Aspect Oriented Programming

2018-2019
Course 5

Course 5 Contents

- AspectJ Language:
  - Aspects
  - Examples

Aspects

- Aspects represent the unit of modularization in AOP and AspectJ.
- They provide a way to include crosscutting constructs such as pointcuts and advice.
- An aspect encapsulates the implementation of a crosscutting functionality.
- The AspectJ weaver takes the rules specified in each aspect and uses them to modify the behavior of the core modules in a crosscutting manner.
- An aspect declaration looks similar to the declaration of a class:

```
[access specification] [abstract] aspect <AspectName>
[extends class-or-aspect-name] [implements interface-list]
[[<association-specifier>(Pointcut)] |[pertypewithin(TypePattern)]]
{
   ... aspect body
}
```

Aspects

- The keyword **`aspect`** is used to declare an aspect.
- Each aspect declaration has the following characteristics:
  - It has a name to enable the other parts of the program to refer to it and its elements.
  - It may have an access specification.
  - It may be marked as **`abstract`**.
  - It may extend another aspect or a class, and implement interfaces.
  - It may specify an instantiation model using the optional specifiers **`[<association-specifier>(Pointcut)]`** or **`[pertypewithin(TypePattern)]`.**
  - Its body contains pointcuts, dynamic and static crosscutting constructs, data members and methods, as well as nested types.

Aspects vs Classes

- Aspects are similar to classes in many ways.
- Similarities:
  - They both have data and members.
  - They both have access specifications.
  - They can be marked as abstract.
  - They may implement or extend other types.
  - They can be nested inside other types.
- Differences:
  - Aspects instantiation model is different.
  - Aspects have additional restrictions on inheritance.
  - Aspects may have relaxed access rules.

Aspects vs Classes

Aspects can include data members and methods.

- The data members and methods in aspects have the same role as in classes.
  - An aspect can manage its state using data members.
  - Methods can implement behavior that supports the crosscutting concern's implementation or can be utility methods.
- Aspects may include constructors.
  - If a concrete aspect includes a constructor, it must also include a default constructor to allow the system to instantiate the aspect.
- You should not instantiate an aspect directly using any constructor.
  - Only the default constructor can call constructors that take arguments.

Aspects vs Classes

Aspects can have access specifications

- An aspect's access specifier governs its visibility following the same rules as classes and interfaces.
- Top-level aspects can have only public or package (specified by omitting the access specifier) access.
- Nested aspects, like nested classes, can have a public, private, protected, or package access specifier.

Aspects vs Classes

Aspects can be abstract

- An abstract aspect can mark any pointcut or method as abstract and refer to it from other constructs.
- An aspect that contains abstract pointcuts or methods must declare itself as an abstract aspect.
- Any subaspect of an abstract aspect that does not define every abstract pointcut and method in the base aspect, or that adds additional abstract pointcuts or methods, must be also declared abstract.
- With abstract aspects, you can create reusable units of crosscutting by deferring some of the implementation details to subaspects.
- The mechanism of creating abstract aspects is useful for library aspects.

Aspects vs Classes

```
public abstract aspect AbstractTracing {

   public abstract pointcut traced(); //abstract pointcut

   public abstract Logger getLogger(); //abstract method

   before() : traced() {

    getLogger().log(Level.INFO, "Before: " + thisJoinPoint);

   }

}
```

Aspects vs Classes

- *Aspects can extend classes or abstract aspects, and implement interfaces*

```
public aspect ContestTracing extends AbstractTracing {
    public pointcut traced(): execution(* contest..*.*(..));
    public Logger getLogger() {
        return Logger.getLogger("contest");
    }
}
```

An aspect that extends a class and implement interfaces can access the base-class functionality in the same way as a subclass.

Aspects vs Classes

Aspects can be embedded in classes and interfaces as nested aspects

- Aspects can be embedded into classes and interfaces when the aspect's implementation is closely tied to its enclosing class or interface.
- Because the aspect resides in the same source file, this simplifies the modifications required for the aspect's implementation when the enclosing entity changes.

Remark:

An embedded aspect must be marked as static.

Aspects vs Classes

Aspects cannot be directly instantiated

- The system instantiates aspects appropriately. You never use `new` to create an aspect instance.

Aspects cannot inherit from concrete aspects

- Although aspects can inherit from abstract aspects, they cannot inherit from concrete aspects.
- This limitation exists to reduce complexity.
- The AspectJ compiler considers only concrete aspects for weaving.

Aspects vs Classes

Concrete aspects may not declare generic parameters

- As the system instantiates an aspect, there is no way to bind a generic type parameter during such instantiation.
- Only abstract aspects may declare the generic-type parameters.
- Any concrete aspect extending such an aspect must bind the generic parameters in its declaration.

