An introduction to AOP

Skill Level: Intermediate

Sing Li (westmakaha@yahoo.com)
Author
Wrox Press

30 Sep 2005

Follow along with Sing Li as he guides you through the basic concepts of aspect-oriented programming (AOP). AOP tools give you a way to separate the code for essential crosscutting concerns, such as logging and security, from your Java™ programs’ core application logic cleanly. AOP can make your code more readable, less error-prone, and easier to maintain.

Section 1. Before you start

About this tutorial

This tutorial introduces AOP and its basic concepts. AOP and its associated tools let you separate the code for essential crosscutting concerns, such as logging and security, from a program’s core application logic. AOP improves code quality by making it more readable, less error-prone, and easier to design and maintain.

This tutorial introduces you to:

- The reasons for AOP
- The basic concepts of AOP
- AOP tools and frameworks

Then, you’ll apply AOP techniques to a real-world code example.
The author would like to thank Mik Kersten for his expert comments and valuable review of the early drafts of this tutorial.

Prerequisites

This tutorial is for Java developers who want to explore AOP. You need to be comfortable with object-oriented design and Java programming. Experience working with security and logging systems is helpful but not necessary.

To run this tutorial’s sample code, you need the following:

- A working installation of JDK 1.4.2
- If you use the Eclipse IDE, the latest stable version of the AspectJ Development Tools (AJDT) plug-in, is available at http://www.eclipse.org/ajdt/downloads/
- If you do not use the Eclipse IDE, the stable release of AspectJ 1.2 (version 1.2.1 as of this tutorial’s publication date), is available at http://www.eclipse.org/aspectj/downloads.php#stable_release

Section 2. The need for AOP

As its name implies, aspect-oriented programming (AOP) is programming based on identifying and creating of aspects. You will learn about aspects later in this tutorial. In this section, you’ll gain an understanding of the concepts underlying AOP and how AOP complements object-oriented programming and design to make Java code easier to maintain and more robust.

Beyond object-oriented programming and design

AOP does not contradict or displace object-oriented design and programming, which all Java developers practice. Instead, it complements them. AOP can be viewed as the next natural adaptive step beyond object-oriented programming (OOP) in software engineering for developers and research communities.

Take a look at the code for AccountManager, shown in Listing 1. It is similar to typical production code you might find in an accounting system’s application logic.
Listing 1. The AccountManager class

```java
package com.ibm.dw.tutorial.aop.app;
import com.ibm.dw.tutorial.aop.logging.SystemLogger;
import com.ibm.dw.tutorial.aop.security.AuthorizationManager;
public class AccountManager {
    protected SystemLogger sysLog = SystemLogger.getInstance();
    protected AuthorizationManager authMgr =
        new AuthorizationManager();
    protected static final int ACCOUNT1 = 0;
    protected static final int ACCOUNT2 = 1;
    public static void main(String[] args) {
        AccountManager mgr = new AccountManager();
        mgr.transferFunds(ACCOUNT1, ACCOUNT2, 5);
    }
    public void transferFunds(int fromAcct, int toAcct, int amount) {
        sysLog.logDetails("transferFunds START - from Acct#" + fromAcct + " to Acct#" + toAcct + " for $" + amount);
        if (authMgr.verifyAccess(fromAcct) && authMgr.verifyAccess(toAcct)) {
            CustomerAccount from = findAccount(fromAcct);
            CustomerAccount to = findAccount(toAcct);
            from.debit(amount);
            sysLog.logDetails("transferFunds DEBIT - Acct#" + fromAcct + " for $" + amount);
            to.credit(amount);
            sysLog.logDetails("transferFunds CREDIT - Acct#" + toAcct + " for $" + amount);
        } else {
            sysLog.logDetails("Security VERIFY - access denied");
        }
        sysLog.logDetails("transferFunds END - from Acct#" + fromAcct + " to Acct#" + toAcct + " for $" + amount);
    }
    protected CustomerAccount findAccount(int acctNum) {
        // simple stub code for tutorial
        return new CustomerAccount(acctNum, 10);
    }
}
```

The core application logic in Listing 1 is the `transferFunds()` method. This method transfers funds between two accounts by debiting from one account and then crediting the other.

