ASPECTJ

A Module for Beginner

Abstract

Aspect Oriented Programming (AOP) extension to Java

Kenrick Satrio

Table of Content

Contents

Chapter 1

Introduction to Aspect Oriented Programming

Aspect oriented Programming or for short AOP is a programming technique that cleanly modularize component or function by separates cross-cutting concerns which result in a better code quality like modularity, reusability, readability and correctness. AOP is not a programming language, but it's a compliment to other existing programming language such as Object Oriented Programming. Considering a problem that might occured, when we have two or more different behaviour that we want to compose into one function, it will resulting in a tangling code which hard for maintainabilty, understandbility, etc. For example a very basic error handling, the behaviour of handle an error and the behaviour of particular method execution is different, so it's cross-cutting each other.

In AOP, we separte cross-cutting behaviours from the component into a modularize unit, called aspect. In our previos example, the error handler code will be just written in one aspect, then apply the aspect into other components. It makes the code easier to maintain, future modification at the error handler function will be made in just one place, rather than the entire code. Applying the behaviour to a component is done by specifying a join-point – a particular point in the execution of a program. When the program hits the join-point, then an advice (beaviour) will be executed.

There are two type of feature that aspect have, the dynamic and static type. Static type means it can change the structure of a component statically, meanwhile dynamic don't. So static crosscutting can add a member, function or add new inheritance hierarchy to a component. The dynamic aspect consists of:

- pointcut, a collection of join points
- Advice, a behaviour that we want to add to the application when it hit a particular pointcut.

AspectJ Getting Started

AspectJ is an implmentation of AOP in Java. It adds some new language constructs like aspect, pointcut and so on (we will discuss more later).

Software that you need to download:

JDK (minimum version 1.4): http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

Download most recent build aspectj compiler at http://www.eclipse.org/aspectj/downloads.php#most_recent. In this module, I am going to use intellij IDEA as the IDE, so I will show you how to setup aspectj base on intellij in the following section, but you can use your favorite IDE. If you are using other IDE you can follow download instruction at http://www.eclipse.org/aspectj/downloads.php#ides.

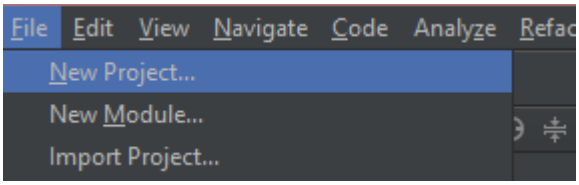
After the download has finished, open the jar file to install aspectj. Follow the installation instruction until finish then you are good to go.

We're going to make a quickstart example. To create this getting started example, follow these instructions below:

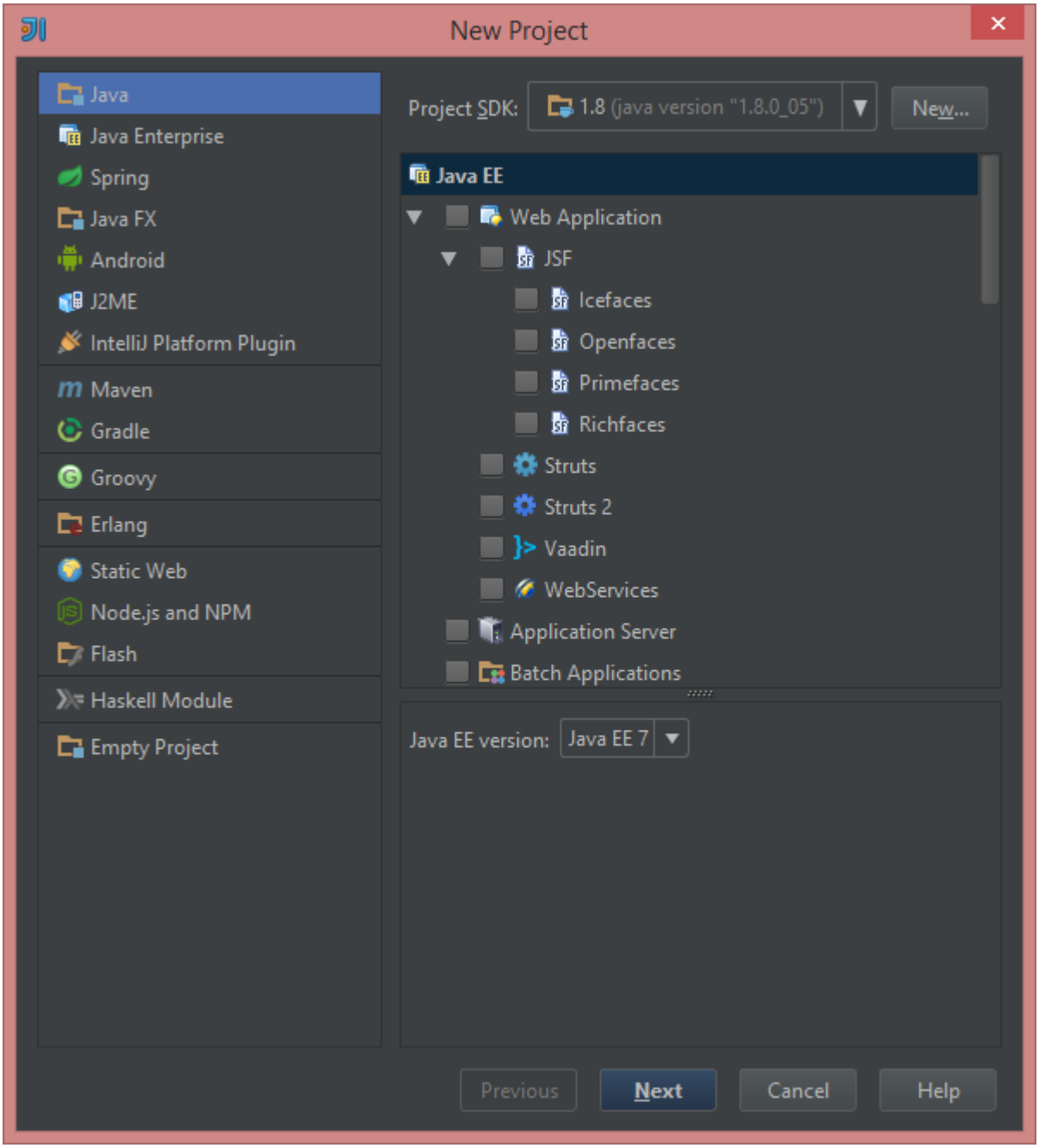
1. Create a new java Project

1.1 Create a new java project

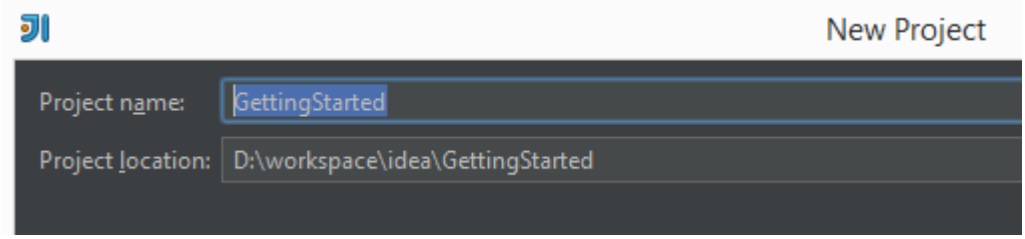
File > New Project



1.2 Choose java then click next two times



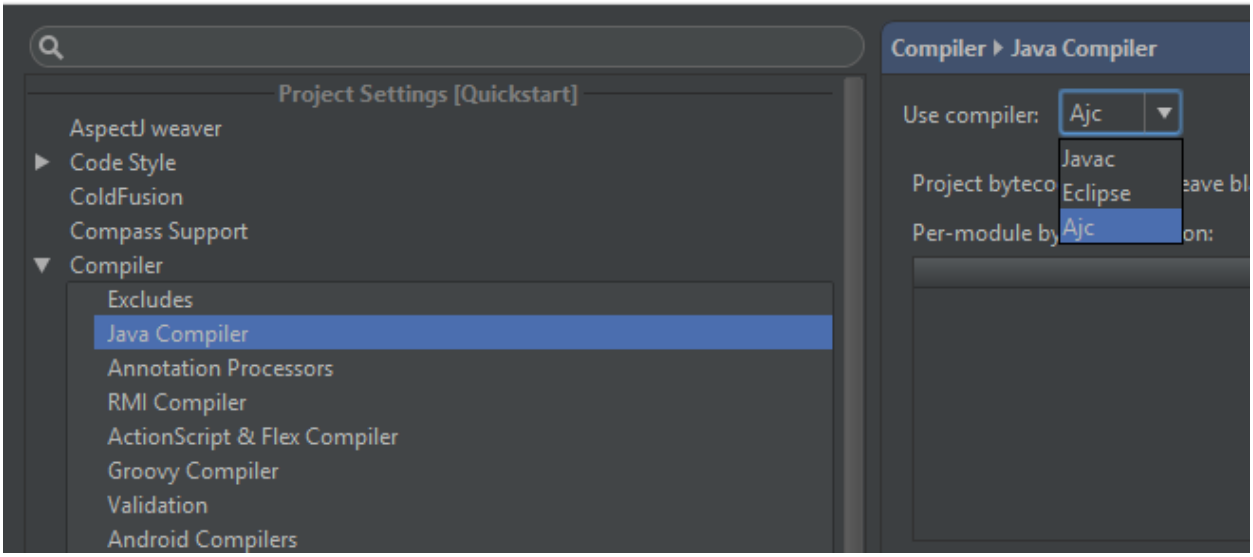
1.3 Named the project **GettingStarted**, then click finish



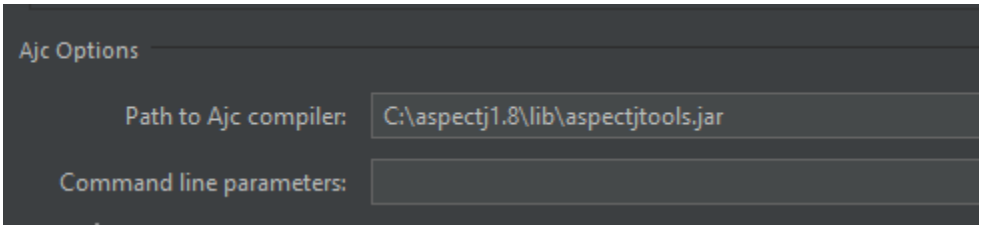
2 Use AJC(AspectJ Compiler) as the project compiler

2.1 Open Settings dialog by pressing **Ctrl + Alt + S**

2.2 Choose Compiler > Java Compiler in the left pane. The change compiler setting to use AJC as the compiler



2.3 In the Ajc option fieldset, specify Path to Ajc compiler to aspectjtools.jar. Aspectjtool.jar should be located at aspectj library folder when you install it.

