

2.2 Developing a Simple Aspect

Problem

You want to write a simple aspect.

Solution

First, write your business logic classes, as shown in Example 2-1.

Example 2-1. A simple business logic Java class

```
package com.oreilly.aspectjcookbook;

public class MyClass
{
    public void foo(int number, String name)
    {
        System.out.println("Inside foo (int, String)");
    }

    public static void main(String[] args)
    {
        // Create an instance of MyClass
        MyClass myObject = new MyClass();
        // Make the call to foo
        myObject.foo(1, "Russ Miles");
    }
}
```

Define an aspect that will be applied to this class. The aspect in Example 2-2 parodies the traditional “Hello World” for AspectJ by providing an aspect that captures all calls to the void `foo(int, String)` method in the `MyClass` class.

Example 2-2. A simple HelloWorld aspect in AspectJ

```
package com.oreilly.aspectjcookbook;

public aspect HelloWorld
{
    pointcut callPointcut() :
        call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));

    before() : callPointcut()
    {
        System.out.println(
            "Hello World");
        System.out.println(
            "In the advice attached to the call pointcut");
    }
}
```

Save this file in the same directory as your business logic class as *HelloWorld.aj* or *HelloWorld.java*. Run the `ajc` command to compile this simple application and produce the byte code `.class` files for both the aspect and the class:

```
> ajc -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% com/oreilly/aspectjcookbook/MyClass.java com/oreilly/aspectjcookbook/HelloWorld.java
```

If you get the following message then you will need to add the *aspectjrt.jar* to your classpath:

```
warning couldn't find aspectjrt.jar on classpath, checked:
```

```
error can't find type org.aspectj.lang.JoinPoint
```

```
1 error, 1 warning
```

To add the *aspectjrt.jar* to your classpath just for this compilation, type the following command to invoke the `ajc` compiler (Use `;` instead of `:` to separate the components of the classpath on Windows):

```
> ajc -classpath %MY_CLASSPATH%;%ASPECTJ_INSTALLATION_DIRECTORY%/lib/aspectjrt.jar -d %MY_DESTINATION_DIRECTORY% com/oreilly/aspectjcookbook/MyClass.java com/oreilly/aspectjcookbook/HelloWorld.java
```

The `ajc` compiler will produce two `.class` files; *MyClass.class* and *HelloWorld.class*. AspectJ 1.2 produces regular Java byte code that can be run on any 1.2 JVM and above, so you can now use the normal `java` command to run this application:

```
> java -classpath %MY_CLASSPATH% com.oreilly.aspectjcookbook.MyClass
Hello World
In the advice attached to the call point cut
Inside foo (int, String)
```

Congratulations! You have now compiled and run your first aspect-oriented application using AspectJ.

Discussion

This recipe has shown you your first example of an aspect and how AspectJ extends the Java language. At first, the new syntax can appear a little strange and a good portion of this book is dedicated to examining the ways the new language constructs can be used to create your aspects. To demystify some of this syntax up front, Example 2-3 briefly examines what each line of the aspect from this recipe specifies.

Example 2-3. A simple example of the new AspectJ syntax

```
1 package com.oreilly.aspectjcookbook;
2
3 public aspect HelloWorld
4 {
5     pointcut callPointcut() :
6         call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));
7 }
```

Example 2-3. A simple example of the new AspectJ syntax

```
8   before() : callPointcut()  
9   {  
10      System.out.println(  
11         "Hello World");  
12      System.out.println(  
13         "In the advice attached to the call pointcut");  
14   }  
15 }
```

Line 3 declares that this is an aspect.

Lines 5 and 6 declare the logic for a single named pointcut. The pointcut logic specifies that any join points in your application where a call is made to the void `MyClass.foo(int, String)` method will be caught. The pointcut is named `callPointcut()` so that it can be referred to elsewhere within the aspect's scope.

Lines 7 through 13 declare a single advice block. The `before()` advice simply states that it will execute before any join points that are matched by the `callPointcut()` pointcut. When a join point is matched the advice simply outputs a couple of messages to the system to inform you that the advice has been executed.

This recipe provides a good mechanism for testing a development environment to ensure that things are working as they should before performing any customization to the development tools.

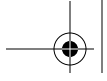


AspectJ aspects can be saved with the `.aj` or `.java` extension. The `ajc` tool compiles the file supplied, regardless of the extension. The different extensions, `.aj` and `.java`, are largely a matter of personal preference.

The compilation of the aspect and the Java class produces only `.class` files. This is a very important feature of AspectJ; aspects are treated as objects in their own right. Because of this treatment, they can be encoded as class files; this ensures that when the application is run, the Java Runtime Environment (JRE) does not need to understand any additional aspect-specific file formats. With the inclusion of the `aspectjrt.jar` support library in your JRE class path, an aspect-oriented software application can be deployed to any JRE on any platform in keeping with the 'Write Once, Run Anywhere' philosophy of Java.

See Also

Prior to using this recipe, it is necessary to get the AspectJ tools and prepare a simple command-line build environment as covered in Recipe 2.1; pointcuts are described in Chapters 4 through 12 and specifically the `call(Signature)` pointcut is examined in Recipe 4.1; the `within(TypePattern)` pointcut is described in Recipe 9.1; the NOT



(!) operator used in relation to pointcuts is described in Recipe 12.4; the before() form of advice can be found in Recipe 13.3.