Note that:

- The `com.ibm.dw.tutorial.aop.accts.CustomerAccount` class groups a customer account's state and behaviors.
- The `com.ibm.dw.tutorial.aop.logging.SystemLogger` class encapsulates the details of the logging subsystem.
The com.ibm.dw.tutorial.aop.security.AuthorizationManager class is used to shield the logic from the complexity of the security subsystem.

You can see the full benefit of object-oriented design and programming at work here. Without object orientation, this code would be significantly more complex.

Mix of crosscutting concerns

Even though the code for AccountManager in Listing 1 is excellent object-oriented code, it looks and feels complicated and messy. Unfortunately, this is exactly the state of most of today's production code.

The code appears to be complex because calls to the logging system are interspersed throughout the code for the core application logic (for transferring funds between accounts). Calls to the authorization security subsystem are also embedded within the application logic.

Security and logging are concerns that are orthogonal to the application logic. The embedded code for these concerns distracts a developer or maintainer's attention from the application logic, and it makes the intent of the logic less apparent.

In AOP, orthogonal concerns such as security and logging are identified as commonly observed crosscutting concerns in systems. They are crosscutting in that they always cut across the multiple units of modularity (such as packages, classes, and code files) throughout the application. Even though they might not be part of the core business logic, crosscutting concerns are essential parts of the application.

Many other common crosscutting concerns have been identified in addition to security and logging. See the excellent research and articles library at the aods.net Community Wiki for further exploration (see Resources). You can also identify crosscutting concerns in your own problem domain, within your own software systems. A substantial portion of the work in AOP consists of identifying and encapsulating crosscutting concerns.

The transferFunds() method

The rest of this tutorial focuses on the transferFunds() method within the AccountManager code. Figure 1 shows the code for this method, with crosscutting concerns highlighted for easy identification.
Figure 1. The transferFunds() method

```java
public void transferFunds(int fromAcct,
                           int toAcct, int amount) {
    syslog.logDetails("transferFunds START - from Acct# " +
                       fromAcct + " to Acct# " + toAcct + " for $ " +
                       amount);
    if (authMgr.verifyAccess(fromAcct) &&
        authMgr.verifyAccess(toAcct)) {
        syslog.logDetails("Security VERIFY - access verified");
        CustomerAccount from = findAccount(fromAcct);
        CustomerAccount to = findAccount(toAcct);
        from.debit(amount);
        syslog.logDetails("transferFunds DEBIT - Acct# " +
                           fromAcct + " for $ " + amount);
        to.credit(amount);
        syslog.logDetails("transferFunds CREDIT - Acct# " +
                           toAcct + " for $ " + amount);
    } else {
        syslog.logDetails("Security VERIFY - access denied");
    }
    syslog.logDetails("transferFunds END - from Acct# " +
                      fromAcct + " to Acct# " + toAcct +
                      " for $ " + amount);
}
```

In Figure 1, the application-logic code is the code without highlighting. The code to control logging -- a crosscutting concern -- is highlighted in light orange. The code to implement security -- another crosscutting concern -- is highlighted in light green.

**Concern mixing: A system-wide, cross-modules problem**

If you were to examine all the source files for the application logic, the shuffling in of code from the two crosscutting concerns would be evident. Schematically, the
situation looks like Figure 2.

**Figure 2. Mixed concerns**

You can see in Figure 2 that although the security and logging subsystems are both well encapsulated using object-oriented design principles, the code within the application logic remains hopelessly intertwined with crosscutting concerns.

The situation depicted in Figure 2 is typically fairly extensive in a production system. Application logic with mixed concerns can involve hundreds or even thousands of classes and/or source files.