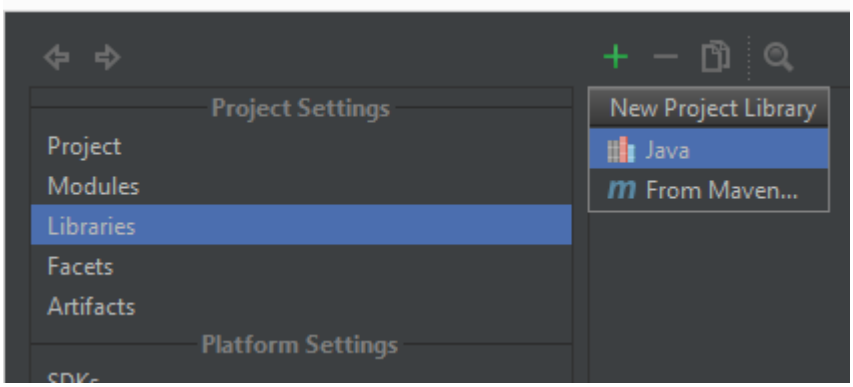


2.4 Finally click apply then ok.

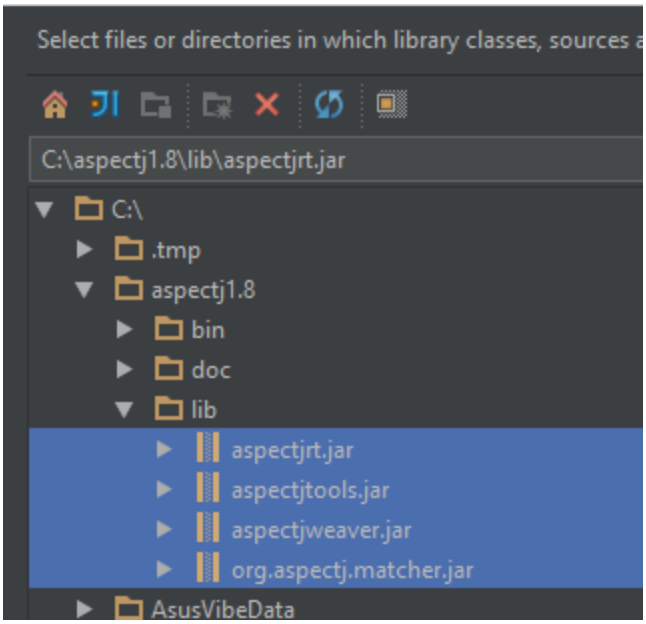
3 Add AspectJ Library to Project

3.1 Open Project Structure by pressing **Ctrl + Shfit + Alt + S**

3.2 Choose library in the left, click the + button then choose java



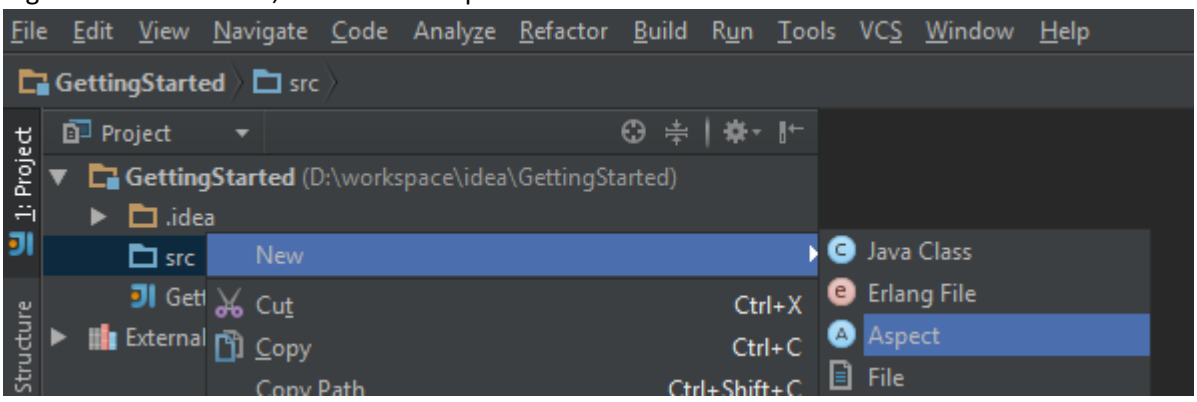
3.3 Choose the all jar files in aspectj library folder. It's located in aspectj library installation folder



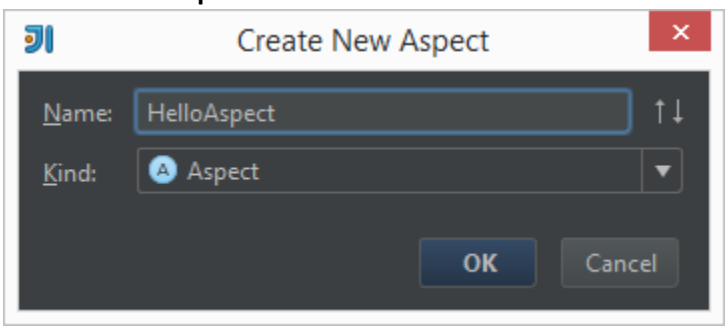
3.4 Click Ok, Choose the current module, then click ok.

