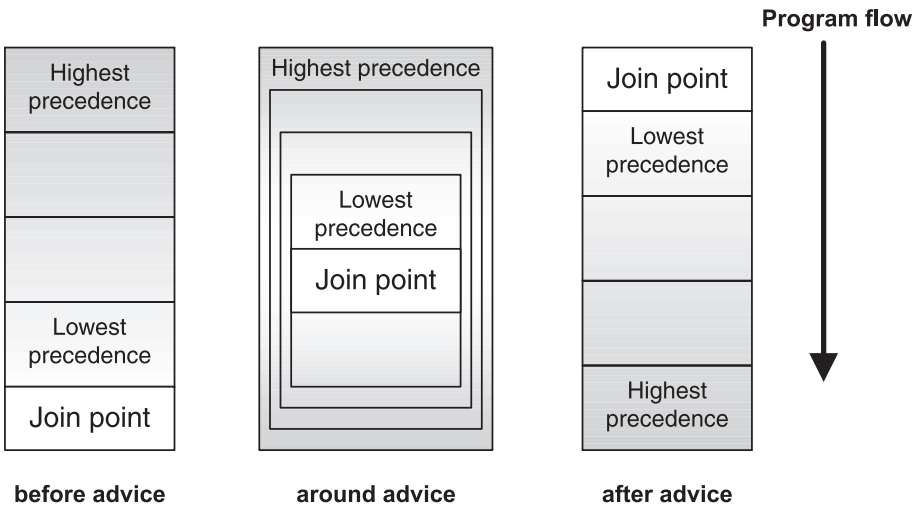


Figure 4.2 Ordering the execution of advice and join points. The darker areas represent the higher-precedence advice. The around advice could be thought of as the higher-precedence advice running the lower-precedence advice in a nested manner.

WARNING In the absence of any special precedence control, the order in which the advice is applied is unpredictable.

4.2.2 Explicit aspect precedence

It is often necessary to change the precedence of advice as it is applied to a join point. AspectJ provides a construct—declare precedence—for controlling aspect precedence. The declare precedence construct must be specified inside an aspect. The construct takes the following form:

```
declare precedence : TypePattern1, TypePattern2, ..;
```

The result of this kind of declaration is that aspects matching the type pattern on the left dominate the ones on the right, thus taking a higher precedence. In this example, the precedence of *TypePattern1* is higher than the precedence of *TypePattern2*. Precedence ordering considers only the concrete aspects when matching the type pattern and ignores all the abstract aspects. By controlling the aspect precedence, you can control the order in which advice is applied to a pointcut. For example, the following declaration causes `AuthenticationAspect` to dominate `AuthorizationAspect`:

```
declare precedence : AuthenticationAspect, AuthorizationAspect;
```

Let's use this declaration to correct the precedence between `HomeSecurityAspect` and `SaveEnergyAspect` in the `Home` class example. Since we want to run the before advice to arm before the before advice to switch off the lights, and the after advice to disarm after the after advice to switch on the lights, we need `HomeSecurityAspect` to dominate `SaveEnergyAspect`. We achieve this goal by writing another aspect (listing 4.7) that declares the correct and explicit precedence between the two.

Listing 4.7 HomeSystemCoordinationAspect.java

```
public aspect HomeSystemCoordinationAspect {
    declare precedence: HomeSecurityAspect, SaveEnergyAspect;
}
```

Now when we compile our code and run the test program we see the following output:

```
> ajc Home.java TestHome.java
    HomeSecurityAspect.java SaveEnergyAspect.java
    HomeSystemCoordinationAspect.java
> java TestHome
Engaging
Switching off lights
Exiting

Entering
Switching on lights
Disengaging
```

This is exactly what we wanted. We could have added the declare precedence clause in either `HomeSecurityAspect` or `SaveEnergyAspect` and gotten the same result. However, this kind of modification would require the creation of an undesirable coupling between the two.

Let's examine more examples of the declare precedence clause to better understand it. Since the clause expects a list of *`TypePattern`*s, we can use wildcards in aspect names. The following declaration causes all aspects whose names start with `Auth`, such as `AuthenticationAspect` and `AuthorizationAspect`, to dominate the `PoolingAspect`:

```
declare precedence : Auth*, PoolingAspect;
```

In this declaration, however, the precedence between two aspects starting with `Auth` is unspecified. If controlling the precedence between two such aspects is important, you will need to specify both aspects in the desired order.

Since declare precedence takes a type list, you can specify a sequence of domination. For example, the following declaration causes aspects whose names start with `Auth` to dominate both `PoolingAspect` and `LoggingAspect`, while also causing `PoolingAspect` to dominate `LoggingAspect`:

```
declare precedence : Auth*, PoolingAspect, LoggingAspect;
```

It is common for certain aspects to dominate all other aspects. You can use a `*` wildcard to indicate such an intention. The following declaration causes `AuthenticationAspect` to dominate all the remaining aspects in the system:

```
declare precedence : AuthenticationAspect, *;
```

It is also common for certain aspects to be dominated by all other aspects. You can use a wildcard to achieve this as well. The following declaration causes `CachingAspect` to have the lowest precedence:

```
declare precedence : *, CachingAspect;
```

It is an error if a single declare precedence clause causes circular dependency in the ordering of aspect precedence. The following declaration will produce a compile-time error since `Auth*` will match `AuthenticationAspect`, causing a circular dependency:

```
declare precedence : Auth*, PoolingAspect, AuthenticationAspect;
```

However, it *is* legal to specify a circular dependency causing precedence in two different clauses. You can use this to enforce that two different, potentially conflicting or redundant aspects, such as two pooling aspects, share no join points. You will get a compile-time error if the two aspects in question share a join point. The following declarations will *not* produce an error unless `PoolingAspect` and `AuthenticationAspect` share a join point:

```
declare precedence : AuthenticationAspect, PoolingAspect;
declare precedence : PoolingAspect, AuthenticationAspect;
```

You can include a declare precedence clause inside any aspect. A common usage idiom is to add such clauses to a separate coordination aspect (such as the one we used in the previous `HomeSystemCoordinationAspect` example) so that aspects themselves are unaware of each other and need no modification to the core aspects. Such a separation is particularly important for third-party, off-the-shelf aspects where you may not have the control over source files you would need to add such clauses. Separating precedence control also avoids the tangling of the core functionality in the precedence relationship with other aspects. The following snippet shows the use of a separate precedence-coordinating aspect in a banking system:

```
aspect BankingAspectCoordinator {
    declare precedence : Auth*, PoolingAspect, LoggingAspect;
}
```

The precedence control offered by AspectJ is simple yet powerful, and is immensely helpful for a complex system. You can now create multiple aspects independently as well as use aspects developed by others without requiring modifications to any other aspect.