Although Figure 2 shades the logging and security subsystems in a solid color for simplicity and clarity, in production code crosscutting concerns permeate even these subsystems.

Problems that result from the intermixing of crosscutting concerns with core
application logic include:

- The code for the application logic is harder to maintain.
- The probability of coding error increases, and errors are more difficult to identify. (This is known as \textit{brittle code}.)
- Any changes in the interface or implementation of one of these subsystems can require changes in many source files.

\textbf{Difficulty of separating concerns}

Concern mixing occurs for two reasons:

- Object-oriented design provides partitioning and modularization mechanisms that are based only on related state/data and behavior.
- Existing programming languages and design methodologies provide no convenient means of easily separating crosscutting concerns.

Before AOP, programmers had no way to separate crosscutting concerns cleanly.

\textbf{Separation of concerns}

AOP lets you separate crosscutting concerns cleanly. The result is represented schematically in Figure 3.

\textbf{Figure 3. Separation of concerns}
In Figure 3, the intermixed crosscutting-concern code has been extracted out of the application logic. Now it is placed within the code module(s) of the associated subsystems instead. AOP lets you separate and organize the code that relates to crosscutting concerns. You can modularize this code by creating a related set of aspects, in a manner similar to grouping related states and behaviors within classes in object-oriented design. Now all the code associated with logging and security is contained within logging-related and security-related aspects and objects, not mixed throughout the system and polluting the application logic.

You'll apply AOP to the AccountManager code in the Applying AOP section later in this tutorial. The next section introduces you to aspects and the other fundamental concepts of AOP.
Section 3. Fundamentals of AOP

This section explains in more detail what a successful AOP solution must do and introduces you to the basic elements of AOP.

Calling subsystems without inline code

By applying AOP to AccountManager, you can completely eliminate the intermixed code for the crosscutting concerns (logging and security).

Listing 2 shows the AOP version of the code, called AccountManagerAOP.

Listing 2. The AccountManagerAOP class

```java
package com.ibm.dw.tutorial.aop.app;

public class AccountManagerAOP {

    protected static final int ACCOUNT1 = 0;
    protected static final int ACCOUNT2 = 1;

    public static void main(String[] args) {
        AccountManagerAOP mgr = new AccountManagerAOP();
        mgr.transferFunds(ACCOUNT1, ACCOUNT2, 5);
    }

    public void transferFunds(int fromAcct, int toAcct, int amount) {
        CustomerAccount from = findAccount(fromAcct);
        CustomerAccount to = findAccount(toAcct);
        from.debit(amount);
        to.credit(amount);
    }

    protected CustomerAccount findAccount(int acctNum) {
        // simple stub code for tutorial
        return new CustomerAccount(acctNum, 10);
    }
}
```

The clarity and understandability of the resulting AccountManagerAOP code is a significant improvement. It is now focused solely on the business logic of fund transfer. This code includes no intermixed security code or distracting embedded logging code.
AOP solutions

To factor out the crosscutting-concern code, an AOP-based solution must:

- Let you apply the code for crosscutting concerns into the execution flow of the business-logic code
- Enable the dynamically applied code to call the subsystems for the crosscutting concerns
- Enable packaging and organization of this additional code, without affecting the main application-logic code

You also need a way to:

- Describe where and when to apply the crosscutting code
- Specify what code to apply
- Ensure that the applied code has some access to the execution information (context information) at the point and time of application

This is exactly how today's AOP systems work. AOP operation is about run-time code interception (also called weaving). Using an AOP framework such as AspectJ (see Resources), you can intercept the flow of application execution and apply your own crosscutting code -- depending on dynamic conditions -- at run time.

You tell AspectJ exactly where and when to apply your code by specifying pointcuts, which select join points. The rest of this section explains these and other AOP concepts.

