10
Aspect-oriented Programming with Spring

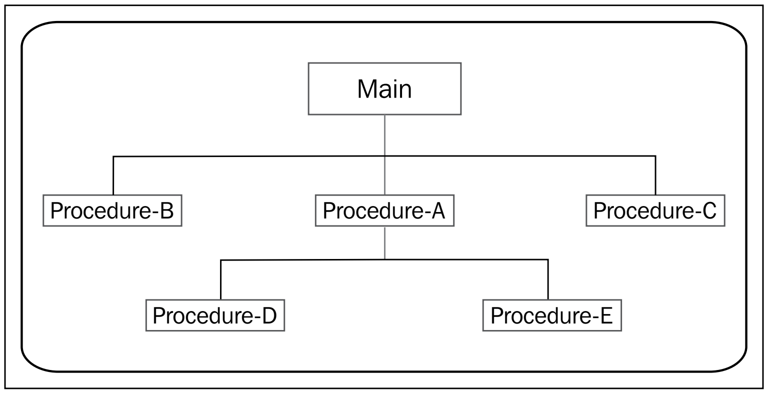
In this chapter, we will discuss the problems with using the **object-oriented programming** (**OOP**) methodology in developing enterprise applications, and how to solve these problems using **aspect-oriented programming** (**AOP**). You will learn the different terminology in AOP. Then, we will take a look at AOP support in Spring. We will look at defining and creating pointcuts. We will also discuss advice and their declarations using XML and annotation. We will learn about proxies and be introduced to AspectJ.

The list of topics covered in this chapter is as follows:

- Problems with OOP in developing applications
- Introduction to AOP
- AOP terminology
- AOP support in Spring
- XML schema-based configuration

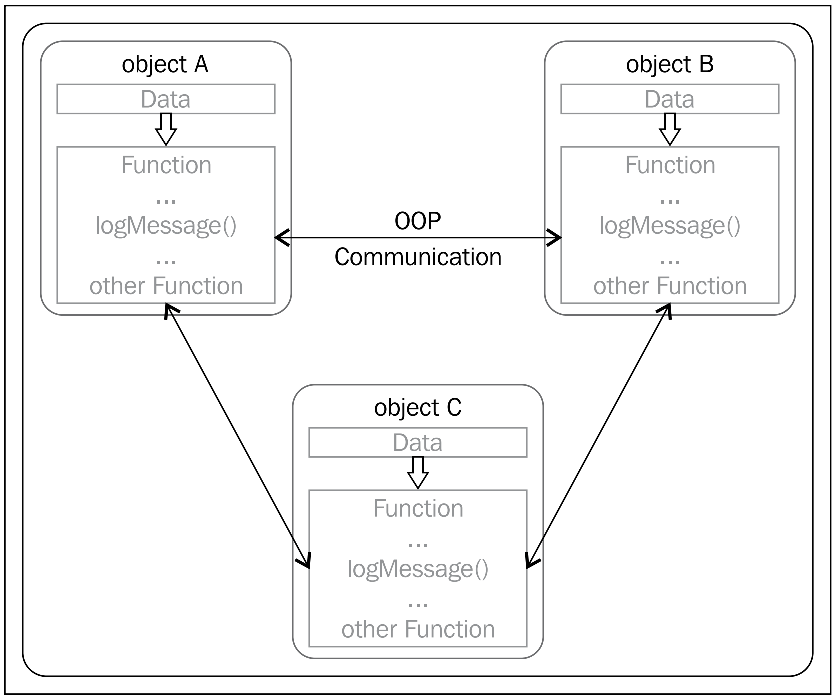
The problem with OOP in developing applications

Before we look into OOP, remember that, in the C programming language, we use procedural language. In C, we break down complex programs into few procedures. Each procedure specifies the step the program must take, in order to reach a desired state. The main procedure in the C programming language relates all other procedures as black box and calls other procedures without having information about how they are implemented.



Procedural language

The problem with procedural language is that if it consists of a complex design, then we will have a greater number of procedures and dependencies between procedures accordingly. So we opted for a better design to solve the problem using OOP. We encapsulated the data and associated functions inside a class that will form the blueprint. OOP can handle very complex programs.



Common procedure across different objects in OOP

OOP language uses many features present in procedural language. In OOP, we don't think of function when we are trying to solve a problem by writing code; we think of individual entities as objects and we would write objects. Each object has member variable and methods.

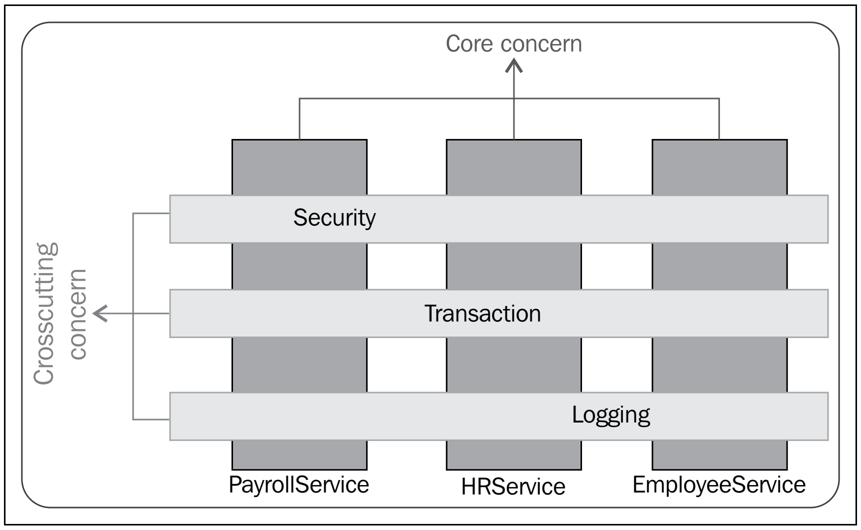
OOP looks good until now, but here is the problem: not all the tasks that you would want your program to do can be classified as objects. OOP allows us to modularize concerns into distinct packages, classes, and methods. Sometimes, it is difficult to place some concerns within methods or classes as they cross the class' and packages' boundaries. Let's take an example of security, which is a cross-cutting concern. Even though displaying and managing employees on the employee list page is the main purpose of the employee package in the payroll system, it needs to implement security in order to authenticate and authorize users to add and delete employees in the list. Other packages will also need to have the security functionality to protect the Create-Update-Delete functions of employees in the list.

One way to handle this requirement is to create a separate object for it. So we can have a security method to perform authentication and authorization within a new object called `UserSecurity` object. Then, other objects will not have that method anymore. Whenever a security method needs to be called, it would reference the `UserSecurity` object. So, in this design, the `UserSecurity` object will appear as an important object, as every other object that needs security is dependent on this `UserSecurity` object, which is not really a part of our business problem as it's just performing a support role, rather than a business role.

So, the main problem with OOP is that it has too many relationships to crosscutting objects.

Crosscutting objects refers to objects that concern other objects in your problem domain.

There are two types of concerns we generally have: the first is the primary concern, which is the core code itself, and then there is a secondary concern, for example, the payroll system with its payroll service, employee service, and HR service, is a primary concern. Adding security, transaction management, and logging are all secondary concerns, also called crosscutting concerns.