Aspects can be marked as privileged

- Aspects can have a special privileged access specifier. This gives them access to the private members of the classes they are crosscutting.

Aspect association

- By default, only one instance of an aspect type exists (i.e., like singleton class). The state of such an aspect is effectively global.
- But in some situations (e.g. when creating reusable aspects), you want to associate the aspect's state with an individual object, a specific class, or a control flow.
- The aspect-association mechanism offers different ways to associate aspect instances.
- AspectJ offers four different kinds of aspect association:
  - Singleton (default)
  - Per object
  - Per control-flow
  - Per type

Aspect association

- Although an aspect is a singleton aspect by default, you can explicitly specify so:

```
aspect <AspectName> [issingleton()] {
 ... aspect body
}
```

- For per-object and per–control-flow association, you can specify association by modifying the aspect declaration that takes the following form:

```
aspect <AspectName> [<perthis|pertarget|percflow|
   percflowbelow>(<Pointcut>)] {
... aspect body
}
```

Aspect association

- For per-type association:

```
aspect <AspectName> [pertypewithin(TypePattern)] {
... aspect body
}
```

Aspect precedence

- It is possible to have a system that has multiple advices in the same join point or different aspects that interact with the same join point.
- How does AspectJ determine the order in which advices are applied, and how can you control this order?

```
package ajia;

public class Home {
   public void enter() {
     System.out.println("Entering home");
   }
   public void exit() {
     System.out.println("Exiting home");
   }
}
```

Aspect precedence

- An aspect to start the security system.

```
package ajia;

public aspect HomeSecurityAspect {

  before() : execution(void Home.exit()) {
    System.out.println(“Starting security system");
  }
  after() : execution(void Home.enter()) {
    System.out.println(“Stopping security system");
  }
}
```

Aspect precedence

- An aspect for energy-saving functionality.

```
package ajia;

public aspect SaveEnergyAspect {

   before() : execution(void Home.exit()) {
     System.out.println("Switching off lights");
   }
   after() : execution(void Home.enter()) {
     System.out.println("Switching on lights");
   }
}
```

Aspect precedence

- Test class

```
package ajia.main;
public class Main {
  public static void main(String[] args) {
    Home home = new Home();
    home.exit();
    System.out.println();
    home.enter();
  }
}
```

- Possible output

```
Switching off lights
Starting security system
Exiting

Entering
Stoping security system
Switching on lights
```

Aspect precedence

- The output depends on many factors
  - The version of the AspectJ compiler.
- Aspect precedence is arbitrarily determined, unless you specify the advice precedence.
- The preferred sequence when entering the home is

```
Enter > Switch on lights > Stoping security...
```

and when exiting,

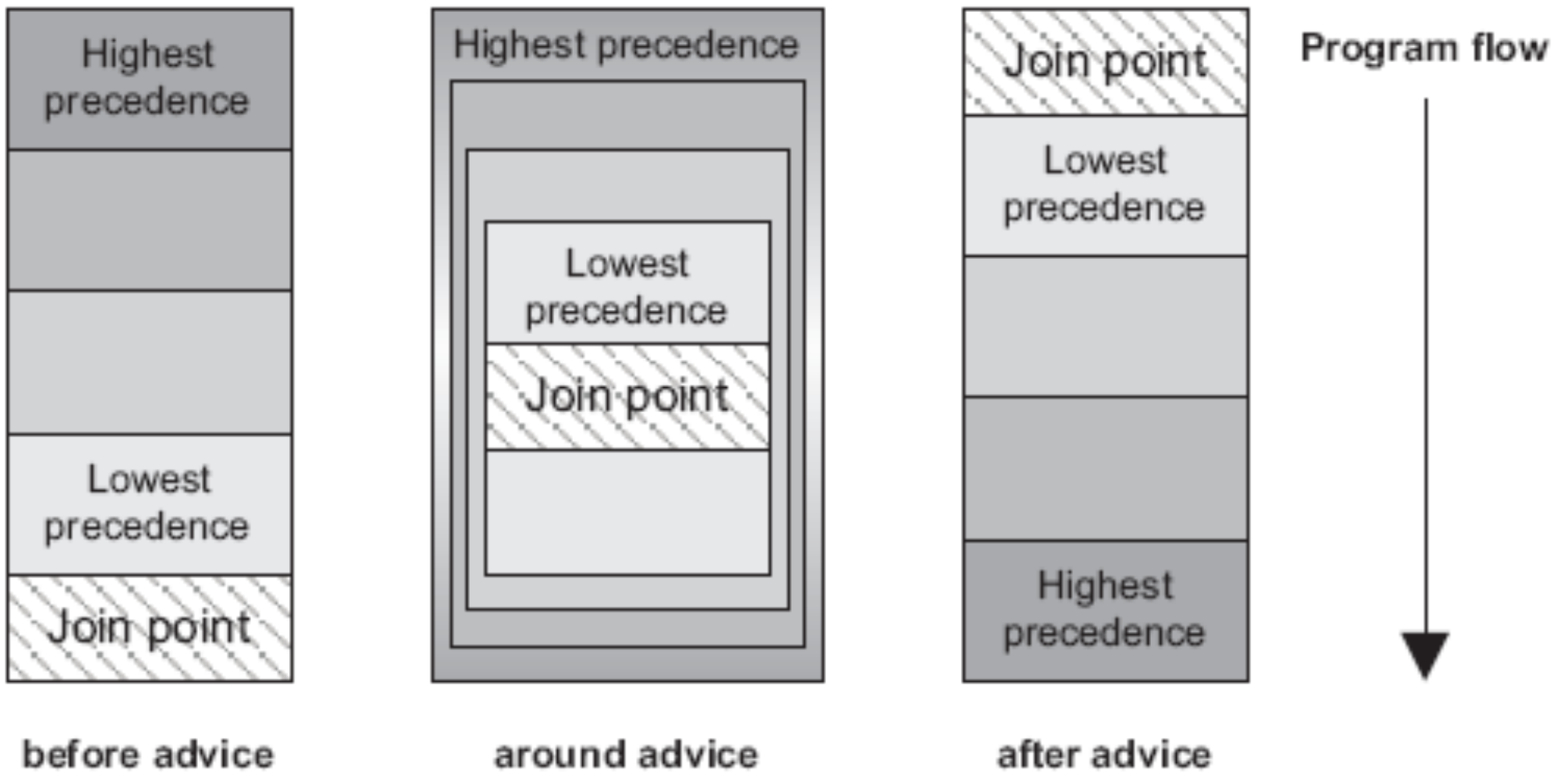
```
Starting security... > Switch off lights > Exit.
```