Elements of AOP: Join points

Join points are well-defined points during execution where you can apply crosscutting code. The available join points are dependent on the particular AOP framework or tool that you use. AspectJ has a rich, dynamic join-point model.

With AspectJ, join points are available when:

- Calling a method
- Executing a method
• Calling an exception handler
• Executing an exception handler
• Calling a constructor
• Executing a constructor
• Reading a field
• Writing a field

Note the differentiation between calling a method and executing a method. Join points associated with calling a method have access to the execution context information prior to the actual call of the method. Join points associated with executing a method have access to the execution context information inside the body of the method.

Other, more-advanced join points are also available. See the AspectJ documentation or AJDT documentation for more details (see Resources).

You cannot work with join points directly with AspectJ. Instead, you must select them using pointcuts.

Elements of AOP: Pointcuts

To specify where and when to apply your crosscutting code, you declare pointcuts. A pointcut selects a set of join points. Table 1 shows some of the more commonly used pointcuts in AspectJ.

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call(signature)</code></td>
<td>Select join points whenever the specified method or constructor is called</td>
</tr>
<tr>
<td><code>execute(signature)</code></td>
<td>Select join points whenever the specified method or constructor is executed</td>
</tr>
<tr>
<td><code>get(signature)</code></td>
<td>Select join points whenever the specified field is read</td>
</tr>
<tr>
<td><code>set(signature)</code></td>
<td>Select join points whenever the specified field is written to</td>
</tr>
<tr>
<td><code>handler(type-pattern)</code></td>
<td>Select join points whenever the exception handler associated with the Throwable type-pattern is executed</td>
</tr>
</tbody>
</table>
The signature and type-pattern in Table 1 can contain wildcard characters to match a range of methods, constructors, or subtypes.

In AspectJ, you declare pointcuts within aspects (see Elements of AOP: Aspects) or classes. You can also combine multiple pointcuts using the && and !! operators. You’ll see how to declare and combine pointcuts within an aspect in the Applying AOP section of this tutorial.

Elements of AOP: Advice

An advice ties the code that you want to apply together with the join points selected by your declared pointcut. You place the code that you want to execute inside the advice and then specify when you want it executed with respect to the matched join point. You have three general advice choices, shown in Table 2.

<table>
<thead>
<tr>
<th>Advice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>before()</td>
<td>Execute your code before the selected join point(s)</td>
</tr>
<tr>
<td>after()</td>
<td>Execute your code after the selected join point(s)</td>
</tr>
<tr>
<td>around()</td>
<td>Execute your code at the location of the join points, letting you wrap or skip the execution of the join point(s) if desired</td>
</tr>
</tbody>
</table>

Code that is affected by an advice, typically at matched join points, is said to be advised. AspectJ includes variations on the general advice types in Table 2 (see Resources). You’ll get some hands-on experience declaring advice in the Applying AOP section.

Elements of AOP: Inter-type declarations

AspectJ lets you add new members -- methods and fields -- to an existing Java class or type.

These additional declarations, contained within aspects, are called inter-type declarations. To the users of the affected classes, these newly declared members appear as if they are directly implemented by the original class or type.

The Applying AOP section shows how to create and use inter-type declarations.
Elements of AOP: Aspects

When you use AspectJ, the Java programming language is extended with the definitions of aspects. You define an aspect in much the same way that you define a class.

An aspect lets you bundle together:

- Pointcuts
- Advice
- Inter-type declarations

Like Java classes, aspects can have fields and methods (both static and nonstatic). You can create abstract aspects, just like classes. You can also extend classes, or aspects, to create new aspects.

You will see how to create aspects in the Applying AOP section.

Section 4. AOP frameworks and tools

The de facto standard tool for AOP is AspectJ, a seamless extension to the Java language developed and maintained by the Eclipse Foundation. The Eclipse Foundation's AspectJ Development Tools (AJDT) project provides Eclipse platform based tool support for AOP with AspectJ. Other alternative AOP frameworks and tools available are also available (see Resources for several examples).