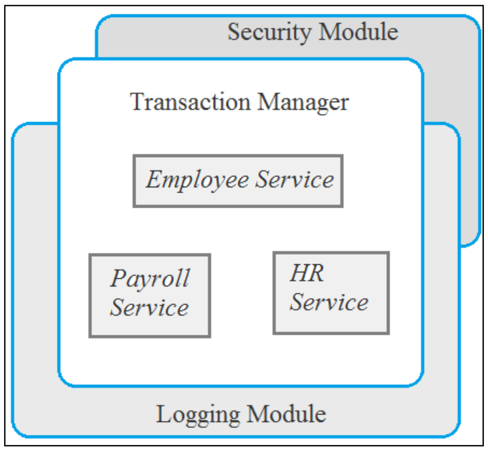
AOP is not just a feature that Spring provides; in fact it's a programming model itself. An aspect refers to a crosscutting concern, such as security, transaction, or logging, which is typically scattered across class hierarchies, and enables the modularization of concerns in the system. The class is the key unit of modularity in OOP, whereas the aspect is the key unit of modularity in AOP.

Introduction to aspect-oriented programming

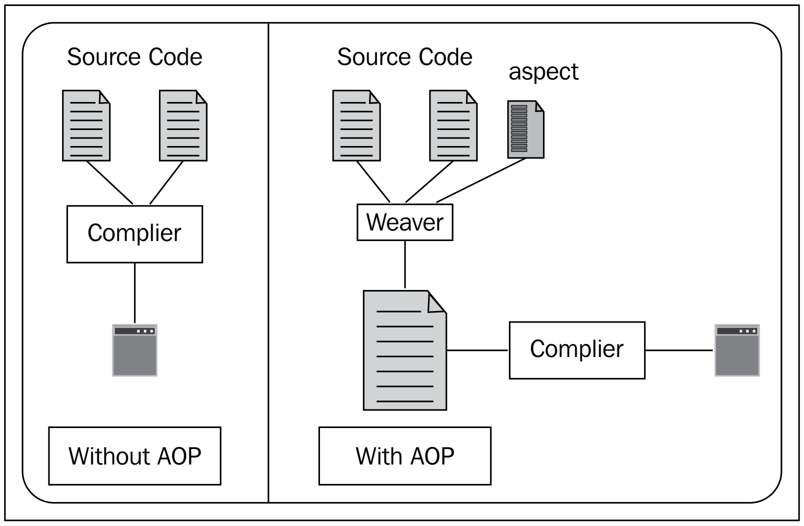
Aspect-oriented programming (**AOP**) is a promising technology to separate crosscutting concerns, something that is usually hard to do in OOP. AOP refers to the programming paradigm that isolates supporting functions from the main program's business logic.

It is used to provide declarative enterprise services in Spring, especially as a replacement for EJB declarative services. Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging, security, auditing, locking, and event handling. AOP is the methodology of applying middleware services such as security services, transaction management services, and so on, to a Spring application.

Let us consider a payroll management application (as shown in the following figure) where there will be employee service, HR service, and payroll service, which will perform some functional requirements in the system, such as adding/updating employee details, removing employees, browsing through employee-wise salary details, and many more functions. Implementing business functionality also requires taking care of non-functional requirements, such as role-based access to UI, maintaining logging details, and so on. AOP leaves the application component to focus on business functionality. Here, the core application implements business functionality and is covered with layers of functionality, such as security, logging, and transaction, provided by AOP:



You can add or remove aspects, as required, without changing any code. You can use the IoC container to configure the Spring aspects. Spring AOP includes advisors that contain advice and pointcuts filtering.



As shown in the preceding diagram, without AOP, we have source code that is given to the compiler and got executable. But with AOP, we now have source code and aspect, which are injected together as return in weaved class, which is given to the compiler which gives us executable.

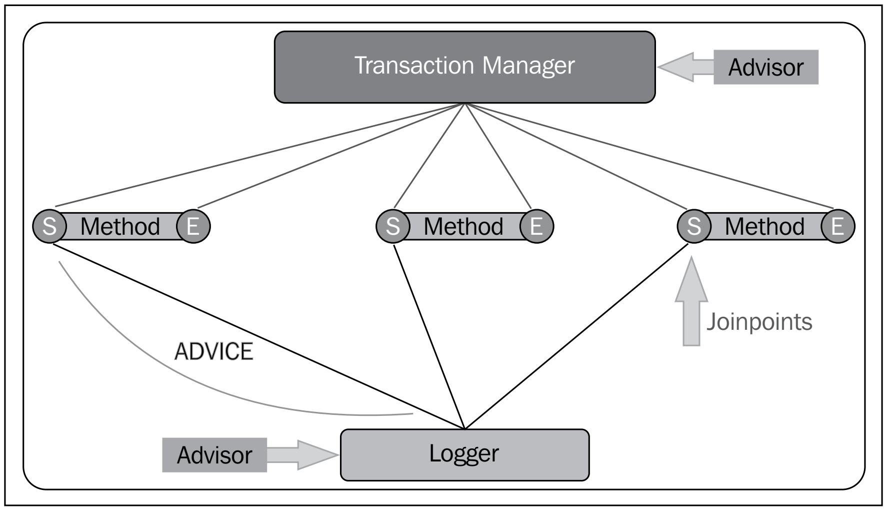
AOP terminology

Before we discuss AOP and its usage further, you should learn its terminology.

Joinpoint

A joinpoint refers to a point in the execution flow of an application, such as class initialization, object instantiation, method invocation, field access, or throwing an exception. Aspect code can be inserted at a joinpoint to add a new behavior into the application. The crosscutting concern is automatically added before/after a joinpoint by AOP.

As shown in the following diagram, we have methods within applications where the advice needs to be applied. **S** is the point before the method execution and **E** is the point after the method execution. These points are called joinpoints where the advices will be applied:



Advice

Advice represents code that is executed at a joinpoint. It includes API invocation to the system-wide concern.

As shown in the preceding diagram, we want to log the details of each and every method before it is invoked in the application, so we will choose the joinpoint **S** that is points before the method. Then, we also want to apply transaction functionality to all the methods before and after method execution. Here, **Logger** and **Transaction Manager** are called **advisors** and the code from advisor that is executed at these joinpoints is called advice.

There are different types of advice:

- **Before advice**: Before advice is executed before a joinpoint. Using it, we can execute the advisor code before the method is invoked.
- **After-returning advice**: After-returning advice is used to apply advice after the method successfully finishes its execution.
- **Throws advice**: During execution, when a method throws an exception, throws-advice can be applied.
- **Around advice**: Before and after a method execution, around-advice can be applied.

Pointcut

A pointcut represents a collection of joinpoints that specify where an advice is to be applied. For instance, in the preceding diagram, the collection of the joinpoints **S** and **E** would be a pointcut. The next step after identifying the joinpoints in the system is to apply the aspect. Instead of applying an aspect for each and every joinpoint, we can use pointcut to apply the aspect. For example, in the payroll system, the executions of all methods in classes that accept `employeeId` as a parameter can be considered to be pointcuts.

Aspect

The combination of pointcut and advice is referred to as an aspect, which is a crosscutting functionality that should be included in the application.

Introduction

An introduction allows you to introduce new methods or attributes to existing classes without having to change them, giving them a new behavior and state. By using an introduction, a specific interface can be implemented to an object, without needing that object's class to implement the interface explicitly.

Target

A target is either a class you write or a third-party class that is advised to add custom behavior. The target object is referred to as the advice object.

Proxy

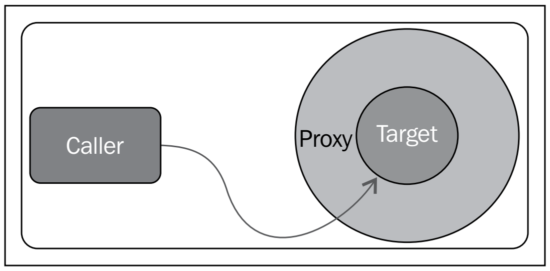
Proxy represents an object that is created after an advice is applied to the target object.