Aspect precedence

- AspectJ uses the following precedence rules to determine the order in which the advice is applied (for a system with multiple aspects, and pieces of advice in different aspects that apply to the same join point):
  - The aspect with higher precedence executes its **`before`** advice on a join point before the aspect with lower precedence.
  - The aspect with higher precedence executes its **`after`** advice on a join point after the aspect with lower precedence.
  - The **`around`** advice in the higher-precedence aspect encloses the around advice in the lower-precedence aspect.
    - This kind of arrangement allows the higher precedence aspect to control whether the lower-precedence advice will run by controlling the call to **`proceed()`**.
    - If the higher-precedence aspect does not call **`proceed()`** in its advice body, the lower-precedence aspects will not execute, and the advised join point also will not be executed.

Aspect precedence

- Advice ordering



Explicit Aspect Precedence

- AspectJ provides a construct for controlling aspect precedence: **`declare precedence`**.
- The **`declare precedence`** construct must be specified inside an aspect.
- The construct takes the following form:

```
declare precedence : TypePattern1, TypePattern2, ...;
```

```
package ajia;
public aspect HomeSystemCoordinationAspect {
   declare precedence: HomeSecurityAspect, SaveEnergyAspect;
}
//Output
Starting security system
Switching off lights
Exiting
Entering
Switching on lights
Stopping security system
```

Explicit Aspect Precedence

- Remarks:
  - In the absence of any special precedence control, the order in which the advice is applied is unpredictable.
  - The AspectJ compiler can help you spot situations where the advice order is unpredictable—you pass it the **`-Xlint:warning`** flag.
  - Wildcards can be used in the TypePattern list for the aspect name.

```
declare precedence : Auth*, PoolingAspect;
```

There is no precedence define between two aspects whose name start with **`Auth`** (**`AuthenticationAspect`** and **`AuthorizationAspect`**). If controlling the precedence between two such aspects is important, you must specify both aspects in the desired order.

- You can specify a sequence of precedence (as declare precedence takes a type list).

```
declare precedence : Auth*, PoolingAspect, LoggingAspect;
```

Explicit Aspect Precedence

- Remarks:
  - It is common for certain aspects to have higher precedence among all aspects. You can use the *, +, and .. wildcards to indicate such an intention.

```
eg. declare precedence : AuthenticationAspect, *;
```

`AuthenticationAspect` dominates all the remaining aspects in the system.

  - It is common for certain aspects to have lower precedence among all aspects. You can also use a wildcard to achieve this.

```
declare precedence : *, CachingAspect;
```

  - It is an error if a single declare precedence clause causes circular dependency in the ordering of aspect precedence.

```
declare precedence : Auth*, PoolingAspect, AuthenticationAspect;
```

Explicit Aspect Precedence

- Remarks:
  - It is legal to specify a circular dependency with multiple precedence clauses.
  - It is used to enforce that two different, potentially conflicting or redundant aspects share no join points.
  - A compile-time error is generated if the two aspects in question share a join point.

```
declare precedence : AuthenticationAspect, PoolingAspect;
declare precedence : PoolingAspect, AuthenticationAspect;
```

  - You can include a declare precedence clause inside any aspect.
  - A common usage idiom is to add such clauses to a separate coordination aspect so that aspects are unaware of each other.