4.2.3 Aspect inheritance and precedence

Besides explicitly controlling aspect precedence using the declare precedence construct, AspectJ implicitly determines the precedence of two aspects related by a base-derived aspect relationship. The rule is simple: If the inheritance relates two aspects, the derived aspect implicitly dominates the base aspect. Here's an

example to illustrate this rule. In listing 4.8, the `TestPrecedence` class sets up the scenario to test the precedence in aspect inheritance by calling the `perform()` method from the `main()` method:

Listing 4.8 TestPrecedence.java

```
public class TestPrecedence {
    public static void main(String[] args) {
        TestPrecedence test = new TestPrecedence();
        test.perform();
    }

    public void perform() {
        System.out.println("<performing/>");
    }
}
```

In listing 4.9, the abstract aspect `SecurityAspect` advises the `perform()` method in the `TestPrecedence` class. The advice simply prints a message.

Listing 4.9 SecurityAspect.java

```
public abstract aspect SecurityAspect {
    public pointcut performCall() :
        call(* TestPrecedence.perform());

    before() : performCall() {
        System.out.println("<SecurityAspect:check/>");
    }
}
```

In listing 4.10, the aspect `ExtendedSecurityAspect` uses `SecurityAspect` as the base aspect. It too advises the `perform()` method in the `TestPrecedence` class and prints a message.

Listing 4.10 ExtendedSecurityAspect.java

```
public aspect ExtendedSecurityAspect extends SecurityAspect {
    before() : performCall() {
        System.out.println("<ExtendedSecurityAspect:check/>");
    }
}
```

Now when we compile the class with the aspects, we get the following output. You can observe that the before advice of the derived class was executed before that of the base class:

```
> ajc *.java
> java TestPrecedence
<ExtendedSecurityAspect:check/>
<SecurityAspect:check/>
<performing/>
```

WARNING Since only concrete aspects in the declare precedence clause are designated for precedence ordering, the declaration of a base aspect (which is always abstract) to dominate a child has *no* effect. For example, adding the following clause in the system has no effect:

```
declare precedence : SecurityAspect, ExtendedSecurityAspect;
```

4.2.4 Ordering of advice in a single aspect

It is also possible to have multiple pieces of advice in one aspect that you want to apply to a pointcut. Since the advice resides in the same aspect, aspect precedence rules can no longer apply. In such cases, the advice that appears first lexically inside the aspect executes first. Note that the only way to control precedence between multiple advice in an aspect is to arrange them lexically. Let's illustrate this rule through a simple example (listing 4.11) that shows both the effect of the precedence rule and its interaction between different types of advice. Chapter 10 presents a real-world example in which understanding interadvice precedence is important in authentication and authorization aspects.

Listing 4.11 InterAdvicePrecedenceAspect.java: testing advice ordering in a single aspect

```
public aspect InterAdvicePrecedenceAspect {
    public pointcut performCall() : call(* TestPrecedence.perform());

    after() returning : performCall() {
        System.out.println("<after1/>");
    }

    before() : performCall() {
        System.out.println("<before1/>");
    }

    void around() : performCall() {
        System.out.println("<around>");
        proceed();
```

```
            System.out.println("</around>");
        }

        before() : performCall() {
            System.out.println("<before2/>");
        }
    }
```

After compiling the aspect with the same `TestPrecedence` class in listing 4.8, when we run the code we get this output:

```
> ajc *.java
> java TestPrecedence
<before1/>
<around>
<before2/>
<performing/>
<after1/>
</around>
```

The output shows that:

1. The first before advice is followed by around advice due to their lexical ordering.
2. The second before advice runs after the around advice starts executing, but *before* executing the captured join point. Note that, regardless of precedence, all before advice for a join point must execute before the captured join point itself.
3. The after advice executes *before* completing the around advice, since it has higher precedence than the around advice. Note that the earliest an after advice can run is *after* the join point's execution.

4.2.5 Aspect precedence and member introduction

In rare cases, when multiple aspects introduce data members with the same name or methods with the same signature, the members introduced by the aspect with the higher precedence will be retained and the matching members introduced by other aspects will be eliminated. For example, if you have introduced a method and its implementation in one aspect, and another implementation for the same method in another aspect, only the dominating aspect's implementation will survive. The same is true for data members. If two aspects introduce a member with the same name, type, and initial value, only the member from the dominating aspect will survive.

Listing 4.12 illustrates the effect of aspect precedence on member introduction. Compile these two aspects with `TestPrecedence.java` from listing 4.8. Running the test program will show that the dominating aspect's data and method win over the members of the other aspect.

Listing 4.12 SecurityAspect.java

```
public aspect SecurityAspect {
    private String TestPrecedence._id
        = "SecurityAspect:id";                          // Introducing data

    private void TestPrecedence.printId() {             // Introducing method
        System.out.println(
            "<SecurityAspect:performSecurityCheck id=" + _id + "/>");
    }

    public pointcut performCall() : call(* TestPrecedence.perform());

    before(TestPrecedence test) : performCall() && target(test) {
        System.out.println("<SecurityAspect:before/>");
        System.out.println(test._id);   // Printing introduced data
        test.printId();                 // Invoking introduced method
    }
}
```

`SecurityAspect` introduces a data member and a method in the `TestPrecedence` class. It also invokes the introduced method as well as prints the value of the introduced member in the before advice to test the effect.