Weaving

Weaving is the process of inserting aspects into the application at the appropriate point. The weaving can take place at different stages in the target class's lifetime:

- **Compile time**: Injecting the byte code of the advice into the joinpoint during the compile time is called compile time weaving.
- **Classload time**: This injects the byte code at the when the class is loaded. During this, the byte code is injected into the loaded class to have the advice code at the joinpoint.
- **Runtime – Spring way**: The target object is shielded with the proxy bean, which is created by the Spring Framework. Whenever the caller calls the method on the target bean, the Spring Framework invokes the proxy and applies advices to the target method. Once the method execution is over, Spring applies advices to the target method again, if required, and the response returns to the caller.

Spring uses this kind of weaving. Runtime weaving is an effective method as it keeps code clean:



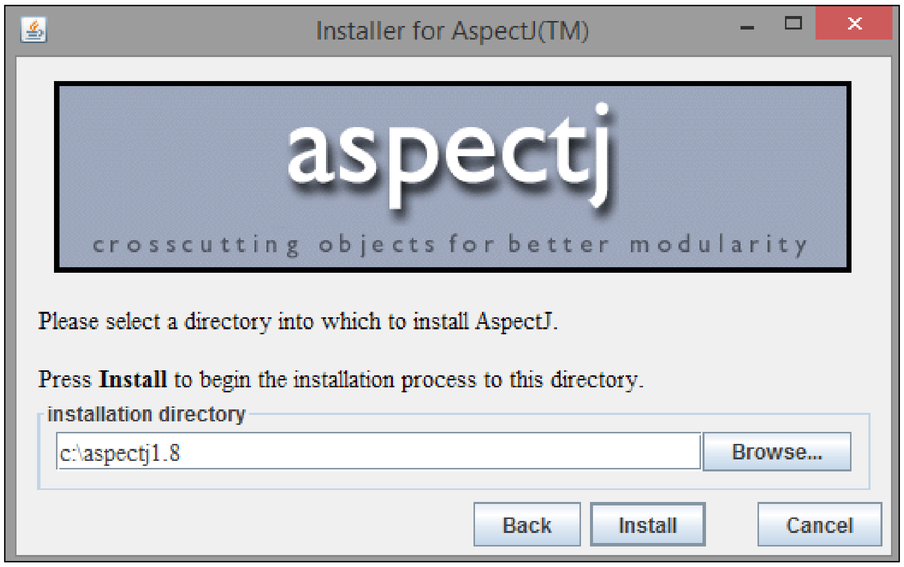
Weaver

The weaver is the actual processor that performs the weaving.

AOP implementations

AOP is implemented by a variety of frameworks, as explained here:

- **AspectJ**: AspectJ is well known in AOP language; it provides specialized syntax to express concerns. It also provides tools to add a concern into the system and enables crosscutting concern and modularization, such as logging, error checking and handling, and so on. It can be downloaded from `http://www.eclipse.org/aspectj/downloads.php`, where we can get the latest version of the jar file. In this book, we have used the latest version of AspectJ, which is 1.8. Double-clicking the downloaded jar file will automatically launch the installation window, where we will need to specify the installation directory by clicking on **Next**, as shown in the following screenshot:


Installer for AspectJ(TM)
aspectj
crosscutting objects for better modularity
Browse...
Back
Install
Cancel

Please select a directory into which to install AspectJ.

Press **Install** to begin the installation process to this directory.

installation directory: c:\aspectj1.8

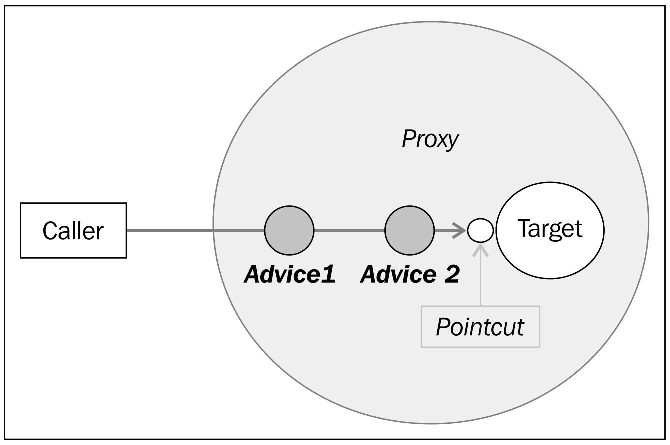
From the `C:\aspectj1.8\lib` folder, we can find `aspectjrt.jar` and `aspectjweaver.jar`, which we can add in the classpath of the project.

- **JBoss AOP**: JBoss provides its own AOP implementation known as JBoss AOP, which is a pure Java implementation.
- **Spring AOP**: Spring provides its own implementation of AOP, which is slightly different from the AspectJ implementation.

Let's discuss AOP support in the Spring Framework.

AOP support in Spring

The implementation of AOP in the Spring Framework is proxy-based and supports autoproxying. By autoproxying, we mean that if a bean is advised by one or more aspects, Spring will automatically generate a proxy for that bean to intercept the method invocation and ensure that the required advice is properly executed. The proxy is used to authenticate the access of the fields and methods. The request is first transferred to the proxy. Thereafter, it is the responsibility of the proxy to forward it to the concerned sources to process it. The principle of proxy design is to wrap the object with a proxy and use it as a substitution for the original object. Due to these features, proxies are used to implement crosscutting concerns in Spring AOP. The following figure represents a proxy in a Spring AOP implementation:



In the preceding figure, a method is called on a proxy-based system; method is called on the **Plain Old Java Object** (**POJO**). In the Spring Framework, POJO is extensively used as it neither enforces any particular interface to be implemented nor extends any class in order to create the class. Instead of a direct call to the object, the method is first called on the proxy, which forwards the method call to the actual object.

The Spring Framework doesn't follow any particular implementation, similar to the popular framework, AspectJ. A Spring programmer can select any implementation. The two Spring AOP implementations are:

- AspectJ annotation style
- Spring XML configuration style

Each implementation style has advantages and disadvantages. Next, we will discuss these implementations.

AspectJ annotation style

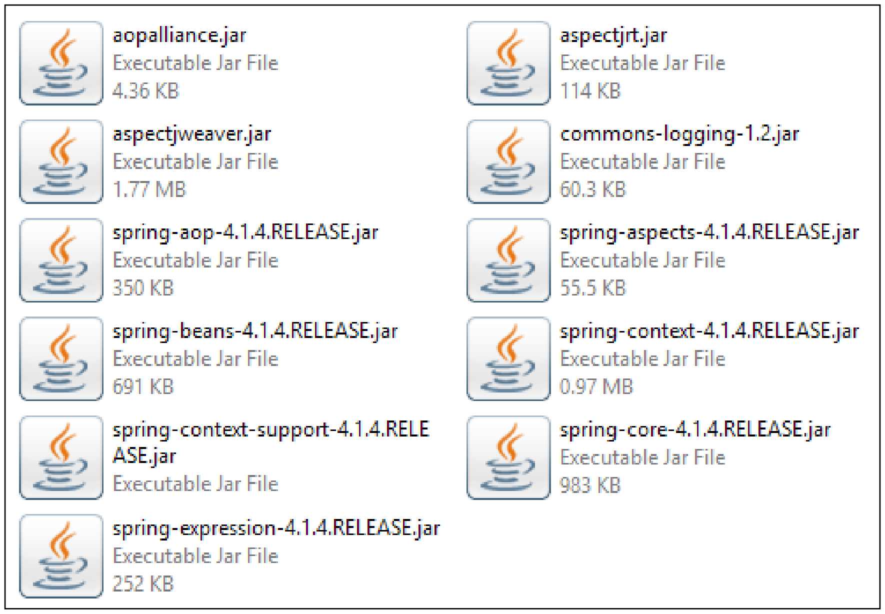
AspectJ provides the `@AspectJ` annotation. A class can be converted to an aspect by annotating `@AspectJ`. The following is the list of annotations declared in the `org.aspectj.lang.annotation` package:

AspectJ annotation	**Description**
`@Aspect`	This allows us to declare an aspect in a Java class
`@Pointcut`	This allows us to declare a pointcut along with an expression
`@Before`	This specifies the before-advice declaration
`@After`	This specifies the after-advice declaration
`@AfterReturning`	This specifies the after-returning advice declaration
`@AfterThrowing`	This specifies the after-throwing declaration
`@Around`	This specifies the around-advice declaration

We need to configure the Spring Framework before using the AspectJ annotation.