4 Create a new aspect file

4.1 Right click at src folder, then choose aspect



4.2 Name it **HelloAspect** then click ok



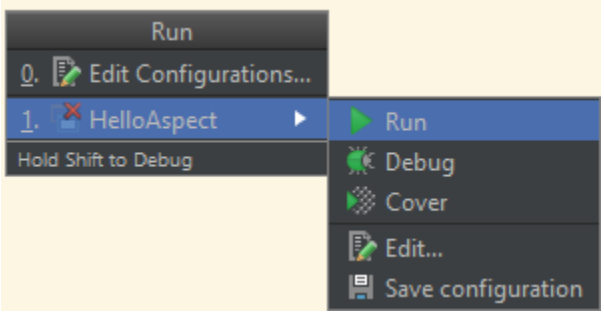
4.3 Finally write the followig code

```
public aspect HelloAspect {

    before() : execution(public static void main(String[])) {
        System.out.println("Before advice is called");
    }

    public static void main(String[] args){
        System.out.println("Hello world");
    }
}
```

4.4 Press Shift + Alt + F10 to run the code



Okay, lets me explain this code. This program will do print a “Hello World” to the console. But before it does, it will be intercepted by another action which print another sentence first.

First, aspect – like a class, it is the modular unit in OOP, it can encapsulate member property, method and also pointcut, advice and inter-type declaration which is new component in AspectJ. Then Before – an advice that will be executed when a particular join point reach. This before advice will be executed before the join point reached. There are other advice besides before, there are after and around. Finally we defined the join point type. In this example, execution of a method is the type of join point, and we tell it’s method signature in the parameter. In summary, this advice says, before a method execution of a public static void main(String[]) method, we want to execute another piece of code first.

Chapter 2

The AspectJ Language

Pointcut

Pointcut is a variable like that stores join point(s) and can be used in advices. Below is the structure to write a pointcut:

```
[access-modifier] pointcut name-of-pointcut(args) : join-points;
```

For example:

```
public pointcut mainExecution() : execution(public static void
main(String[]));
```

We will use a Book class to show various type of join points:

```
public class Book {

    static {
        System.out.println("Static initialization");
    }
    private String ISBN;
    private String name;
    private String publisher;

    public Book() { }

    public String getISBN() { return ISBN; }

    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    public String getName() { return name;}

    public void setName(String name) { this.name = name; }
}
```

1. **call(MethodPattern)** . Picks a method call

```
pointcut setBook() : call(public void Book.setName(..))
            || call(public void Book.setISBN(..));
```

This point cut will pick a setName method call or a setISBN method call. In the above example, we use or logical operator to match one of those join points. Beside **or** logical operator (**||**) we can use **and** logical operator (**&&**) or not operator (**!**) – to achieve a point cut that match both of the join points, or a join point which not match with some particular join point. The **..** in setter method is the wildcard to match any number and any type of arguments.

```
pointcut setBook() : call(void Book.set*(..));
```

We can also can use wildcard to match any word. In the above example, that pointcut will match all setter in Book class with any access modifier.

2. **execution(MethodPattern)**.

Picks a method execution Like a call join point, but the difference is execution join point happens on the reciever's side, that's the actual code that is being executed. Meanwhile, call join point happens, before the code executed, and it happened on the sender's side. We will see the example latter.

3. **set(FieldPattern)** and **get(FieldPattern)**.