In listing 4.13, the `TrackingAspect` aspect introduces the same-named data and a same-named method as the `SecurityAspect` aspect. However, it uses a different initial value for the data member and a different body for the method.

Listing 4.13 TrackingAspect.java

```
public aspect TrackingAspect {
    private String TestPrecedence._id = "TrackingAspect:id";   // Introducing data

    private void TestPrecedence.printId() {
        System.out.println(
            "<TrackingAspect:performTracking id=" + _id + "/>");   // Introducing method
    }
}
```

Let's also add the following precedence-coordinating aspect (listing 4.14).

Listing 4.14 SystemAspectCoordinator.java

```
aspect SystemAspectCoordinator {
    declare precedence : SecurityAspect, TrackingAspect;
}
```

When we compile all the files and run the `TestPrecedence` class, we see this output:

```
> ajc *.java
> java TestPrecedence
<SecurityAspect:before/>
SecurityAspect:id
<SecurityAspect:performSecurityCheck id=SecurityAspect:id/>
<performing/>
```

As you can see, the initial value and method implementation introduced by the dominating `SecurityAspect` override the same in `TrackingAspect`.

4.3 Aspect association

By default, only one instance of an aspect exists in a virtual machine (VM)—much like a singleton class. All the entities inside the VM then share the state of such an aspect. For example, all objects share a resource pool inside a pooling aspect. Usually, this kind of sharing is fine and even desirable. However, there are situations, especially when creating reusable aspects, where you want to associate the aspect's state with an individual object or control flow.

The aspect associations can be classified into three categories:

- Per virtual machine (default)
- Per object
- Per control-flow association

You can specify a nondefault association by modifying the aspect declaration that takes the following form:

```
aspect <AspectName> [<association-specifier>(<Pointcut>)]  {
    ... aspect body
}
```

Note the part in bold. This optional aspect association specification determines how the aspect is associated with respect to the join points captured by the specified pointcut.

4.3.1 Default association

Default association is in effect when you do not include an association specification in the aspect declaration. All the aspects you have seen so far in this book are of this type. This type of association creates one instance of the aspect for the VM, thus making its state shared. To understand aspect creation, let's create an aspect for the banking-related `Account` class (listing 2.5, chapter 2), which provided a simple API for crediting and debiting amounts for an account. Later, we will modify this aspect to show the other kinds of associations: per object and per control flow.

For our discussion of aspect association in this section, let's create an aspect, `AssociationDemoAspect`. Listing 4.15 shows the default association aspect that illustrates when an aspect instance is created. We will also use the `Account` class developed in listing 2.5 in chapter 2. The aspect logs a message in its constructor to designate its creation. Then it prints the aspect's instance and the aspected object.

Listing 4.15 AssociationDemoAspect.java: using default association

```
public aspect AssociationDemoAspect {
    public AssociationDemoAspect() {
        System.out.println("Creating aspect instance");
    }

    pointcut accountOperationExecution(Account account)
        : (execution(* Account.credit(..))
           || execution(* Account.debit(..)))
           && this(account);

    before(Account account)
        : accountOperationExecution(account) {
        System.out.println("JoinPoint: " + thisJoinPointStaticPart
                           + "\n\taspect: " + this
                           + "\n\tobject: " + account);
    }
}
```

❶ Aspect constructor

❷ Account operation pointcut

❸ Advice that prints the aspect and account instance

❶ We print a simple message in the aspect constructor to keep track of when the aspect instance is created.

❷ The `accountOperationExecution()` pointcut captures the execution of the `credit()` and `debit()` methods in the `Account` class. It also captures the `Account` object using the `this()` pointcut so that we can print it in the advice.

❸ The advice to `accountOperationExecution()` prints the static context of the captured join point, the aspect instance, and the `Account` object captured by the pointcut. Note that when used from advice, the object `this` refers to the instance of an aspect and not the execution object at a join point.

Next let's write a simple test program (listing 4.16) that creates two `Account` objects and calls methods on them.

Listing 4.16 TestAssociation.java: testing associations

```
public class TestAssociation {
    public static void main(String[] args) throws Exception {
        SavingsAccount account1 = new SavingsAccount(12245);
        SavingsAccount account2 = new SavingsAccount(67890);
        account1.credit(100);
        account1.debit(100);

        account2.credit(100);
        account2.debit(100);
    }
}
```

When we compile the classes and run the `TestAssociation` program, we see output similar to the following:

```
> ajc *.java
> java TestAssociation
Creating aspect instance                                    <-- Aspect instance creation
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@187aeca
        object: SavingsAccount@e48e1b
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@187aeca
        object: SavingsAccount@e48e1b
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@187aeca
        object: SavingsAccount@12dacd1
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@187aeca
        object: SavingsAccount@12dacd1
```

The output shows that only one instance of the aspect is created, and that instance is available to all advice in the aspect.

4.3.2 Per-object association

Oftentimes, reusable base aspects need to keep some per-object state consisting of the data that is associated with each object, without having sufficient information about the type of objects that will participate in the static crosscutting mechanism of member introduction. Consider a cache-management aspect that needs to track the last access time for each object in the cache so that it can remove from the cache objects that are not accessed for a long duration. Since such cache management is a reusable concept, we want to create a reusable base aspect. By associating a separate aspect instance with each object under cache management and by keeping the field definition for the last accessed time inside the base aspect, we can track the required information for each cache-managed object.