AspectJ annotation configuration in Spring

Some configuration is required to implement the `@AspectJ` annotation. The first step is to include JAR files in the classpath.


aopalliance.jar
Executable Jar File
4.36 KB

aspectjrt.jar
Executable Jar File
114 KB

aspectjweaver.jar
Executable Jar File
1.77 MB

commons-logging-1.2.jar
Executable Jar File
60.3 KB

spring-aop-4.1.4.RELEASE.jar
Executable Jar File
350 KB

spring-aspects-4.1.4.RELEASE.jar
Executable Jar File
55.5 KB

spring-beans-4.1.4.RELEASE.jar
Executable Jar File
691 KB

spring-context-4.1.4.RELEASE.jar
Executable Jar File
0.97 MB

spring-context-support-4.1.4.RELEASE.jar
Executable Jar File

spring-core-4.1.4.RELEASE.jar
Executable Jar File
983 KB

spring-expression-4.1.4.RELEASE.jar
Executable Jar File
252 KB

Next, add the AspectJ support to Spring using an auto-proxy declaration in the `<aop:aspectj-autoproxy/>` configuration file. The Spring Container creates proxies for the beans that match the aspect declaration details. Finally, add the AOP schema definition to the configuration file. The Spring configuration, file after proper configuration, looks similar to the following code snippet, which shows the contents of the `beans.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/aop http://www.
springframework.org/schema/aop/spring-aop-4.1.xsd
         http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context-
4.1.xsd">

   <aop:aspectj-autoproxy />
   <context:component-scan base-
package="org.packt.Spring.chapter4.aspectJ" />
</beans>
```

The preceding code snippet represents the configured XML file after the AOP schema and autoproxy declaration have been added. The `<aop:aspectj-autoproxy/>` syntax is used here to enable `@AspectJ` support in Spring.

Component scan is used in the `beans.xml` file to auto-detect the aspect like any other managed bean.

Now, after the initial setting has been made, we can declare the AOP concept using the `@AspectJ` annotation.

Declaring an aspect – @AspectJ

An aspect can be declared by annotating a POJO class with the `@Aspect` annotation and required to import `org.aspectj.lang.annotation.Aspect` package. The following code snippet represents the aspect declaration in the `@AspectJ` form:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {
   // ...
}
```

Here, an aspect is declared using the `@Aspect` annotation, which indicates that the class is being declared as an aspect. We have an annotated class with another annotation `@Component`, which allows the `LoggingAspect` class to be auto-detected. Otherwise we will have to configure it in XML like any other bean, as follows:

The following code snippet shows the `beans.xml` file:

```
<bean name="loggingAspect"
   class="org.packt.Spring.chapter4.aspectJ.aspect.
LoggingAspect" >
</bean>
```

Let's define the pointcut by using the `@Pointcut` annotation.

Declaring pointcut

A pointcut declaration determines the execution of a method on which an advice is applied. The pointcut is declared using the `@Pointcut` annotation defined under `org.aspectj.lang.annotation.Pointcut`. A pointcut declaration contains two parts:

- A pointcut expression which the methods that need to be executed
- A pointcut signature containing the name and any parameters

The following code snippet defines a pointcut named `serviceMethod()` that matches the execution of all methods in the classes under the `org.packt.Spring.chapter4.aspectJ.service` package:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {
   @Pointcut("execution(* org.packt.Spring.chapter4.aspectJ.
service.*.*(..))") //expression
   public void serviceMethod() {} // signature
}
```

In the preceding code snippet, a pointcut is declared:

- The `execution()` denotes which method execution we are interested in.
- `* org.packt.Spring.chapter4.aspectJ.service.*.*(..)`; here, the first `*`, which is a wildcard, denotes any return type of the matched method; the second `*` denotes any class under the `org.packt.Spring.chapter4.aspectJ.service package`; the third `*` denotes any method within the matched class.
- The `(..)` notation denotes that the method can have any set of parameters.

The following example defines a pointcut named `getEmpName` that matches the execution of the `getEmpName()` method available in the `Employee` class under the `org.packt.Spring.chapter4.aspectJ.model package`:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(*
org.packt.Spring.chapter4.aspectJ.model.Employee.getEmpName(..))")
   public void getEmpName() {} // signature
}
```

A pointcut is a point of execution at which the crosscutting concerns may be applied. Pointcuts are applied at joinpoints that match a method execution. Whenever the program execution reaches one of these joinpoints, an advice, that is, the code associated with the pointcut is executed. Spring AOP only supports joinpoints associated with a method execution for the Spring beans in its own container. If the pointcut expression is used outside this scope, an `IllegalArgumentException` exception is thrown.

Spring AOP supports many **Pointcut Designators** (**PCD**) that can be used in pointcut expression. However, Spring AOP users most often use the **execution** pointcut designator. The format of execution expression is as follows:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-
pattern? name-pattern(param-pattern)
          throws-pattern?)
```

Here, except `ret-type-pattern`, all other patterns are optional.

- `ret-type-pattern`: This pattern determines the return type of a method in order to match a joinpoint. Most frequently, we use `*` as `ret-type-pattern` to match any return type.
- `name-pattern`: This pattern matches the method name. We can use the `*` wildcard to match all methods.
- `param-pattern`: This pattern matches the method with parameters; for instance, `( )` matches a method that takes no parameters, whereas `(..)` matches any number of parameters.
- `modifiers-pattern`: This pattern defines the access modifier. In Spring AOP, because of the proxy-based nature, protected methods are not intercepted and hence match any given pointcut against public methods only.

Declaring advice

There are five kinds of advice in Spring AOP implementation. All these advices are declared as aspects and applied as conditions to deal with a specific instance of method execution. Advices can be classified as follows:

- Before
- After
- After returning
- After throwing
- Around

Before advice

A before advice takes place before a particular program/method is executed, or is executed before the joinpoint execution. The `@Before` annotation is used to handle crosscutting concerns before method execution.

The before advice declaration, declared in the `@Aspect` style, is represented in the following code snippet:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(* org.packt.Spring.chapter4.aspectJ.
service.*.*(..))")
   public void serviceMethod() {} // signature

   @Before("serviceMethod()")
   public void beforeLoginAdvice() {
         //...
   }
}
```

In the preceding code snippet, the annotation `@Before` is used to declare before advice under aspect declaration, and the advice refers to the pointcut declared separately. The `@Before` annotation makes the `beforeLoginAdvice` advice method to run before the `serviceMethod` target method, which is a pointcut. The `@Before` annotation takes the parameter where this advice method has to run before the target method runs. Here, the advice method will run before the pointcut `serviceMethod()` method is run.

