2 - Introducing AspectJ

AspectJ is an aspect-oriented extension to the Java programming language, every valid Java program is also a valid AspectJ program. An AspectJ compiler produces class files that conform to the Java byte-code specification, allowing any compliant Java virtual machine to execute those class files.

The AspectJ compiler uses the modules containing the weaving rules, which address the crosscutting concerns, to add new behavior into the modules that address the core concerns, all without making any modifications to the core modules' source code; the weaving occurs only in the byte code that the compiler produces.

Crosscutting in AspectJ

In AspectJ, the implementation of the weaving rules by the compiler is called crosscutting; the weaving rules cut across multiple modules in a systematic way in order to modularize the crosscutting concerns. AspectJ defines two types of crosscutting: **static** crosscutting and **dynamic** crosscutting.

Dynamic crosscutting

Dynamic crosscutting is the weaving of new behavior into the execution of a program. Most of the crosscutting that happens in AspectJ is dynamic.

Dynamic crosscutting augments or even replaces the core program execution flow in a way that cuts across modules, thus modifying the system behavior. For example, if you want to specify that a certain action be executed before the execution of certain methods in a set of classes, you can just specify the weaving points and the action to take upon reaching those points in a separate module.

Static crosscutting

Static crosscutting is the weaving of modifications into the static structure of the system (classes, interfaces, aspects).

By itself, it does not modify the execution behavior of the system. The most common function of static crosscutting is to support the implementation of dynamic crosscutting. For instance, you may want to add new data and methods to classes and interfaces in order to define class-specific states and behaviors that can be used in dynamic crosscutting actions. Another use of static crosscutting is to declare compile-time warnings and errors across multiple modules.

Crosscutting elements

Join point

A **join point** is an identifiable point in the execution of a program. It could be a call to a method or an assignment to a member of an object. In AspectJ, everything revolves around join points, since they are the places where the crosscutting actions are woven in.

Let's look at some join points in this code snippet:

```
public class Account {
    ...

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

The join points in the `Account` class include the execution of the `credit()` method and the access to the `_balance` instance member.

Pointcut

A **pointcut** is a program construct that selects join points and collects context at those points.

For example, a pointcut can select a join point that is a call to a method, and it could also capture the method's context, such as the target object on which the method was called and the method's arguments.

We can write a pointcut that will capture the execution of the `credit()` method in the `Account` class shown earlier:

```
execution(void Account.credit(float))
```

To understand the difference between a join point and pointcut, think of pointcuts as specifying the weaving rules and join points as situations satisfying those rules.

Advice

Advice is the code to be executed at a join point that has been selected by a pointcut. Advice can execute before, after, or around the join point.

Around advice can modify the execution of the code that is at the join point, it can replace it, or it can even bypass it. Using an advice, we can log a message before executing the code at certain join points that are spread across several modules. The body of advice is much like a method body, it encapsulates the logic to be executed upon reaching a join point.

Using the earlier pointcut, we can write advice that will print a message before the execution of the `credit()` method in the `Account` class:

```
before() : execution(void Account.credit(float)) {
    System.out.println("About to perform credit operation");
}
```

Pointcuts and advice together form the dynamic crosscutting rules. While the pointcuts identify the required join points, the advice completes the picture by providing the actions that will occur at the join points.

Introduction

The **introduction** is a static crosscutting instruction that introduces changes to the classes, interfaces, and aspects of the system. It makes static changes to the modules that do not directly affect their behavior.

For example, you can add a method or field to a class. The following introduction declares the `Account` class to implement the `BankingEntity` interface:

```
declare parents: Account implements BankingEntity;
```

Compile-time declaration

The compile-time declaration is a static crosscutting instruction that allows you to add compile-time warnings and errors upon detecting certain usage patterns.

The following declaration causes the compiler to issue a warning if any part of the system calls the `save()` method in the `Persistence` class. Note the use of the `call()` pointcut to capture a method call:

```
declare warning : call(void Persistence.save(Object))
: "Consider using Persistence.saveOptimized()";
```

Aspect

The **aspect** is the central unit of AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Pointcuts, advice, introductions, and declarations are combined in an aspect. In addition to the AspectJ elements, aspects can contain data, methods, and nested class members, just like a normal Java class.

We can merge all the code examples from this section together in an aspect as follows:

```
public aspect ExampleAspect {
    before() : execution(void Account.credit(float)) {
        System.out.println("About to perform credit operation");
    }

    declare parents: Account implements BankingEntity;

    declare warning : call(void Persistence.save(Object))
        : "Consider using Persistence.saveOptimized()";
}
```

Let's take a look at how this all functions together

When you're designing a crosscutting behavior, the first thing you need to do is identify the join points at which you want to augment or modify the behavior, and then you design what that new behavior will be. To implement this design, you first write an aspect that serves as a module to contain the overall implementation. Then, within the aspect, you write pointcuts to capture the desired join points. Finally, you create advice for each pointcut and encode within its body the action that needs to happen upon reaching the join points. For certain kinds of advice, you may use static crosscutting to support the implementation.

AspectJ: under the hood

Since the byte code produced by the AspectJ compiler must run on any compliant Java VM, it must adhere to the Java byte-code specification. This means any crosscutting element must be mapped to one of the Java constructs.

- Aspects are mapped to classes
- Advice is usually mapped to one or more methods
- Pointcuts are intermediate elements that instruct how advice is woven and usually aren't mapped to any program element
- Introductions are mapped by making the required modification
- Compile-time warnings and errors have no effect on byte code. They simply cause the compiler to print warnings or abort the compilation when producing an error.

Exposed join point categories

AspectJ exposes several categories of join points:

- Method join points
- Constructor join points
- Field access join points
- Exception handler execution join points
- Class initialization join points
- Object initialization join points
- Object pre-initialization join points (rarely used)
- Advice execution join points