The per-object association feature lets us associate a new aspect instance with an execution or target object by using a pointcut. In the following snippet, a new aspect instance is associated with each new execution object using `perthis()`, which matches the abstract `access()` pointcut:

```
public abstract aspect CacheManagementAspect perthis(access()) {

    ... aspect's state – instance members such as the last accessed time

    abstract pointcut access();

    ... advice to access() pointcut to update the last accessed time

    ... advice using the aspect's state
}
```

As an example, we can enable cache management in a banking application by simply creating a subaspect that provides a definition for the abstract `access()` pointcut:

```
public aspect BankingCacheManagementAspect extends CacheManagementAspect {
    pointcut access() : execution(* banking..Account+.*(..))
                        || execution(* banking..Customer+.*(..));
}
```

Now whenever a join point that is captured by the `access()` pointcut executes (such as the `debit()` method), and the execution object is not previously associated with a `BankingCacheManagementAspect` instance, a new instance of the aspect is created and associated with the execution object. The same scenario will take place with `Customer` objects as well. Effectively, the aspect's state now forms a part of each execution object's state. The advice in the base and derived aspects may then use the state of the aspect as if it were the cached object's state.

With per-object associations, an aspect instance is associated with each object matching the association specification. You can specify two kinds of per-object associations:

- `perthis()`—Associates a separate aspect instance with the execution object (`this`) for the join point matching the pointcut specified inside `perthis()`
- `pertarget()`—Associates a separate aspect instance with the target object for the join point matching the pointcut specified inside `pertarget()`

With object associations, the aspect instance is created when executing a join point of a matching object for the first time. Once an association is created between an object and an instance of the declaring aspect, the association is good for the lifetime of the object. Specifically, executing another matching join point on the same object does not create a new aspect with the object. Figure 4.3 illustrates object association using a UML sequence diagram.

To illustrate, let's modify the aspect `AssociationDemoAspect`. Listing 4.17 shows the use of the `perthis()` association with the `accountOperationExecution` pointcut.

Listing 4.17 AssociationDemoAspect.java: with `perthis()` association

```
public aspect AssociationDemoAspect
    perthis(accountOperationExecution(Account)) {

    public AssociationDemoAspect() {
        System.out.println("Creating aspect instance");
    }

    pointcut accountOperationExecution(Account account)
        : (execution(* Account.credit(..))
           || execution(* Account.debit(..)))
           && this(account);

    before(Account account)
        : accountOperationExecution(account) {
        System.out.println("JoinPoint: " + thisJoinPointStaticPart
                           + "\n\taspect: " + this
                           + "\n\tobject: " + account);
    }
}
```

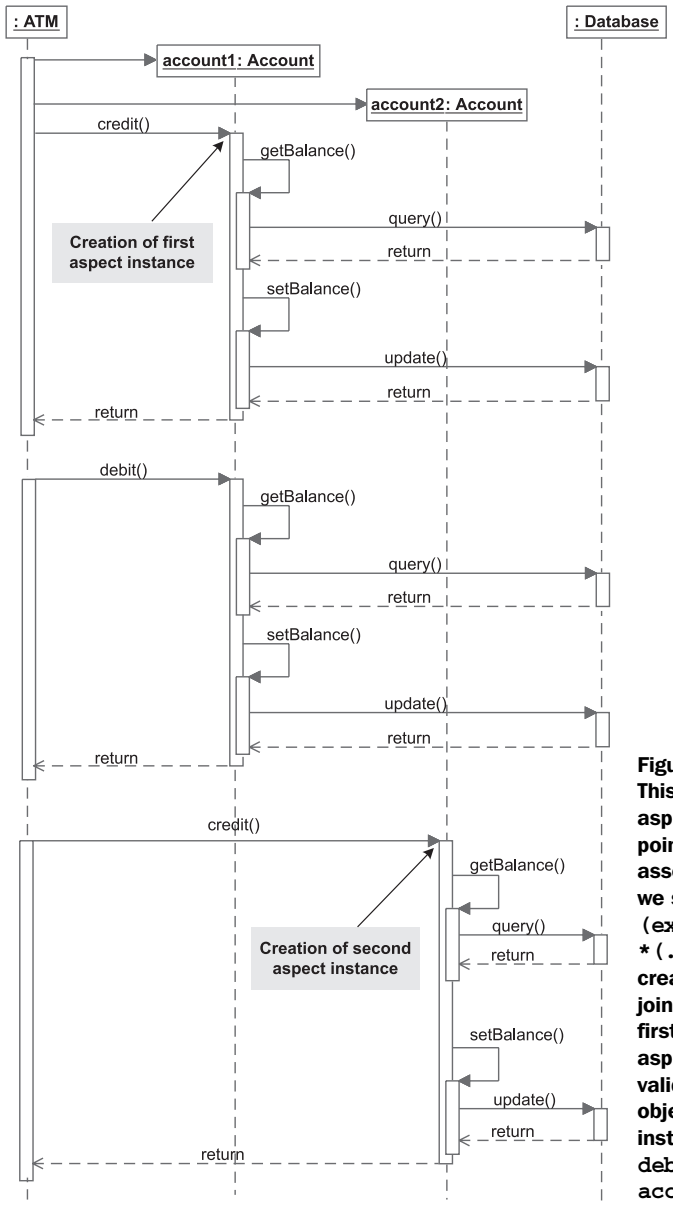
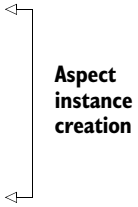


Figure 4.3
This sequence diagram shows aspect creation and association points for object-based association. For this illustration, we specify the `perthis (execution(* Account.*(..)))` association. An aspect is created for each object when the join point matching the pointcut is first executed for that object. The aspect association then remains valid during the lifetime of the object. Notice that no new aspect instance is created when the `debit()` method is invoked on the `account1` object.

Now when we compile this using the modified aspect and run the test program, we get following output:

```
> ajc *.java
> java TestAssociation
Creating aspect instance
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@e48e1b
        object: SavingsAccount@12dacd1
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@e48e1b
        object: SavingsAccount@12dacd1
Creating aspect instance
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@1ad086a
        object: SavingsAccount@10385c1
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@1ad086a
        object: SavingsAccount@10385c1
```



The output shows:

1. Two instances of `AssociationDemoAspect` are created.
2. Each aspect is created right before the execution of the first join point with each `Account` object.
3. In each advice body, the same aspect instance is available for each join point on an object.

