

AspectJ

IN ACTION

Practical Aspect-Oriented Programming

Ramnivas Laddad



 MANNING

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact:

Special Sales Department

Manning Publications Co.

209 Bruce Park Avenue

Greenwich, CT 06830

Fax: (203) 661-9018

email: orders@manning.com

©2003 by Manning Publications Co. All rights reserved.

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

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

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



Manning Publications Co.

209 Bruce Park Avenue

Greenwich, CT 06830

Copyeditor: Liz Welch

Typesetter: Denis Dalinnik

Cover designer: Leslie Haines

ISBN 1-930110-93-6

Printed in the United States of America

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

brief contents

PART 1 UNDERSTANDING AOP AND ASPECTJ..... 1

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

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

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

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

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

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

10

Authentication and authorization

This chapter covers

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

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

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

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

10.1 Problem overview

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

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

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

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

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

10.2 A simple banking example

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

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

Listing 10.1 Account.java

```
package banking;

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

```
    public void credit(float amount);

    public void debit(float amount)
        throws InsufficientBalanceException;

    public float getBalance();
}
```

Listing 10.2 `InsufficientBalanceException.java`

```
package banking;

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

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

Listing 10.3 `AccountSimpleImpl.java`

```
package banking;

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

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

    public int getAccountNumber() {
        return _accountNumber;
    }

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

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



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

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

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

Listing 10.4 `InterAccountTransferSystem.java`

```
package banking;

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

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

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

```
package banking;

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

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

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

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

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

Listing 10.6 `AuthLogging.java`: logging banking operations

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

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

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

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

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

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

10.3 Authentication: the conventional way

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

10.3.1 Implementing the solution

The authentication functionality in JAAS consists of the following:

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

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

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

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

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

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

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

Listing 10.7 Test.java: with authentication functionality

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

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

        lc.login();

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

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

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

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

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

10.3.2 Testing the solution

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

Listing 10.8 AuthLogging.java: with authentication logging implemented

```

package banking;

import org.aspectj.lang.*;

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

import logging.*;

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

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

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

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

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

```

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

```

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

<credit>
<debit>
<transfer>

```



```

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

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

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

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

```

3 Authentication advice

4 Authentication logic

5 Authentication exception

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

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

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

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

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

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

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

Listing 10.10 BankingAuthAspect.java: authenticating banking operations

```
package banking;

import auth.AbstractAuthAspect;

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

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

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

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

10.4.2 Testing the solution

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

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

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

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

10.5 Authorization: the conventional way

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

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

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

10.5.1 Understanding JAAS-based authorization

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

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

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

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

10.5.2 Developing the solution

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

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

```
package banking;

import java.security.*;

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

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

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

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

Listing 10.12 `AccountSimpleImpl.java`: the conventional way

```
package banking;

import java.security.AccessController;

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

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

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

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

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



```

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

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

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

```

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

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

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

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

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

10.5.3 Testing the solution

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

Listing 10.14 `AuthLogging.java`: adding authorization logging

```
package banking;

import org.aspectj.lang.*;

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

import logging.*;

public aspect AuthLogging extends IndentedLogging {
```



```

declare precedence: AuthLogging, *;

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

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

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

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

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

```

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

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

Listing 10.15 security.policy: the policy file for authorization

```

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

```

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

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

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

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

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

10.5.4 Issues with the conventional solution

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

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

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

10.6 Authorization: the AspectJ way

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

10.6.1 Developing the solution

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

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

Listing 10.16 `AbstractAuthAspect.java`: adding authorization capabilities

```
package auth;

import org.aspectj.lang.JoinPoint;

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

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

public abstract aspect AbstractAuthAspect {
```

```

private Subject _authenticatedSubject;

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

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

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

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

Object around()
: authOperations() && !cflowbelow(authOperations()) { ❸
    try {
        return Subject
            .doAsPrivileged(_authenticatedSubject,
                new PrivilegedExceptionAction() {
                    public Object run() throws Exception {
                        return proceed();
                    }
                }, null);
    } catch (PrivilegedActionException ex) {
        throw new AuthorizationException(ex.getException());
    }
}

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

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

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

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

```

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

5 Authorization exception

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

Let's examine the aspect in more detail:

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

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

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

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

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

Listing 10.17 `BankingAuthAspect.java`: adding authorization capabilities

```
package banking;

import org.aspectj.lang.JoinPoint;

import java.security.Permission;

import auth.AbstractAuthAspect;

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

```

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

```

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

10.6.2 Testing the solution

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

```

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

```

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

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

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

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

```
package banking;

import auth.AbstractAuthAspect;

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

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

```

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

```

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

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

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

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

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

10.7 Fine-tuning the solution

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

10.7.1 Using multiple subaspects

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

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

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

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

10.7.2 Separating authentication and authorization

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

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

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

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

10.8 Summary

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

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

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

AspectJ IN ACTION

AspectJ
v 1.1

Practical Aspect-Oriented Programming

Ramnivas Laddad

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

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

What's Inside

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

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

“Speaks directly to me as a developer”

—Alan Cameron Wills
fastnloose

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

—Chris Bartling, Identix, Inc.

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

—Arno Schmidmeier
AspectSoft

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

—Jean Baltus
Metafro-Infosys

www.manning.com/laddad

AUTHOR
ONLINE

Author responds to reader questions



Ebook edition available

9 781930 110939

ISBN 1-930110-93-6



MANNING

\$44.95 US/\$67.95 Canada