After advice

An after advice takes place after a particular method is executed. The `@After` annotation is used to handle crosscutting concerns after method execution. The declaration of an after advice in `@Aspect` style is represented in the following code snippet:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(*
org.packt.Spring.chapter4.aspectJ.service.*.*(..))")
   public void serviceMethod() {} // signature

   @After("serviceMethod()")
   public void afterLoginAdvice() {
         //...
   }
}
```

In the preceding code snippet, the `@After` annotation is used to declare the after advice under aspect declaration, and the advice refers to the pointcut declared separately. The `@After` annotation makes the `afterLoginAdvice` advice method run after the `serviceMethod` target method, which is a pointcut. The `@After` annotation takes the parameter where this advice method runs after the target method runs. Here, the advice method will run after the pointcut `serviceMethod()` method is run.

The `@After` annotation executes the advice method after the method has run, whether the method has completed successfully or not because of some problem in execution of method and there is an exception thrown.

After returning advice

The `@AfterReturning` annotation is used to handle crosscutting concerns returned after method execution with normal termination. It is used to perform logging or any crosscutting concern only after returning a method.

As we have discussed, the `@After` annotation executes the advice method after the target method has run, whether the method has completed successfully or not. Now let's say we want little bit of control over this. We want an advice to run only when the target method completes successfully without throwing any exception.

The `@AfterReturning` annotation is another type of advice in Spring AOP from the `org.aspectj.lang.annotation.AfterReturning` package. The after returning advice declared in the `@Aspect` style is represented in the following code snippet:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(* org.packt.Spring.chapter4.aspectJ.
service.*.*(..))")
   public void serviceMethod() {} // signature

   @AfterReturning("serviceMethod()")
   public void afterReturningLoginAdvice() {
         //...
   }
}
```

In the preceding code snippet, the `@AfterReturning` annotation is used to declare `afterReturningLoginAdvice` under aspect declaration and the advice refers to the pointcut declared separately.

After throwing advice

The after throwing advice runs after the joinpoint execution completes with abnormal termination. The `@AfterThrowing` annotation is used to handle crosscutting concerns that are returned after method execution. The after throwing advice in the `@Aspect` style is represented in the following code snippet:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(* org.packt.Spring.chapter4.aspectJ.
service.*.*(..))")
   public void serviceMethod() {} // signature

   @AfterThrowing("serviceMethod()")
```

```
      public void afterThrowingLoginAdvice() {
             //...
      }
   }
```

In the preceding code snippet, the `@AfterThrowing` annotation is used to declare an after throwing advice under aspect declaration.

Around advice

So far, from the preceding topics, we have got an idea of how to write different advice types in different stages through the execution of the target method. We can write an advice method to be executed before a target method, or after a target method, or return of a target method so on. So, with all the concepts that we have discussed so far, we have all the tools required to intercept and run advice at any point in the execution of the target method needed, both before and after. There is one other advice type that we have not discussed so far, and that we will discuss here, the around advice.

The reason you would need around advice is if you happen to use all these different advice types, around advice is required for a particular use case. Say you have a target method and you want some advice code to be executed both before and after, you can achieve this by writing a separate advice method for before, and a separate advice for after. But if we use around, we have a little more control; it is also a little bit more powerful than two separate before and after advice methods.

The advice method used for this type of advice must be defined with the following:

- The first argument as of `org.aspectj.lang.ProceedingJoinPoint` type
- The return type should be `java.lang.Object`.

The `ProceedingJoinPoint` allows us to invoke the `proceed()` method that would proceed to invoke the target method. The around advice declared using the `@Around` annotation in the `@AspectJ` style, is represented in the following code snippet:

```
package org.packt.Spring.chapter4.aspectJ.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;
```

```
@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   @Pointcut("execution(* org.packt.Spring.chapter4.aspectJ.
service.*.*(..))")
   public void serviceMethod() {
   } // signature

   @Around("serviceMethod()")
   public Object aroundLoginAdvice(ProceedingJoinPoint
proceedingJoinPoint)
                throws Throwable {

          // Here is a code that get executed before target method

          Object result = proceedingJoinPoint.proceed();

         // Here is a code that get executed after target method

        return result;
   }
}
```

In the preceding code snippet, we have configured `aroundLoginAdvice` as around advice using the `@Around` annotation. The `proceedingJoinPoint.proceed()` method executes when the actual target methods are executed. So we can have code before `proceedingJoinPoint.proceed()` and after it gets executed, before and after the execution of the target method, and that makes this advice as around advice.

Proxy

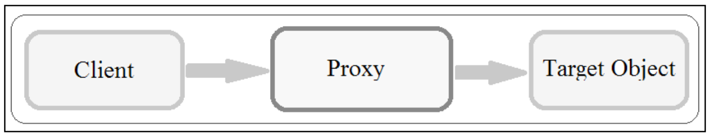
The proxy for the advised objects can be created by using the `org.aspectj.lang.aspectj.annotation.AspectJProxyFactory` class. The following code snippet can be used to create the proxy for the object:

```
AspectJProxyFactory factory = new
AspectJProxyFactory(targetObject);
```

Here, the `AspectJProxyFactory()` constructor is used to create a new proxy. It can be added to an aspect to authenticate the method call.

Here, we will examine AOP in more detail, considering mainly how it works in a Pure Object Oriented Scene. Let's say we have an `Employee.java` class with a `getEmpName()` method that is being called in the `main()` method after creating an `Employee` object in a `PayrollsSystem.java` class. Now, when we run the main method, how is it possible that some advice code, which has been defined in some other class, runs when we run the `getEmpName()` method? Let's say we have an around advice method in the `LoggingAspect.java` class that runs before and after the `getEmpName()` method, even though we don't have a call to that method anywhere in the `PayrollsSystem.java` class or the `Employee.java` class. How did that happen in the Pure Object Oriented paradigm?

AOP is completely different from OOP and the preceding scenario is actually not possible. Spring uses some of the concepts of OOP to achieve AOP, and it does so using a **Proxy**.



Proxy is a design pattern. It is a class that is going to limit access to another class. It may be done for security reasons, or because an object is intensive to create, or is accessed from a remote location. A proxy object is an object that is called by a client to access the real object. So instead of making a call to class A, a call is made to class B, and class B internally makes a call to class A.

When ApplicationContext in the Spring Framework starts up, it creates a bunch of proxies that track method invocation, and thus applies some special code before or/and after method execution. There are two types of proxies: one is the JDK proxy that comes out of the box in JDK, and second one is CGLib, which is from the CGLib library. The JDK proxy will support when our beans implement an interface, which is also the Spring way of using AOP. But, there are many people who like to go for concrete classes, and so they must use CGLib by adding `cglib.jar` to the classpath. This makes the proxy constructors run twice.

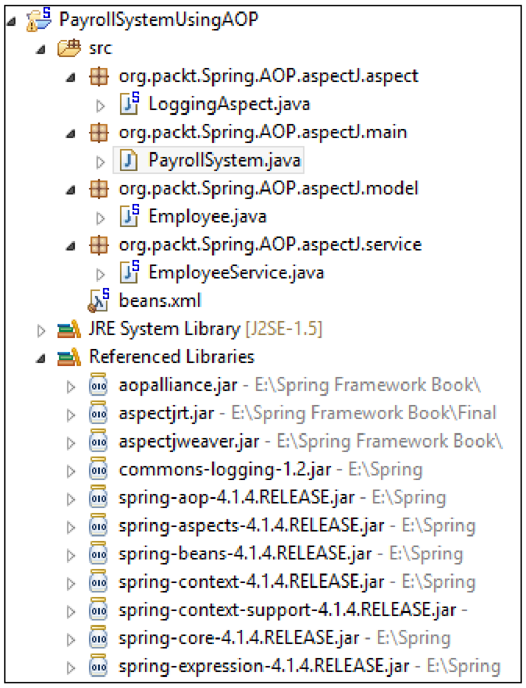
Developing an application in Spring using the @AspectJ annotation

After discussing `@AspectJ` support in the Spring Framework in detail, we will now develop an application using the `@Aspect` annotation that applies advice to a method execution. It loads the XML bean declaration from the `beans.xml` file. In order to create and execute the application successfully, we need to perform the following tasks:

1. Create Java files.
2. Create a Spring configuration file.
3. Configure the application.
4. Execute the application.

Directory structure of the application

The final directory structure of the application is seen in the following screenshot:



```
PayrollSystemUsingAOP
  src
    org.packt.Spring.AOP.aspectJ.aspect
      LoggingAspect.java
    org.packt.Spring.AOP.aspectJ.main
      PayrollSystem.java
    org.packt.Spring.AOP.aspectJ.model
      Employee.java
    org.packt.Spring.AOP.aspectJ.service
      EmployeeService.java
    beans.xml
  JRE System Library [J2SE-1.5]
  Referenced Libraries
    aopalliance.jar - E:\Spring Framework Book\
    aspectjrt.jar - E:\Spring Framework Book\Final
    aspectjweaver.jar - E:\Spring Framework Book\
    commons-logging-1.2.jar - E:\Spring
    spring-aop-4.1.4.RELEASE.jar - E:\Spring
    spring-aspects-4.1.4.RELEASE.jar - E:\Spring
    spring-beans-4.1.4.RELEASE.jar - E:\Spring
    spring-context-4.1.4.RELEASE.jar - E:\Spring
    spring-context-support-4.1.4.RELEASE.jar -
    spring-core-4.1.4.RELEASE.jar - E:\Spring
    spring-expression-4.1.4.RELEASE.jar - E:\Spring
```

The preceding tasks that enable the development of a Spring application using `@AspectJ` annotation have been discussed in the following sections.

Creating Java files

Create a project called `SpringAOPExample` and create a package called `org.packt.Spring.chapter4.aspectJ` under the `src` folder within the project. Create Java classes, namely `Employee.java`, `Logging.java`, and `PayrollsSystem.java` under the `org.packt.Spring.chapter4.aspectJ` package.

Employee.java

The `Employee.java` class contains `empName` as an instance variable and `getEmpName()` and `setEmpName()` as instance methods:

```
package org.packt.Spring.AOP.aspectJ.model;

public class Employee {
   private String empName;

   public String getEmpName() {
         return empName;
   }

   public void setEmpName(String empName) {
         this.empName = empName;
   }
}
```

EmployeeService.java

The `EmployeeService.java` class contains the `getEmployee()` and `printThrowException()` methods:

```
package org.packt.Spring.AOP.aspectJ.service;

import org.packt.Spring.AOP.aspectJ.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Autowired
   private Employee employee;

   public Employee getEmployee() {
         System.out.println("EmployeeService: employee returned");
```

```
            return employee;
      }

      public void printThrowException() {
            System.out.println("EmployeeService: Exception
occurred");
            throw new IllegalArgumentException(
                        "Throwing illegal argument exception from
Service class");
      }
}
```

LoggingAspect.java

The `LoggingAspect.java` class is an aspect module that defines methods to be called at various points:

```
package org.packt.Spring.AOP.aspectJ.aspect;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component("loggingAspect")
public class LoggingAspect {

   /**
   * Pointcut.
    */
   @Pointcut("execution(* org.packt.Spring.AOP.aspectJ.service.
EmployeeService.
getEmployee(..))")
   public void serviceMethod() {
   }

   /**
    * Execute before a selected method execution.
    */
   @Before("serviceMethod()")
   public void beforeAdvice() {
```

```
            System.out.println("Before Advice: Going to setup
Employee.");
   }

   /**
    * Execute after a selected method execution.
    */
   @After("serviceMethod()")
   public void afterAdvice() {
         System.out.println("Employee has been setup.");
   }

   /**
    * Execute if there is an exception raised by any method.
    */
   @AfterThrowing("execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.
printThrowException())")
   public void afterThrowingLoginAdvice() {
         System.out.println("There has been an exception: ");
   }
}
```

In the preceding code, we have used the `@Aspect`, `@Pointcut`, `@Before`, `@After`, and `@AfterThrowing` annotations in the class and method. The `@Aspect` declaration informs the compiler that an aspect has been declared.

Let's create the `main()` method in the `PayrollSystem.java` class.

PayrollsSystem.java

The `PayrollsSystem.java` class contains the main method and loads the XML bean definition from the `beans.xml` file:

```
package org.packt.Spring.AOP.aspectJ.main;

import org.packt.Spring.AOP.aspectJ.service.EmployeeService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {
          ApplicationContext context = new
ClassPathXmlApplicationContext(
                      "beans.xml");
          EmployeeService employeeService =
context.getBean("employeeService",
                      EmployeeService.class);
           System.out.println("Employee Name: "
                    + employeeService.getEmployee().getEmpName());
          try {
                 employeeService.printThrowException();
          } catch (IllegalArgumentException ex) {
               System.out.println(ex.getMessage());
          }
    }
}
```

The `BeanFactory` and `ApplicationContext` are two implementations of the IoC container. `ApplicationContext` is an interface, which is an extension of `BeanFactory`. Therefore, an object of the `ClassPathXmlApplicationContext` class needs to be created, which implements the `ApplicationContext` interface and builds the application context by loading the `beans.xml` file.

To get a bean declaration from an `ApplicationContext`, the `getBean()` method is executed in the `ApplicationContext` with a unique bean ID as its argument and class as another argument, which internally typecasts the `EmployeeService` type object.

Creating a Spring configuration file

Create the beans configuration file `beans.xml` under the `src` folder. The following code snippet shows the contents of the `beans.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.1.xsd">
```

```
    <aop:aspectj-autoproxy />
    <context:component-scan base-
package="org.packt.Spring.AOP.aspectJ" />
    <bean name="employee"
class="org.packt.Spring.AOP.aspectJ.model.Employee">
           <property name="empName" value="Ravi"></property>
    </bean>
</beans>
```

This configuration file contains an autoproxy declaration using the `<aop:aspectj-autoproxy />` tag.

Executing the application

Once you are done with creating source and bean configuration files, it's time to run the application:

```
INFO: Loading XML bean definitions from class path resource
[beans.xml]
Before Advice: Going to setup Employee.
EmployeeService: employee returned
Employee has been setup.
Employee Name: Ravi
EmployeeService: Exception occurred
There has been an exception:
Throwing illegal argument exception from Service class
```

XML schema-based configuration

In the previous sections, we have seen how to configure aspects and advice, and have done the entire configuration using annotation. We used the `@aspect` annotation to define an aspect, and we have seen all the different advice types using annotations such as `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, and `@Around`. We have also defined pointcut using the `@Pointcut` annotation, which can be applied to different advice methods.

In addition to defining aspects and advice methods using annotations, there is another way we can perform all these configurations and definitions in Spring XML. We will also discuss when to choose to define aspects using XML and when to define using annotations instead.

XML schema-based configuration is used to construct the aspect and declare advices. To declaring the aspect, pointcut and advice, Spring uses a bean configuration file along with specialized tags related to particular aspects.

In the XML schema-based configuration approach, all the aspect declarations are placed under the `<aop:config/>` tag. To use AOP namespace tags, we need to import the `spring-aop` schema.