To associate an aspect instance with the target object for a matching join point instead of the execution object, you use `pertarget()` instead of `perthis()`.

4.3.3 Per-control-flow association

As with per-object association, you sometimes need per-control-flow association to store per-control-flow states in implementations. You can think of control flow as a conceptual object that encapsulates the thread of execution encompassing a given join point. The per-control-flow state then is data associated with this conceptual control-flow object. With per-control-flow association, an aspect instance is associated with each control flow matching the association specification. Consider the following snippet of a reusable base aspect providing transaction management. This aspect needs to store states needed by the transaction management, such as a JDBC connection object used by all operations:

```
public abstract
    aspect TransactionManagementAspect percflow(transacted()) {

    ... aspect state:
    ...    instance members such as the connection object used
```

```
    abstract pointcut transacted();

    ... advice using the aspect state
}
```

We can then introduce a transaction management capability in a banking application by extending this aspect and providing a definition for the abstract `transacted()` pointcut:

```
public aspect BankingTransactionManagementAspect
    extends TransactionManagementAspect {

    pointcut transacted() : execution(* banking..Account+.*(..))
                          || execution(* banking..Customer+.*(..));
}
```

In this aspect, we introduced transaction management into a banking system by simply specifying the operations that need transaction management support in the definition of the abstract pointcut `transacted()`. This will capture the execution of appropriate methods in banking-related classes. The bulk of transaction management logic resides in the reusable base `TransactionManagementAspect` aspect.

There are a few ways to achieve the goal of creating reusable aspects that need to keep some per-control-flow state without using a control-flow-based association. For example, you could use a thread-specific storage such as `ThreadLocal` to manage the control flow's state. In many cases, however, using an aspect association creates a simpler implementation.

You can specify two kinds of per-control-flow object associations:

- `percflow()`—Associates a separate aspect instance with the control flow at the join point matching the pointcut specified inside `percflow()`
- `percflowbelow()`—Associates a separate aspect instance with the control flow below the join point matching the pointcut specified inside `percflowbelow()`

Much like the `perthis` and `pertarget` cases, once an association is made between a control flow and an aspect instance, it continues to exist for the lifetime of that control flow. Figure 4.4 illustrates the effect of control-flow-based association.

In figure 4.4, we consider an aspect that associates the aspect instance with the control flow of join points that match the execution of any method in the `Account` class. We see that six aspect instances are created—one each for the top-level `credit()` and `debit()` executions, and two each for `getBalance()` and `setBalance()`, which are called from the `credit()` and `debit()` methods. Each aspect instance continues to exist until its join point's execution completes.

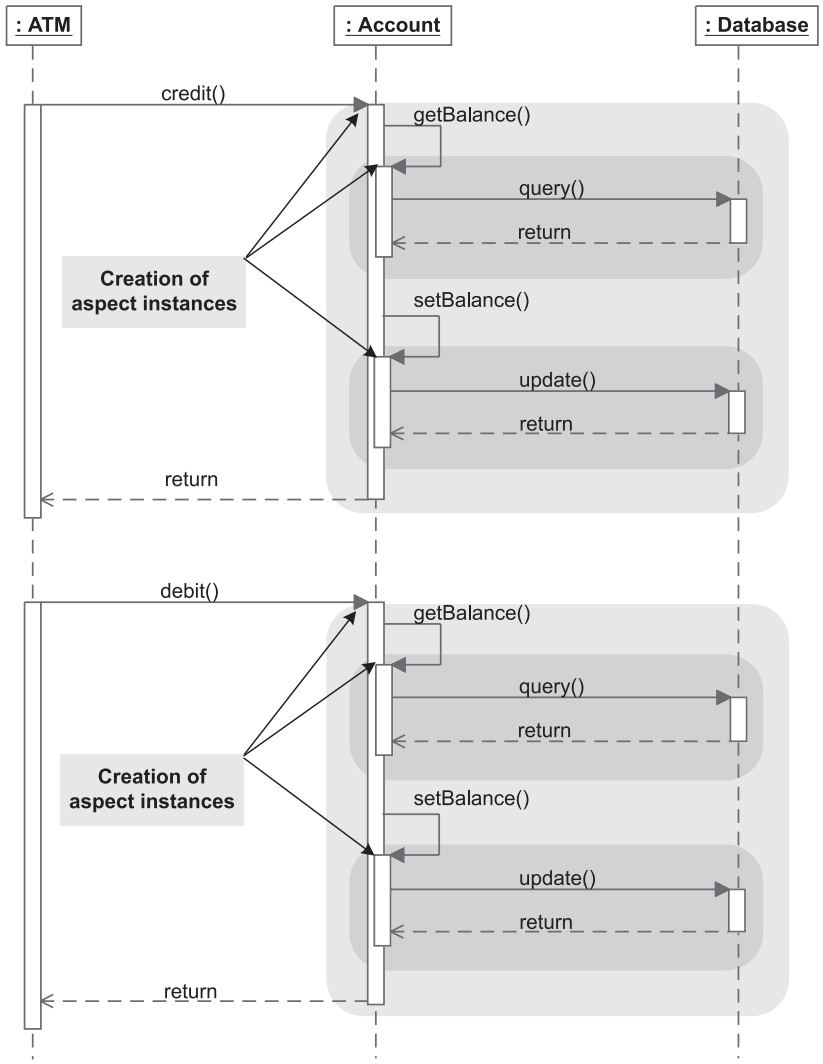


Figure 4.4 This sequence diagram shows aspect creation and association points for control-flow-based associations. In this illustration, we show the `percflow(execution(* Account.*(..)))` association. An aspect is created as soon as each matching control flow is entered for the first time. The aspect association then remains valid during the lifetime of the control flow. Each gray area indicates the scope of the aspect instance that was created upon entering the area.