Explicit Aspect Precedence

- Aspect inheritance and precedence
  - AspectJ implicitly determines the precedence between two aspects related by a base derived aspect relationship.

    Rule: the derived aspect implicitly has higher precedence than the base aspect.
  - Because only concrete aspects in the declare precedence clause are designated for precedence ordering, declaring a base aspect (which is always abstract) to have higher precedence over subaspects has no effect.
  - There is no way to declare higher precedence for base aspects.

Ordering Advice in a Single Aspect

- It is also possible to have multiple advice in one aspect that you want to apply to a pointcut.
- Because the advice reside in the same aspect, aspect-precedence rules can no longer apply.

Rule: the advice that appears first lexically inside the aspect executes first.

- The only way to control precedence between multiple advice in an aspect is to arrange them lexically.

Ordering Advice in a Single Aspect

```
package ajia.main;

public aspect InterAdvicePrecedenceAspect {
   public pointcut performCall() : call(* Main.perform());
   after() returning : performCall() { System.out.println("<after1/>"); }
   before() : performCall() {
      System.out.println("<before1/>"); }
   void around() : performCall() {
     System.out.println("<around>");
      proceed();
      System.out.println("</around>");
   }
   before() : performCall() {
      System.out.println("<before2/>");
   }
}
```

Ordering Advice in a Single Aspect

```
package ajia.main;
public class Main {
  public static void main(String[] args) {
    Main main = new Main();
    main.perform();
  }
  public void perform() {
    System.out.println("<performing/>");
  }
}
//Output
<before1/>
<around>
<before2/>
<performing/>
<after1/>
</around>
```

Privileged Aspects

- For the most part, aspects have the same standard Java access-control rules as classes.
  - An aspect normally cannot access any private members of other classes.
- In a few situations, an aspect may need to access certain data members or operations that are not exposed to outsiders.
- You can gain such access by marking the aspect `privileged`.

Privileged Aspects

```
package ajia.main;
public class Main {
  private int id;
  public static void main(String[] args) {
    Main main = new Main();
    main.method1();
  }
  public void method1() {  System.out.println("Main.method1");   }
}
```

```
package ajia.main;
public aspect PrivilegeTestAspect {
 before(Main callee) : call(void Main.method1()) && target(callee) {
    System.out.println("PrivilegeTestAspect:before objectId="+ callee.id);
 }
}
```

Privileged Aspects

```
Compilation:

ajc Main.java PrivilegeTestAspect.aj

//Compile time error

...PrivilegeTestAspect.aj:9

 [error] The field Main.id is not visible+ callee.id);
                                          ^^^^^^^^^^

1 error

//version 2

privileged public aspect PrivilegeTestAspect {

...

}

//no compilation error
```

Privileged Aspects

Remarks:

- With the privileged aspect, you can access the internal state of a class without changing the class.
- You should be careful when using this feature as privileged aspects have access to implementation details.
- If the classes change their implementation, the aspect accessing such implementation details will need to be changed as well.

Examples

- Observer library
- Performance Monitoring
- Pre and Post Conditions

Performance Monitoring

- Performance monitoring involves measuring the time taken by interesting parts of the system as well as the number of times a particular method is invoked.
- Depending on how it is used, you can monitor activities at various levels.

Eg. you can monitor the service layer to gain more fine-grained information.

- For development purposes, you can focus on parts suspected of slowing down the system.

For example, you can monitor JDBC or ORM calls to monitor database access.

Performance Monitoring

- Performance monitoring of an application involves recording vital parameters of parts of the system: how long it takes to execute certain functionality, how many calls have been made for a particular method, etc.
- You can extend the basic idea of tracing to implement performance monitoring.
- The idea is to compute the difference in timestamps before and after each monitored operation and record that difference against the currently advised join point.
- You can also count the number of invocations for each advised operation.

Pre- and Post-Conditions

- Many programmers use the "*Design by Contract*" style popularized by Bertrand Meyer in Object-Oriented Software Construction, 2/e.
- In this style of programming, explicit pre-conditions test that callers of a method call it properly and explicit post-conditions test that methods properly do the work they are supposed to.
- AspectJ makes it possible to implement pre- and post-condition testing in modular form.
- Even though pre- and post-condition testing aspects can often be used only during testing, in some cases developers may wish to include them in the production build as well.
- Because AspectJ makes it possible to modularize these crosscutting concerns cleanly, it gives developers good control over this decision.