The following code snippet shows the `beans.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">

   <aop:config>

          <!-- contains aspect configuration and all method
related configuration -->

   </aop:config>

</beans>
```

The `aop:config` will contain all aspect configurations and all specific method-related configurations, such as around, pointcut, and so on.

Declaring an aspect

An aspect is a regular Java object and is declared using the `<aop:aspect>` tag within `<aop:config>`. The ref attribute is used to refer backing bean. The following code snippet declares the `myaspect` aspect:

```
<aop:config>
   <aop:aspect id="loggingAspect" ref="loggingAspectBean">
         <!-- Body Content -->
   </aop:aspect>
</aop:config>

<bean id="loggingAspectBean"
class="org.packt.Spring.AOP.aspectJ.aspect.LoggingAspect" >
</bean>
```

In the preceding code snippet, an aspect declaration is given using the `<aop:aspect>` tag. The `<aop:aspect>` tag takes an ID and `ref`. The ID is the ID of the aspect. The `ref` is a reference to the bean that has been defined for the aspect. In the preceding code snippet, `loggingAspectBean` is defined as a bean, which is used as a reference for `<aop:aspect>`. So, this `<aop:aspect>` will replace the `@Aspect` annotation in the class.

The body content of the `<aop:aspect>` element allows zero or more child tags listed here, in any condition:

Tag	**Description**
`<aop:pointcut >`	This tag declares pointcut
`<aop:before >`	This tag declares before advice
`<aop:after >`	This tag declares after advice
`<aop:afterReturning >`	This tag declares after returning advice
`<aop:afterThrowing >`	This tag declares after throwing advice
`<aop:around >`	This tag declares around advice

Declaring a pointcut

A pointcut is declared by using the `<aop:pointcut>` tag. This pointcut ID is the method name in the class. A pointcut is declared either under the `<aop:aspect>` or `<aop:config>` tag:

- If a pointcut is declared under the `<aop:aspect>` tag, then it is visible only under that particular aspect within which it is declared. The following code snippet shows a pointcut declared within an aspect:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
         expression="execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.
getEmployee(..))"
          id="serviceMethod" />
</aop:aspect>
```

  The preceding code snippet defines a pointcut called `serviceMethod`, which has the expression `execution(* org.packt.Spring.AOP.aspectJ.service.EmployeeService.getEmployee(..))` within `loggingAspect`.

- Alternatively, if it is declared under the `<aop:config>` tag, then it is available to all aspects. The following code snippet shows a pointcut declared globally:

  ```
  <aop:config>
     <aop:pointcut
            expression="execution(*
  org.packt.Spring.AOP.aspectJ.service.EmployeeService.
  getEmployee(..))"
          id="serviceMethod" />
  </aop:config>
  ```

Declaring an advice

An advice is an object that includes API invocation to system wide concern representing the action to be performed at a joinpoint, specified by a pointcut. An advice is declared either under the `<aop: aspect>` or `<aop: config>` tag. The advice requires a reference to the particular pointcut, denoted by the `<ref>` tag. The common advices declared are:

- **Before advice**: This advice executes before the joinpoint.
- **After advice**: This advice executes after executing the joinpoint.
- **After returning advice**: This advice executes after the joinpoint execution completes, with normal termination.
- **After throwing advice**: This advice executes after the joinpoint execution, if it completes, with abnormal termination.
- **Around advice**: This advice surrounds the jointpoint providing the advice before and after executing the joinpoint, and controls the jointpoint invocation.

Now, let's discuss these advices in detail with some examples, first starting with before advice.

Before advice

As discussed earlier, before advice is the advice that executes before the jointpoint. That means this type of advice allows us to intercept the request and apply the advice before the execution of the joinpoint. A before advice takes place before a particular program/method is executed. It is used to handle crosscutting concerns before the method execution. Before advice is declared using the `<aop:before>` element within an `<aop:aspect>` tag.

The following code snippet represents the declaration of before advice in an XML schema-based configuration:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
          expression="execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.getEmployee(..))"
         id="serviceMethod" />
   <!-- before advice definition -->
   <aop:before method="beforeAdvice" pointcut-ref="serviceMethod"
/>
</aop:aspect>
```

In the preceding code snippet, the `<aop:before>` tag is used to declare a before advice under an aspect. The attributes of `<aop:before>` elements are:

- `pointcut-ref`: This takes the name of an associated pointcut definition, that is, the ID of the pointcut defined using the `<aop:pointcut>` element. Here, `serviceMethod` is the ID of a pointcut defined within `<aop:aspect>`.
- `method`: This takes the name of the method that has to be invoked before the target method invocation. Here, `<aop:before>` tells the `beforeAdvice` method to be executed before the execution of any getter method.

After advice

An after advice executes after a particular program/method is executed. The `<aop:after >` tag is used to handle crosscutting concern after method execution. The after advice executes after the joinpoint execution, irrespective of how the joinpoint execution exists, that is, whether it completes with normal or abnormal termination. Since this advice acts the same as finally block, that is, being executed in case of normal or abnormal termination of try block, and so considered as finally advice. The after advice declared in the XML schema-based configuration can be represented by the following code snippet:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
         expression="execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.getEmployee(.
.))"
        id="serviceMethod" />
   <!-- after advice definition -->
   <aop:after method="afterAdvice" pointcut-ref="serviceMethod"/>
</aop:aspect>
```

In the preceding code snippet, the `<aop:after>` tag is used to declare an after advice under the aspect. The attributes of `<aop:after>` elements are:

- `pointcut-ref`: This takes the name of an associated pointcut definition, that is, the ID of the pointcut defined using the `<aop:pointcut>` element. Here, `serviceMethod` is the ID of the pointcut defined inside `<aop:aspect>`.
- `method`: This takes the name of the method that has to be invoked after the target method invocation. Here, `<aop:after>` tells the `afterAdvice` method to be executed after the execution of getter method.

After returning advice

After returning advice is the advice that executes after the joinpoint invocation completes with normal termination. This advice is used to perform either logging or any crosscutting concern, only after returning a method. The `<aop:after-returning>` tag is used to handle crosscutting concerns, after returning from a method execution. The after returning advice that is declared in an XML schema-based configuration style is represented by the following code snippet:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
         expression="execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.
getEmployee(..))"
         id="serviceMethod" />
   <!-- after-returning advice definition -->
   <aop:after-returning method="afterReturningAdvice" pointcut-
ref="serviceMethod"/>
</aop:aspect>
```

In the preceding code snippet, the `<aop:after-returning>` tag is used to declare an `after-returning` advice under the aspect.

The attributes of `<aop:after-returning>` elements are:

- `pointcut-ref`: This takes the name of an associated pointcut definition, that is, the ID of the pointcut defined using the `<aop:pointcut>` element. Here, `serviceMethod` is the ID of the pointcut defined within `<aop:aspect>`.
- `method`: This takes the name of the `afterReturningAdvice` method that has to be invoked after the target method invocation completes with normal termination.

After throwing advice

After throwing advice is the advice that executes after the joinpoint invocation completes with an abnormal termination (that is, if the method invoked throws any exceptions). This advice is executed only after an exception is thrown under a method execution. The `<aop:after-throwing>` tag is used to declare an after throwing advice. The after throwing advice declared in an XML schema-based configuration style is represented by the following code snippet:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
          expression="execution(*
org.packt.Spring.AOP.aspectJ.service.EmployeeService.
getEmployee(..))"
         id="serviceMethod" />
   <!-- after-throwing advice definition -->
   <aop:after-throwing method="afterThrowingAdvice" pointcut-
ref="serviceMethod"/>
</aop:aspect>
```

In the preceding code snippet, the `<aop:after-throwing>` tag is used to declare an `after-throwing` advice under the aspect. The attributes of `<aop:after-throwing>` elements are:

- `pointcut-ref`: This takes the name of an associated pointcut definition, that is, the ID of the pointcut defined using the `<aop:pointcut>` element. Here, `serviceMethod` is the ID of the pointcut defined inside `<aop:aspect>`.
- `method`: This takes the name of the afterThrowingAdvice method that has to be invoked after the target method invocation completes with abnormal termination.