To better understand control-flow-based association, let's modify `AssociationDemoAspect` again. We will also modify the pointcut in the before advice to include the `setBalance()` execution, as shown in listing 4.18.

Listing 4.18 AssociationDemoAspect.java: with `percflow()` association

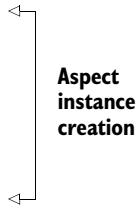
```
public aspect AssociationDemoAspect
    percflow(accountOperationExecution(Account)) {

    public AssociationDemoAspect() {
        System.out.println("Creating aspect instance");
    }

    pointcut accountOperationExecution(Account account)
        : (execution(* Account.credit(..))
           || execution(* Account.debit(..)))
           && this(account);


    before(Account account)
        : accountOperationExecution(account)
        || (execution(* Account.setBalance(..)) && this(account)) {
        System.out.println("JoinPoint: " + thisJoinPointStaticPart
                           + "\n\taspect: " + this
                           + "\n\tobject: " + account);
    }
}
```

When we compile the aspect with the `TestAssociation` class and run the program, we see output similar to the following:

```
> ajc *.java
> java TestAssociation
Creating aspect instance
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@10385c1
        object: SavingsAccount@42719c
JoinPoint: execution(void Account.setBalance(float))
        aspect: AssociationDemoAspect@10385c1
        object: SavingsAccount@42719c
Creating aspect instance
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@30c221
        object: SavingsAccount@42719c
JoinPoint: execution(void Account.setBalance(float))
        aspect: AssociationDemoAspect@30c221
        object: SavingsAccount@42719c
```

Aspect instance creation

```
Creating aspect instance
JoinPoint: execution(void Account.credit(float))
        aspect: AssociationDemoAspect@119298d
        object: SavingsAccount@f72617
JoinPoint: execution(void Account.setBalance(float))
        aspect: AssociationDemoAspect@119298d
        object: SavingsAccount@f72617
Creating aspect instance
JoinPoint: execution(void Account.debit(float))
        aspect: AssociationDemoAspect@1e5e2c3
        object: SavingsAccount@f72617
JoinPoint: execution(void Account.setBalance(float))
        aspect: AssociationDemoAspect@1e5e2c3
        object: SavingsAccount@f72617
```



We now see that:

1. Four instances of the aspect are created, two corresponding to `credit()` and two corresponding to the `debit()` method executions initiated by the `TestAssociation` class. Each execution of the `credit()` and `debit()` methods called from the `TestAssociation` class resulted in a new control flow matching the join point specified in the aspect association pointcut, resulting in a new aspect instance being created.
2. Each instance is created just before the execution of the `credit()` and `debit()` methods, since a new control flow matching the pointcut specified starts with their execution.
3. The `setBalance()` method that is called from the control flow of `debit()` and `credit()` is associated with the same aspect as its caller. Because the `setBalance()` method falls in the control flow of `debit()` and `credit()`, the instance created for the caller continues to be associated with any method called by this caller. Note that if we include the `setBalance()` method in the `accountOperationExecution()` pointcut, it will result in the creation of a new aspect instance upon each execution of the `setBalance()` method, similar to the aspect instances shown in figure 4.4.

4.3.4 Implicit limiting of join points

Using the per-object or per-control-flow association has the side effect of implicitly limiting the advice in the aspect to only join points that match the scope of an aspect instance. The *scope* of an aspect instance is the set of join points that have an aspect instance associated with them. For example, for the `percflow()` association, the scope of an aspect instance is all the join points occurring inside the control flow of the specified pointcut. This means that even if a pointcut

specified for an advice matches a join point, the advice to that join point won't apply unless the join point also matches the scope of the aspect. This side effect often surprises developers when they refactor an aspect to create reusable parts and need to use `per-` associations.

The aspect association implies that advice in an aspect will apply to join points only if:

- For `perthis()` associations, the join point's execution object matches the aspect instance's associated object.
- For `pertarget()` associations, the join point's target object matches the aspect's associated object.
- For `percflow()` associations, the join point is in the control flow of the aspect's associated control flow.
- For `percflowbelow()` associations, the join point is below the control flow of the aspect's associated control flow.

A simple example, shown in listing 4.19, might illustrate this concept better.

Listing 4.19 TestAssociationScope.java

```
public class TestAssociationScope {
    public static void main(String[] args) {
        A a = new A();
        a.m();
    }
}

class A {
    public void m() {
        B b = new B();
        b.m();
    }
}

class B {
    public void m() {
    }
}

aspect TestAspect {
    before() : !within(TestAspect) {
        System.out.println(thisJoinPoint);
    }
}
```

When we compile and run this program, we see logging of each executed join point:

```
> ajc *.java
> java TestAssociationScope
staticinitialization(TestAssociationScope.<clinit>)
execution(void TestAssociationScope.main(String[]))
call(A())
staticinitialization(A.<clinit>)
preinitialization(A())
initialization(A())
execution(A())
call(void A.m())
execution(void A.m())
call(B())
staticinitialization(B.<clinit>)
preinitialization(B())
initialization(B())
execution(B())
call(void B.m())
execution(void B.m())
```

Now let's modify `TestAspect` to use the `perthis()` association.

```
aspect TestAspect perthis(execution(void A.*())) {
    before() : !within(TestAspect) {
        System.out.println(thisJoinPoint);
    }
}
```

When we compile and run this again, we see that only the methods that match the `execution(void A.*())` pointcut are advised:

```
> ajc *.java
> java TestAssociationScope
execution(void A.m())
call(B())
call(void B.m())
```

4.3.5 Comparing object association with member introduction

It is possible to avoid using the `perthis()`/`pertarget()` association with a judicious use of static crosscutting using introduced fields. In that case, instead of keeping the state in an aspect, you introduce that state to the object being aspected. This kind of modification often leads to simpler design. For example, consider this aspect, which associates an aspect instance with each `Account` object. The aspect's state—`_minimumBalance`—effectively becomes part of the `Account` object's state:

```
public aspect MinimumBalanceAspect perthis(this(Account)) {
    private float _minimumBalance;

    ... methods and advice using _minimumBalance
}
```

Now if we want to use member introduction instead of association, we can change the aspect in the following way:

```
public aspect MinimumBalanceAspect {
    private float Account._minimumBalance;

    ... methods and advice using _minimumBalance
}
```

In this snippet, we use the member introduction mechanism to associate a new member—`_minimumBalance`—with each `Account` object. The result is identical in both snippets—a new state is associated with each `Account` object.