Picks out field set and get access.

```
pointcut setPublisherField() : set(String Book.publisher);
```

Whenever a publisher field of type String is set with a new value, then this pointcut is called.

4. **adviceexecution()**. Pick all advice execution join point

5. **preinitialization(ConstructorPattern)** & **initialization(ConstructorPattern)**. Preinitialization happens before an object creation, and initialization happens after an object creation.

```
pointcut newBook() : initialization(Book.new());
```

Match a join point of Book no-argument constructor. You can also write that pointcut with execution and call join point. The execution of constructor has the same meaning as initialization/preinitialization. Meanwhile call is difference, because call happens on the caller side, before the code is executed.

```
pointcut newBook() : execution(Book.new(..));
```

or

```
pointcut newBook() : call(Book.new(..));
```

6. **staticinitialization(CosntructorPattern)**

A static initialization execution, it happens only once.

```
pointcut staticBook() : staticinitialization(Book);
```

This point cut is will be match, whenever a Book object is created, or a Book class is loaded using Class.forName("Book"), for example:

```
try {
    Class.forName("Book");
} catch (ClassNotFoundException e) { }
```

7. **args(..)**.

Publish argument objects, so that we can use the value later in advice.

```
pointcut setBook(String value) : call(void Book.set*(String))
           && args(value);
```

This point capture a setter method in Book which has String as it's arguments, then named it value.

8. **this(Type or Id)** && **target(Type or Id)**.
   The **this** join point specify the type of object that is being executed, But **target** specify the target object that is being executed.

```
pointcut setField(String value) : set(String *.*)
         && target(Book)
         && args(value);
```

In this example, we capture every calls of set field which type is String. Then we specify the target type is Book. Finally we publish is value that is being set.
We can also access the target object that is being executed, for example like this:

```
pointcut setField(Book book, String value) : set(String *.*)
         && target(book)
         && args(value);
```

9. **within(TypePattern)** && **withincode(TypePattern)**

Within pick the join point calls which happened on some Class. The Withincode pick the join point calls which happened on some method

```
pointcut getBookName() : execution(* Book.getName()) && within(Book); //No. 1
pointcut getBookName() : call(* Book.getName()) && within(Book); // No. 2
```

For example, the first pointcut will never match any join point, because Book.getName method is being called outside of it's ownclass, unless Book.getName has a recursive call. But The second pointcut will match the Book.getName method, because the execution of the method is inside the class itself.

10. **cflow(Pointcut)** && **cflowbelow**.

Picks a pointcut which in a control flow of another pointcut. Cflow include the pointcut itself to the call, but cflow below does not. Below is an example to show the difference between cflow and cflowbelow.

```
    public void methodA() {
        methodB();
    }
    public void methodB() { }
```

Assume there is two method, methodA and methodB. methodA will call methodB.

```
pointcut example1() : call(* *.methodB()) && cflow(call(* *.methodA()));
pointcut example2() : call(* *.methodA()) && cflow(call(* *.methodA()));
pointcut example3() : call(* *.methodA()) && cflowbelow(call(* *.methodA()));
//example3 will never match any join point
```

in the first pointcut, it will match any join point of methodB call, but it's call is in methodA's control flow. The second pointcut, it will math any join point of methodA call, and it's in methodA control flow. Because we use cflow, the methodA it's self is counted as part of the control flow. But in the last

pointcut, it's will never match any join point, because we try to match a methodA's call but then we want to the method that is in the control flow below of methodA, which mean not the method itself.

11. **If**.

We can also add some conditional state using if statement in the pointcut. This statement only accept boolean expression which are true and false. Also, we can only use static method or the arguments that is publish in the pointcut. Otherwise it's a compile error.

```
    pointcut setter(String value) : call(void Book.set*(String))
        && args(value) && if(value.toLowerCase().contains("programming"));
```

In this pointcut, it match all setter method in Book class whose object being passed to arguments contain a string "programming".

Advice

Advice is the actual implementation of the cross-cutting behaviour. Advice use pointcut – the join point to know where to implement the behaviour.