AspectJ

You can learn AOP using AspectJ. AspectJ is an extension to the Java programming language that is easy to learn and use.

If you haven't already done so, download the latest stable version of AspectJ at http://www.eclipse.org/aspectj/downloads.php#stable_release. The code in this tutorial has been tested with AspectJ stable build version 1.2.1.

The AspectJ distribution contains a set of batch/script files:
• ajc runs the AspectJ compiler.
• ajbrowser runs the graphical structure browser for AspectJ.
• ajdoc runs the documentation generator for AspectJ.

This tutorial uses only the ajc compiler. Consult the AspectJ documentation for more information on the other tools (see Resources).

Setting up AspectJ

When you compile your AspectJ source code, run the ajc compiler batch file, instead of your regular javac compiler command. To make sure that you can run the ajc compiler, you must do the following:

1. Add the lib/aspectjrt.jar file to your CLASSPATH environment variable.
2. Make sure that the bin directory of the AspectJ distribution is in your PATH, so that your system can locate the ajc batch file.

AspectJ will weave in the bytecode required to run your aspects during the compilation process and class-loading time. To run any compiled AspectJ application, you just need to make sure that your CLASSPATH environment contains the lib/aspectjrt.jar file, and then use your regular java command to launch it.

AspectJ Development Tools

AspectJ Development Tools (AJDT) is a plug-in available for the Eclipse IDE. If you use the Eclipse IDE in your development process, you will find that AJDT can greatly simplify your work using AspectJ. You can find the latest version of AJDT at http://www.eclipse.org/ajdt/downloads/.

Figure 4 shows AJDT running in Eclipse. The arrow marks in the left margin of the code window indicate advised code.

Figure 4. The AJDT Eclipse plug-in
Section 5. Applying AOP

Now that you understand AOP's fundamental concepts, you're ready to get hands-on with AspectJ. This section will take you through the steps for creating aspects for the AccountManagerAOP class.

Creating a logging aspect
The first aspect you'll create implements logging. Aspects in AspectJ are contained in files with a .aj extension. The LogTransfer.aj file for the logging aspect is shown in Listing 3. You can find this file in the com.ibm.dw.tutorial.aop.app package of the sample code (see Download).

Listing 3. Logging aspect for AccountManagerAOP

```java
package com.ibm.dw.tutorial.aop.app;

import com.ibm.dw.tutorial.aop.logging.SystemLogger;

public aspect LogTransfer {
    SystemLogger sysLog = null;
    public LogTransfer() {
        sysLog = SystemLogger.getInstance();
    }

    pointcut transfer():
        call(* AccountManagerAOP.transferFunds(..));

    before(): transfer() {
        Object[] args = thisJoinPoint.getArgs();
        sysLog.logDetails("transferFunds START - from Acct#" +
            args[0] + " to Acct#" + args[1] + " for $" +
            args[2]);
    }

    after(): transfer() {
        Object[] args = thisJoinPoint.getArgs();
        sysLog.logDetails("transferFunds END - from Acct#" +
            args[0] + " to Acct#" + args[1] + " for $" +
            args[2]);
    }

    after(CustomerAccount acct, int amt): args(amt) &&
        target(acct) && within(AccountManagerAOP) &&
        call(* CustomerAccount.debit(..)) {
        sysLog.logDetails("transferFunds DEBIT - Acct#" +
            acct.getAcctNum() + " for $" + amt);
    }

    after(CustomerAccount acct, int amt): args(amt) &&
        target(acct) && within(AccountManagerAOP) &&
        call(* CustomerAccount.credit(..)) {
        sysLog.logDetails("transferFunds CREDIT - Acct#" +
            acct.getAcctNum() + " for $" + amt);
    }
}
```

Method-call join points and thisJoinPoint
The first step is to declare a pointcut named `transfer`, which selects join points whenever the `transferFunds()` method in the `AccountManagerAOP` class is called:

```java
pointcut transfer():
    call(* AccountManagerAOP.transferFunds(..));
```

In the `call()` pointcut, you specify the signature of the method to match. Note the use of wildcard characters (`*` and `..`) for the return type and the arguments.