Around advice

As discussed earlier, the around advice can surround the joinpoint, providing the advice before and after executing the joinpoint. It can even control the joinpoint invocation. Around advice is used to specify the behavior of a method executed around a joinpoint. The `<aop:around>` tag is used to declare an around advice. The following code snippet represents the declaration of around advice in an XML schema-based configuration:

```
<aop:aspect id="loggingAspect" ref="loggingAspectBean">
   <!-- pointcut declared -->
   <aop:pointcut
```

```
          expression="execution(* org.packt.Spring.AOP.aspectJ.
service.EmployeeService.
getEmployee(..))"
         id="serviceMethod" />
   <!-- around advice definition -->
   <aop:around method="aroundAdvice" pointcut-
ref="serviceMethod"/>
</aop:aspect>
```

The attributes of `<aop:after-throwing>` elements are:

- `pointcut-ref`: This takes the name of an associated pointcut definition, that is, the ID of the pointcut defined using the `<aop:pointcut>` element. Here, `serviceMethod` is the id of the pointcut defined within `<aop:aspect>`.
- `method`: This takes the name of the `aroundAdvice` method that has to surround the target method invocation.

Developing an application in Spring using XML schema-based configuration

In this section, an aspect in the Spring AOP application is developed using XML schema-based configuration notation. It applies advice on a method execution. It loads the XML bean declaration from the `bean.xml` file. In order to create and execute the application successfully, we need to perform the following tasks:

1. Create Java files.
2. Create a Spring configuration file.
3. Configure the application.
4. Execute the application.

Creating Java files

To develop the application, we will create all the source files in the `src` folder under the `org.packt.Spring.chapter4.aspectJ` package.

Employee.java

The `Employee.java` class contains `empName` as an instance variable and `getEmpName()` and `setEmpName()` as an instance method:

```
package org.packt.Spring.AOP.aspectJ.model;

public class Employee {
   private String empName;

   public String getEmpName() {
         return empName;
   }

   public void setEmpName(String empName) {
         this.empName = empName;
   }
}
```

EmployeeService.java

The `EmployeeService.java` class contains the `getEmployee()`, `setEmployee()`, and `printThrowException()` methods:

```
package org.packt.Spring.AOP.aspectJ.service;

import org.packt.Spring.AOP.aspectJ.model.Employee;

public class EmployeeService {

   private Employee employee;

   public void setEmployee(Employee employee) {
         this.employee = employee;
   }

   public Employee getEmployee() {
         System.out.println("EmployeeService: employee returned");
         return employee;
   }

   public void printThrowException() {
         System.out.println("EmployeeService: Exception
occurred");
         throw new IllegalArgumentException(
                      "Throwing illegal argument exception from
Service class");
   }
}
```

LoggingAspect.java

The `LoggingAspect.java` class is an aspect module that defines the method that will be called at various points:

```
package org.packt.Spring.AOP.aspectJ.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

public class LoggingAspect {

   /**
    * Pointcut.
    */
   public void serviceMethod() {
   }

   /**
    * Execute before a selected method execution.
   */
   public void beforeAdvice() {
          System.out.println("Before Advice: Going to setup
Employee.");
   }

   /**
    * Execute after a selected method execution.
    */
   public void afterAdvice() {
         System.out.println("Employee has been setup.");
   }

   /**
   * Execute when any method returns.
   */
   public void afterReturningAdvice() {
         System.out.println("After returning advice run");
   }

   /**
    * Execute if there is an exception raised by any method.
    */
  public void afterThrowingAdvice() {
         System.out.println("There has been an exception: ");
   }
   /**
    * Execute around advice.
   */
   public Object aroundAdvice(ProceedingJoinPoint
proceedingJoinPoint)
                throws Throwable {
          System.out.println("Additional Concern Before calling
actual method");
          // Here is a code that get executed before target method
          Object result = proceedingJoinPoint.proceed();

         // Here is a code that get executed after target method
         System.out.println("Additional Concern After calling
actual method");
        return result;
   }
}
```

PayrollsSystem.java

The `PayrollsSystem.java` class contains the main method and loads the XML bean definition from the `beans.xml` file:

```
package org.packt.Spring.AOP.aspectJ.main;

import org.packt.Spring.AOP.aspectJ.service.EmployeeService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {
          ApplicationContext context = new
ClassPathXmlApplicationContext(
                      "beans.xml");
          EmployeeService employeeService = (EmployeeService)
context
                      .getBean("employeeService");
          System.out.println("Employee Name: "
                    + employeeService.getEmployee().getEmpName());
          try {
                employeeService.printThrowException();
          } catch (IllegalArgumentException ex) {
               System.out.println(ex.getMessage());
          }
   }
}
```

The `BeanFactory` and `ApplicationContext` are two implementations of the IoC container. `ApplicationContext` is an interface that is an extension of `BeanFactory`. Therefore, an object of the `ClassPathXmlApplicationContext` class needs to be created, which implements the `ApplicationContext` interface and build the application context by loading the `beans.xml` file.

To get a bean declaration from an application context, the `getBean()` method is executed in the application context with a unique bean ID as its argument, and class as another argument, which internally typecasts the `Employee` type object.

Creating a Spring configuration file

The Spring configuration file `beans.xml` includes all the configuration for the beans, aspects, pointcuts, and advices. The aspect is declared under the `<aop:config>`, using the `<aop:aspect>` tag. Create a beans configuration file called `beans.xml` under the `src` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">

   <aop:config>
        <aop:aspect id="loggingAspect" ref="loggingAspectBean">
                <!-- pointcut declared -->
                <aop:pointcut
                      expression="execution(*
org.packt.Spring.AOP.aspectJ.service.*.*(..))"
                     id="serviceMethod" />

                <!-- before advice definition -->
                <aop:before method="beforeAdvice" pointcut-
ref="serviceMethod" />

                <!-- after advice definition -->
                <aop:after method="afterAdvice" pointcut-
ref="serviceMethod" />

                <!-- after-returning advice definition -->
               <aop:after-returning method="afterReturningAdvice"
                      pointcut-ref="serviceMethod" />
               <!-- after-throwing advice definition -->
              <aop:after-throwing method="afterThrowingAdvice"
                      pointcut-ref="serviceMethod" />

               <!-- around advice definition -->
               <aop:around method="aroundAdvice" pointcut-
ref="serviceMethod" />
         </aop:aspect>
   </aop:config>

   <bean id="loggingAspectBean" class="org.packt.Spring.AOP.aspectJ.
aspect.LoggingAspect" />

   <bean name="employee" class="org.packt.Spring.AOP.aspectJ.model.
Employee">
         <property name="empName" value="Ravi"></property>
   </bean>

   <bean id="employeeService"
class="org.packt.Spring.AOP.aspectJ.service.EmployeeService"
          autowire="byType" />

</beans>
```

Executing the application

Once you are done with creating the source and bean configuration files, let's run the application:

```
INFO: Loading XML bean definitions from class path resource [beans.
xml]
Before Advice: Going to setup Employee.
Additional Concern Before calling actual method
EmployeeService: employee returned
Additional Concern After calling actual method
After returning advice run
Employee has been setup.
Employee Name: Ravi
Before Advice: Going to setup Employee.
Additional Concern Before calling actual method
EmployeeService: Exception occurred
There has been an exception:
Employee has been setup.
Throwing illegal argument exception from Service class
```

Summary

In this chapter, we saw the problems with OOP in developing applications. We looked at AOP concepts and their terminologies, such as aspect, joinpoint, pointcuts, advice, introduction, and proxy. We implemented AOP in a Spring application using the AspectJ annotation style and also using XML schema-based configuration.