Below is the structure of advice:

```
Advice-type(args) : join-points or pointcut(args) { behaviour }
```

There three type of advice:

1. Before advice. Runs before the join point.

```
    pointcut newBook() : initialization(Book.new(..));

    before() : newBook() {
       //do something
    }
```

In the above example, we add a behaviour before the book is initialized.

2. After advice. Runs after the join point

```
    after(Book book) : call(* Book.set*(..)) && target(book){
        //do something
    }
```

In the above example, we add an advice after every call of setter in Book class. Also it publish the current object that is being executed.

There is two type of after, which are after returning or after throwing. After returing happens after a method successfully returns, and after throwing happens if a method fail and throw an exception. If you don't specify the type, then it' will match both.

```
    after() returning(String returnObject) : call(* Book.get*()){
        //do something
    }
```

In this example, we capture the

3. Around advice.

Around advice is a special advice that can capture before and after behaviour. It can also manipulate the result of method call if any, so we have a return type for this advice. Below is the example of around advice that match all getter method in Book class which return type is String, and finally uppercase the result.

```
    String around() : call(String Book.get*()) {
        //before method call
        String result = proceed();
        System.out.println(result);
        //after method call
        return result.toUpperCase();
    }
```

The proceed methos is a special method that will run the join point code. It takes the same argument and return the same object type as the actual code.

Inter-type declaration

Aspectj inter-type can declare members which change the structure of a class, add new hirarchy inheritence to existing classes, add a compile error, so we can debug our code in development phase. Here is the example:

Syntax of writing new member field/method

```
    Access-modifier return-type class.field/method
```

Syntax of adding new hirarchy of existing class

```
    Declare parents : class extends/implments class/interface
```

Syntax of declare an error message

```
    Declare error : joinpoint : error message
```

```
public aspect HelloAspect {

    interface CustomInterface {}

    private String Book.summary = "..."; //add field to Book class
    //add a method to Book class
    private String Book.getSummary(String name) {
        return "...";
    }

    //add a new hirarchy for Book class
    declare parents: Book implements CustomInterface;

    //make a compile error
    declare error: call(* CustomInterface+.set*(..))
                 : "Should not call the set method";

    public static void main(String[] args){
        Book b = new Book();
        b.setName("...");
```

```
    }
}
```

In this example, first we add new member field and method to Book class. The access modifier is important. If we use private, then this new field and method can only be accessed by this aspect. If we specify public as its access modifier then we can have a compile error of name conflict if there's already a field/method with the same name.

In our declare error, we use plus(+) symbol to indicate all calls of setter method in CustomerInterface child class, including book, because we already add our CustomerInterface as an interface for Book. This declare parents only works in HelloAspect.

Aspect

Aspect is a cross-cutting type. It is defined by aspect keyword. We can think aspect as a class but with additional feature and constraint.

Aspect extension

Aspect can extends other class implements interface and also extends other aspect. But class cannot extends aspect. The aspect, the one that being extended must be an abstract aspect. Usually we make abstract aspect to make a variation of pointcut but they all have the same advice. Here is the example.

```
abstract aspect MyAspect {

    abstract pointcut setter();

}

aspect MyAspectImpl extends MyAspect {

    pointcut setter() : call(* *.set*(..));

    before() : setter() {
        //before behaviour
    }

}
```

By default, an aspect is instantiate as an singleton. If you want an aspect is instantiate for every certain object or cflow you can use **perthis(pointcut)** or **percflow(pointcut)** syntax. Here is the example:

```
abstract aspect MyAspect perthis(book) {

    pointcut book() : execution(* Book.*(..));

}
```

Beside perthis, there are also pertarget, percflow and percflowbelow. Each will instantiate an aspect acording to the each join points rule. For example percflow will instantiate an aspect every control flow of a method.

At the end, I will show you one simple example of implementing very simple observer pattern using aspectj. Our example will demonstrate a renting service in online bookstore – the one without aspectJ and with aspectj, so we can compare both codes.