You associate the `transfer` pointcut with the logging code by using a `before()` and an `after()` advice. Their declarations are shown in Listing 4.

**Listing 4. Advice declarations**

```java
before(): transfer() {
    Object[] args = thisJoinPoint.getArgs();
    sysLog.logDetails(
        "transferFunds START - from Acct#" +
        args[0] + " to Acct#" + args[1] + " for $" +
        args[2]);
}
```

```java
after(): transfer() {
    Object[] args = thisJoinPoint.getArgs();
    sysLog.logDetails(
        "transferFunds END - from Acct#" +
        args[0] + " to Acct#" + args[1] + " for $" +
        args[2]);
}
```

You can see the use of the special context variable called `thisJoinPoint` within the advice in Listing 4. `thisJoinPoint` has members and methods that let you access both dynamic and static context information at the matched join points. See the AspectJ documentation for information on the available members and methods (see Resources). In this case, you first use the `getArgs()` method to obtain the arguments used in calling the `transferFunds()` method. You can then use these argument values within the advice to create the log entry.

The advice in Listing 4 behaves exactly the same as the logging code in the original `AccountManager` class in Listing 1. It creates a log entry prior to the call of `transferFunds()` and another entry immediately after the call.

**Anonymous pointcuts, context-exposure pointcuts, and pointcut combinations**

Just as with non-AspectJ Java programming, you often have more than one way to
You can use a slightly different AspectJ technique to create log entries when the CustomerAccount class's debit() and credit() methods are called.

Instead of a named pointcut, like the transfer pointcut in the preceding subsection (selecting method-call join points and use of thisJoinPoint), an anonymous pointcut is used here. Anonymous pointcuts are unnamed and appear inline with the advice declaration, as in Listing 5.

Listing 5. Anonymous pointcuts

```java
after(CustomerAccount acct, int amt): args(amt) &&
  target(acct) && within(AccountManagerAOP) &&
  call(* CustomerAccount.debit(..)) {
    sysLog.logDetails(
      "transferFunds DEBIT - Acct#" +
      acct.getAcctNum() + " for $" + amt);
  }

after(CustomerAccount acct, int amt): args(amt) &&
  target(acct) && within(AccountManagerAOP) &&
  call(* CustomerAccount.credit(..)) {
    sysLog.logDetails(
      "transferFunds CREDIT - Acct#" +
      acct.getAcctNum() + " for $" + amt);
  }
```

The two after() advices in Listing 5 both use anonymous pointcuts declared inline and a combination of pointcuts. Note the use of && to combine multiple pointcuts. This means that the advised code will execute only for the join points matched by all the combined pointcuts.

The lexical pointcut, within(), ensures that only code within the AccountManagerAOP class will be advised. Because the AccountManager class is also in this package, this lexical pointcut prevents its call to the credit() and debit() methods within AccountManager from being accidentally advised.

Instead of using thisJoinPoint as it did in Listing 4, the advice obtains context information by matching special context-exposure pointcuts. The args(amt) pointcut matches arguments of the method call. The target(acct) pointcut matches the type of the object, in this case CustomerAccount, in which the matched method-call join point occurs.

You can access the context values matched by these context-exposure pointcuts as formal arguments to the advice. Note that the argument list of the after() advice declares the types of the matched identifiers. The acct and amt information, provided through these formal arguments, are available directly within the body of the advice. In the body of the advice, the code creates the associated debit and
credit logging entries using these matched arguments directly. There is no need for thisJoinPoint in this case.

Creating an authorization aspect

The AccessCheck aspect authorizes access to the bank accounts and is located in the AccessCheck.aj aspect file in the com.ibm.dw.tutorial.aop.app package.