Certain reusable aspects, such as cache management, that need to work with diverse types of objects may not have any common shared type. For example, `Customer` and `Account` probably have no class or interface common to their inheritance hierarchy. Therefore, to introduce a state, you will first need to specify a common type using a declare parent. For example, you can declare the interface `Cacheable` to be a parent type of `Account` and `Customer`. Then you may introduce the required state to `Cacheable`. This way, you get the same effect as per-object association using a simple introduction mechanism.

Developing reusable aspects using introduction instead of per-object association can get tricky. The main reason is that a reusable base aspect, unaware of the application-specific classes, cannot use the declare parents construct to specify a common type. While you can get around this issue by using a complex design, per-object association can offer an elegant alternative solution. When you're using per-object associations, the base aspect includes an abstract pointcut that associates the aspect with the object at the matching join points. Then, all that a derived aspect needs to do is provide a definition for that pointcut so that it captures join points whose associated objects need additional per-object state. Chapter 9 (section 9.7.2) provides a concrete example of the simplification of a reusable aspect using per-object association.

The choice between use of per-object association and member introduction is a balance between elegance and simplicity. Experience is usually the best guide.

4.3.6 Accessing aspect instances

Aspect instances are created automatically by the system according to the association specification. To access their state from outside the aspect, however, you will need to get its instance. For example, in a profiling aspect that collects duration for the execution of profiled methods, typically you would keep the profile data inside the profile aspect. When you need to retrieve this data, say from

another thread that will print the latest profile information, you have to get the aspect instance first. For all types of aspect associations, you can get the aspect instance using the static method `aspectOf()` that is available for each aspect. The method returns an instance of the aspect. For a profiler case, we could retrieve the data as follows:

```
Map profileData = ProfilerAspect.aspectOf().getProfileData();
```

If the `getProfileData()` method were static (which would require data returned to be marked static), we could directly access the data using `ProfilerAspect.getProfileData()` irrespective of the association specification. In certain cases, such as when using third-party aspects or aspects with a class as a base, it may not be possible to mark certain members static due to other design considerations. In any case, marking a certain state static for easy access may not be a good practice and may prevent reusability through the use of different aspect associations.

Each aspect contains two static methods—`aspectOf()` to obtain the associated aspect instance and `hasAspect()` to check if an instance is associated. For aspects with default and control-flow association, both these methods take no arguments, whereas for aspects with per-object association, these methods take one argument of type `Object` to specify the object for which the associated aspect instance is sought. In all cases, the `aspectOf()` method returns the instance of an aspect if one is associated; otherwise, it throws a `NoAspectBoundException`. The method `hasAspect()` returns true if an aspect instance is associated; otherwise, it returns false. Note that since an aspect instance with a control-flow-based association lives only during the control flow (or below, for `percflowbelow()`), you can get the aspect instance only in the control flow associated with the aspect.

4.4 Exception softening

Java specifies two categories of exceptions that can be thrown by a method: checked and unchecked exceptions. When an exception is checked, callers must deal with it either by catching the exception or by declaring that they can throw it. Unchecked exceptions, which can be either `RuntimeException` or `Error`, do not need to be dealt with explicitly. Exception softening allows checked exceptions thrown by specified pointcuts to be treated as unchecked ones. Softening eliminates the need to either catch the exception or declare it in the caller's method specification.

The softening feature helps to modularize the crosscutting concerns of exception handling. For example, you can soften a `RemoteException` thrown in a Remote Method Invocation (RMI)-based system to avoid handling the exception at each level. This may be a useful strategy in some situations. For instance, if you know that you are using local objects of RMI-capable classes that won't throw any `RemoteException`, you can soften those exceptions.

To soften exceptions, you use the declare soft construct that takes the following form:

```
declare soft : <ExceptionTypePattern> : <pointcut>;
```

If a method is throwing more than one checked exception, you will have to individually soften each one. In listing 4.20, the aspect declares the softening of an exception thrown by the `TestSoftening.perform()` method. The method now behaves as if it is throwing an `org.aspectj.lang.SoftException`, which extends `RuntimeException`.

Listing 4.20 TestSoftening.java: code for testing the effect of softening an exception

```
import java.rmi.RemoteException;

public class TestSoftening {
    public static void main(String[] args) {
        TestSoftening test = new TestSoftening();
        test.perform();
    }

    public void perform() throws RemoteException {
        throw new RemoteException();
    }
}
```

Compiling the `TestSoftening` class will result in a compiler error, since `main()` neither catches the exception nor declares that it is throwing that exception:

```
> ajc TestSoftening.java
F:\aspectj-book\ch04\section4.4\TestSoftening.java:6
➥ Unhandled exception type RemoteException
test.perform();
^^^^^^^^^^^^^^
```

Listing 4.21 shows `SofteningTestAspect`, which softens the `RemoteException` thrown by the join point that corresponds to the call to the `TestSoftening.perform()` method.