Here is the domain object for online bookstore service

```
//Book.java
import java.util.HashSet;
import java.util.Observable;
import java.util.Observer;
import java.util.Set;

public class Book extends Observable {

    private String name;
    private boolean available;

    private Set<Observer> observers = new HashSet<Observer>();

    public Book(String name) {
        this.name = name;
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(Object arg) {
        for (Observer observer : observers) {
            observer.update(this, arg);
        }
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
        notifyObservers("Change book name to " + this.name);
    }

    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
```

```
        this.available = available;
        notifyObservers("Change book availability to " + available);
    }
}
```

And here is example of Observer source code

```
//RentBookService.java
import java.util.Observable;
import java.util.Observer;

public class RentBookService implements Observer {

    private String name;

    public RentBookService(String name) {
        this.name = name;
    }

    @Override
    public void update(Observable o, Object arg) {
        System.out.println(this.name + ". Book update:" + arg);
    }

}
```

And here is the main program

```
//Application.java
public class Application {

    public static void main(String[] args) {

        Book book1 = new Book("C How to program");
        Book book2 = new Book("Object Oriented Software Engineering");

        RentBookService onlineRentService = new RentBookService("Online
Service");
        RentBookService onSiteRentService = new RentBookService("OnSite
Service");

        book1.addObserver(onlineRentService);
        book1.addObserver(onSiteRentService);
        book2.addObserver(onSiteRentService);


        book1.setName("Java How to Program");
        book2.setName("Functional Programming Software Engineering");

        book2.setAvailable(true);
    }

}
```

```
/*
Output:
Online Service. Book update:Change book name to Java How to Program
OnSite Service. Book update:Change book name to Java How to Program
OnSite Service. Book update:Change book name to Functional Programming
Software Engineering
OnSite Service. Book update:Change book availability to true

*/
```

As you can see, in Book.java the concern between domain object and observerable is tangling around each other. The domain object itself doesn't suppose to have add, remove or notify the observer, as well as the set of obeserver. With Aspectj we can separate these concerns. Below is the source code of Book.java and BookAspect.js.

```
//Book.java
import java.util.HashSet;
import java.util.Observable;
import java.util.Observer;
import java.util.Set;

public class Book extends Observable {

    private String name;
    private boolean available;

    public Book(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }
}
```

```
//BookAspect.aj
import java.util.Observer;
import java.util.Set;
import java.util.HashSet;

/**
```

```
 * Created by Kenrick Satrio on 10/7/2014.
 */
public aspect BookAspect {

    //inter type
    private Set<Observer> Book.observers = new HashSet<Observer>();

    public void Book.addObserver(Observer observer) {
        this.observers.add(observer);
    }

    public void Book.removeObserver(Observer observer) {
        this.observers.remove(observer);
    }

    public void Book.notifyObservers(Object arg) {
        for (Observer observer : this.observers) {
            observer.update(this, arg);
        }
    }

    //pointcut
    pointcut beforeSetName(Book book, String name)
            : call(* Book.setName(String)) && args(name) && target(book);
    pointcut beforeSetAvailability(Book book, boolean available)
            : call(* Book.setAvailable(boolean)) && args(available) &&
target(book);

    //advice
    after(Book book, String name) : beforeSetName(book, name) {
        book.notifyObservers("Change book name to " + name);
    }

    after(Book book, boolean available) : beforeSetAvailability(book,
available) {
        book.notifyObservers("Change book availability to " + available);
    }

}
```

We still have the same output, but now the domain object is clearly seperated from it's observerable aspect.

In summary AspectJ can help you clearly separate cross cutting concern, that will make the code have higher maintainability, readability, modularity and usability. I hope this module can help you understand about AspectJ. Of course there is still alot research in this area, I hope you will learn more deep about AOP and AspectJ.

References:

http://www.eclipse.org/aspectj/
http://en.wikipedia.org/wiki/Aspect-oriented_programming