This aspect tells AspectJ that before it transfers funds, it should verify with the authorization subsystem that the user has access rights to both accounts and then log the action. Listing 6 shows the code for this aspect.

Listing 6. The AccessCheck aspect

```java
package com.ibm.dw.tutorial.aop.app;
import com.ibm.dw.tutorial.aop.security.AuthorizationManager;

public aspect AccessCheck {
    private AuthorizationManager AccountManagerAOP.authMgr =
      new AuthorizationManager();

    public boolean AccountManagerAOP.check(
        int fromAcct, int toAcct) {
        return (authMgr.verifyAccess(fromAcct) &&
               authMgr.verifyAccess(toAcct));
    }

    pointcut startingTransfer(AccountManagerAOP mgr): target(mgr) &&
    call(void AccountManagerAOP.transferFunds(..));

    void around(AccountManagerAOP mgr): startingTransfer(mgr) {
        Object[] args = thisJoinPoint.getArgs();
        // unbox
        int acct1 = ((Integer) args[0]).intValue();
        int acct2 = ((Integer) args[1]).intValue();
        if ( mgr.check( acct1, acct2) ) {
            proceed(mgr);
        }
    }
}
```

Working with inter-type declarations

Implementation of the authorization aspect requires some thought. The implementation must check for access before the call to transferFunds(). If the access check fails, transferFunds() must not execute.

Because the AccountManagerAOP class has no code that verifies access, you use AspectJ's inter-type declaration to create a check() method for the AccountManagerAOP class. This method needs to have public access, because
your advice code will use it. Listing 7 shows the code for this declaration.

**Listing 7. Inter-type declaration**

```java
private AuthorizationManager AccountManagerAOP.authMgr =
    new AuthorizationManager();

public boolean AccountManagerAOP.check(int fromAcct, int toAcct) {
    return (authMgr.verifyAccess(fromAcct) &&
        authMgr.verifyAccess(toAcct));
}
```

Note also the inter-type declaration of the private `authMgr` member, which is local to the `AccountManagerAOP` class and used inside the `check()` method.

**Skipping join-point execution with the around() advice**

If you look back at the original `AccountManager` code in Listing 1, you'll notice that the check for access is performed via an enclosing `if()` statement. You can achieve an equivalent result by using the `around()` advice technique.

To match the calling of the `transferFunds()` method, you declare a pointcut named `startingTransfer()`. This pointcut has a formal argument, which is the `AccountManagerAOP` instance associated with the matched join point. You declare this pointcut as:

```java
pointcut startingTransfer(AccountManagerAOP mgr): target(mgr) &&
call(void AccountManagerAOP.transferFunds(..));
```

Now, to call the newly declared `check()` method, you use an `around()` advice with this pointcut. In the body of an `around()` advice, you can either:

- Call `proceed()` with the same arguments as the `around()` advice, to execute the matched join point.
- Do not call `proceed()`, thereby ensuring that the matched join point is not executed. If the join point matches a method call, the method will not be called.

In the `around()` advice code, the matched `transferFunds()` method is called (via `proceed()`) only if the `check()` method succeeds:

```java
void around(AccountManagerAOP mgr): startingTransfer(mgr) {
    Object[] args = thisJoinPoint.getArgs();
    // unbox
```
int acct1 = ((Integer) args[0]).intValue();
int acct2 = ((Integer) args[1]).intValue();
if ( mgr.check( acct1, acct2)) {
  proceed(mgr);
}

Note how the inter-type declared `check()` method is called in the advice, as if the original `AccountManagerAOP` class had defined it.

Advising an aspect

Last but not least, you must make sure that the success or failure of the authorization check is logged.

You cannot simply log the information, because doing so will introduce crosscutting-concern code into the `AccessCheck` aspect. That's correct -- crosscutting concerns such as logging often cut across not only classes, but also aspects.