Listing 4.21 Softening aspect

```
import java.rmi.RemoteException;

public aspect SofteningTestAspect {
    declare soft : RemoteException : call(void TestSoftening.perform());
}
```

By softening the exception, we can compile the code without errors. When we run the program, we see a call stack due to a thrown `SoftException`:

```
> ajc TestSoftening.java SofteningTestAspect.java
> java TestSoftening
Exception in thread "main" org.aspectj.lang.SoftException
        at TestSoftening.main(TestSoftening.java:6)
```

An aspect declaring an exception for a join point wraps the join point execution in a try/catch block. The catch block catches the original exception, and the throw block throws a `SoftException` that wraps the original exception. This means that in listing 4.21, if we were to specify `execution` instead of `call` in the pointcut, the compiler would still give us a compiler error for the unhandled exception. To illustrate this, let's look at the code in listings 4.20 and 4.21 again. First let's see that compiling `TestSoftening` together with `SofteningTestAspect` results in a woven `TestSoftening` class that looks like the following:

```
import java.rmi.RemoteException;

public class TestSoftening {
    public static void main(String[] args) {
        TestSoftening test = new TestSoftening();
        try {
            test.perform();
        } catch (RemoteException ex) {
            throw new SoftException(ex);
        }
    }

    public void perform() throws RemoteException {
        throw new RemoteException();
    }
}
```

The portion marked in bold shows the effective code that was inserted due to `SofteningTestAspect`. As you see, the `RemoteException` is now caught by the `main()` method, which throws a `SoftException` wrapping the caught exception. Since the `SoftException` is an unchecked exception, `main()` no longer needs to declare that it can throw it.

Now, instead of the aspect in listing 4.20, let's apply the following aspect (which softens the exception at an `execution` pointcut rather than a `call` pointcut) to the original `TestSoftening` class:

```
public aspect SofteningTestAspect {
   declare soft : RemoteException : execution(void TestSoftening.perform());
}
```

Compiling this aspect with the `TestSoftening` class will result in woven code that looks like this:

```
import java.rmi.RemoteException;

public class TestSoftening {
    public static void main(String[] args) {
        TestSoftening test = new TestSoftening();
        test.perform();
    }

    public void perform() throws RemoteException {
        try {
            throw new RemoteException();
        } catch (RemoteException ex) {
            throw new SoftException(ex);
        }
    }
}
```

Here too, the portion marked in bold is the result of effective code added in the process of weaving. Since we have specified the softening of the execution of the `perform()` method, the try/catch is added to the `perform()` method itself. Note that although `perform()` would now never throw a `RemoteException`, its specification has not been altered, and therefore the compiler will complain that the `RemoteException` that may be thrown by `perform()` must be caught or declared to be thrown.

Exception softening is a quick way to avoid tangling the concern of exception handling with the core logic. But be careful about overusing this, because it can lead to masking off checked exceptions that you actually should handle in the normal way by making a conscious decision to handle the exception or propagate it to the caller. We will look at another pattern to handle exceptions in chapter 8.

4.5 Privileged aspects

For the most part, aspects have the same standard Java access-control rules as classes. For example, an aspect normally cannot access any private members of other classes. This is usually sufficient and, in fact, desirable on most occasions.

However, in a few situations, an aspect may need to access certain data members or operations that are not exposed to outsiders. You can gain such access by marking the aspect "privileged."

Let's see how this works in the following example. The `TestPrivileged` class (listing 4.22) contains two private data members.

Listing 4.22 TestPrivileged.java

```
public class TestPrivileged {
    private static int _lastId = 0;
    private int _id;

    public static void main(String[] args) {
        TestPrivileged test = new TestPrivileged();
        test.method1();
    }

    public TestPrivileged() {
        _id = _lastId++;
    }

    public void method1() {
        System.out.println("TestPrivileged.method1");
    }
}
```

Consider a situation where `PrivilegeTestAspect` (listing 4.23) needs to access the class's private data member to perform its logic.

Listing 4.23 PrivilegeTestAspect.java

```
public aspect PrivilegeTestAspect {
    before(TestPrivileged callee) : call(void TestPrivileged.method1())
                                    && target(callee) {
        System.out.println("<PrivilegeTestAspect:before objectId=\""
                           + callee._id + "\"");
    }
}
```

If we tried to compile this code, we would get a compiler error for accessing the `TestPrivileged` class's private member `_id`:

```
> ajc *.java
F:\aspectj-book\ch04\section4.5\PrivilegeTestAspect.java:7
```

```
➥ The field callee._id is not visible
+ callee._id + "\"");
  ^^^^^^^^^^

1 error
```

If, however, we mark the aspect as privileged (as follows), the code compiles without error and behaves as expected:

```
privileged public aspect PrivilegeTestAspect {
   ...
}
```

Now with the privileged aspect, we could access the internal state of a class without changing the class.

WARNING Privileged aspects have access to implementation details. Therefore, exercise restraint while using this feature. If the classes change their implementation—which they are legitimately entitled to do—the aspect accessing such implementation details will need to be changed as well.

4.6 Summary

Einstein said, "Keep things as simple as possible, but no simpler." The AspectJ concepts and constructs presented in this and the previous chapter are consistent with this advice. You can start writing crosscutting implementations of moderate complexity without using the advanced concepts presented in this chapter; however, you may eventually face situations that require the use of these more advanced constructs to simplify your implementation significantly.

The reflection support in AspectJ provides access to the join point's static and dynamic information through a small number of interfaces. This information can be used in logging to gain more insight into the system's inner workings. The dynamic and static information together can produce an enriched log output with a simple logging aspect.

Aspect-precedence control and aspect-association choices help manage complexity in systems that have a large number of aspects. As you begin to realize the benefits of aspect-oriented programming, you may find that you are implementing more aspects to handle typical crosscutting concerns that affect the same parts of the system, such as authorization and transaction management. Aspect precedence will help you coordinate these aspects so that they function correctly. The design and implementation of off-the-shelf reusable aspects will also benefit from the aspect-association feature. Developers will now be able to

create reusable aspects more effectively while knowing only minimal information about the target systems.

Using the privileged aspect feature will help in handling situations where you need to access the private members of classes. In this case, though, it is perhaps more important to understand the negative implications of using this technique.

These concepts, along with the ones presented in the earlier chapters, complete our introduction to the AspectJ language. Now that you have an understanding of the concepts and constructs in AspectJ, we are ready to dive into practical examples in areas such as logging, resource pooling, and authorization. The material presented in this and the two previous chapters will serve as a reference for you while reading the remainder of the book.