To keep concerns separate, you must apply AOP one more time to advise the `AccessCheck` aspect. The `LogSecurity` aspect, part of the `com.ibm.dw.tutorial.aop.app` package, serves this purpose. Listing 8 shows this aspect.

Listing 8. The `LogSecurity` aspect

```java
package com.ibm.dw.tutorial.aop.app;

import com.ibm.dw.tutorial.aop.logging.SystemLogger;

public aspect LogSecurity {
  private SystemLogger sysLog = null;
  public LogSecurity() {
    sysLog = SystemLogger.getInstance();
  }
  pointcut inAccessCheck():
    execution(boolean AccountManagerAOP.check(..));
  after() returning(boolean ck): inAccessCheck() {
    if (ck)
      sysLog.logDetails("Security VERIFY - access verified");
    else
      sysLog.logDetails("Security VERIFY - access denied");
  }
}
```

In this aspect, the pointcut called `inAccessCheck()` matches the execution of the `check()` method of the `AccountManagerAOP` class. You declared this `check()` method, via inter-type declaration, in the `AccessCheck` aspect.
You use the `after()` returning() advice to capture the return value from the method call as a formal argument. The returned value from the `check()` method is available as a formal argument called `ck` within the body of the advice.

---

Section 6. Summary

Summary

AOP does not conflict with or displace object oriented design and programming; rather, it is a complementary technology. Crosscutting concerns cause code pollution across modular units that interferes with core business logic and creates tangled, difficult-to-maintain, and brittle code. Existing programming languages and design methodologies have no mechanisms to separate crosscutting concerns cleanly. AOP lets you encapsulate and organize code for crosscutting concerns in the form of aspects. You control when and where the crosscutting code executes by selecting join points with pointcuts. You use an advice to specify the code that is executed when join points are matched. Inter-type declarations let you add new fields and methods to existing classes as part of an aspect.

You have become familiar with the basic concepts of AOP and have explored them by working through an AspectJ example. You are now ready to investigate the possibilities of AOP and apply some of these concepts in your own day-to-day development tasks.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete AspectJ source code for this tutorial</td>
<td>j-aopintrocode.zip</td>
<td>10 KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

*Information about download methods*
Resources

Learn

- **AspectJ project Web site**: Code downloads, documentation, news, and community information for AspectJ.

- **AspectJ Development Tool (AJDT) site**: Eclipse-based plug-in and tips that can simplify your AOP life.

- "**New AJDT releases ease AOP development**" (Matt Chapman, developerWorks, August 2005): Details the latest releases of the AspectJ development tools for Eclipse 3.0 and 3.1, respectively.

- **PARC AspectJ information page**: A historical perspective of AspectJ, including information on its inventor and research-oriented heritage.

- **Aspect-Oriented Software Association**: Host of the annual International Conference on Aspect-Oriented Software Development and home to the aosd.net Community Wiki, a comprehensive source of information about AOP.

- **AOP@Work series**: Authored by recognized experts in the industry, this series is dedicated to helping you incorporate AOP into your day-to-day Java programming.

- "**AOP: Patching in the 21st Century**" Sing Li takes another approach to introducing AOP.

- Find more resources for Java developers in the developerWorks Java zone.

Get products and technologies

- Download the latest **AspectJ** stable release.

- Download the latest **AJDT** Eclipse plug-in.

- You will find alternative AOP frameworks and tools at the **AspectWerkz 2** site, the **Spring Framework** site, and the **JBoss AOP** site.

Discuss

- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Sing Li
Sing Li is a consultant and freelance writer. He has contributed to *Beginning JavaServer Pages*, *Professional Apache Tomcat 5*, *Pro JSP - Third Edition*, *Early Adopter JXTA*, *Professional Jini*, *Beginning J2ME: From Novice to Professional*, *Third Edition*, and numerous other books. He is also a regular contributor to technical magazines and an active evangelist of the VON and P2P evolutions. You can reach Sing at westmakaha@yahoo.com.