Aspect Oriented Programming

2014-2015

Course 7

Course 7 Contents

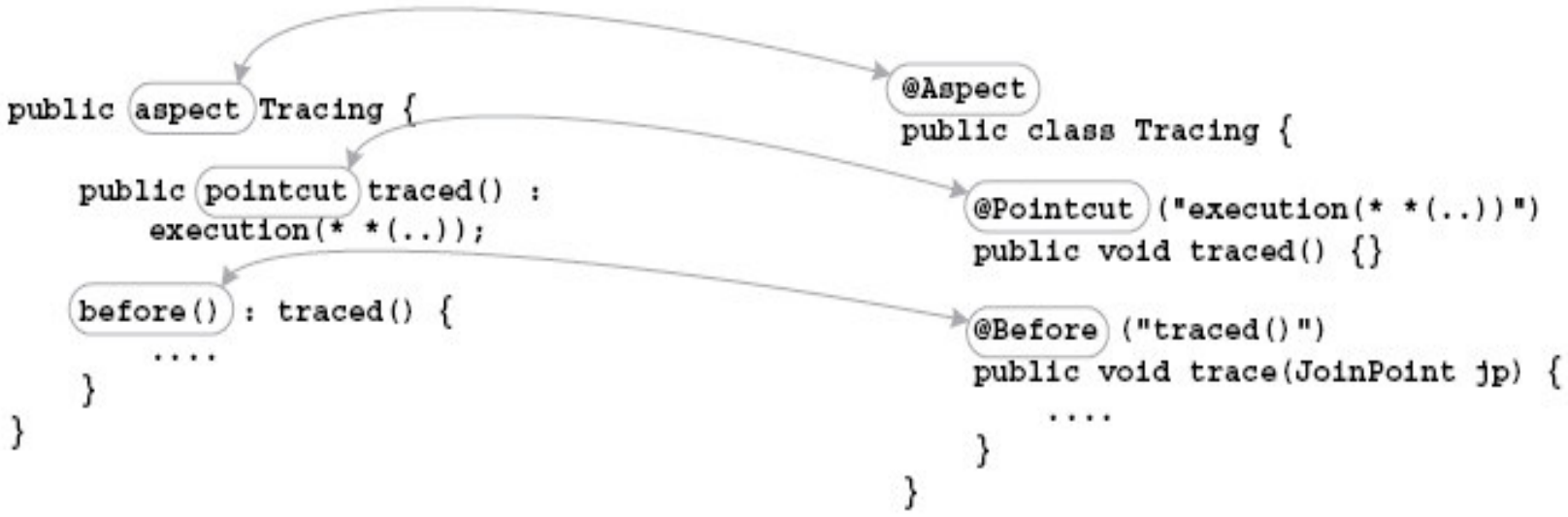
- AspectJ Language:
  - @AspectJ syntax

@AspectJ

- The @AspectJ syntax offers the option of compiling source code with a plain Java compiler and makes it easier to work with any Java IDE.
- The @AspectJ syntax was added due to the merging of AspectWerkz with AspectJ.
- It uses plain Java along with annotations to express crosscutting elements.
- The syntax tends to be verbose and slightly less powerful, but aspects written in @AspectJ syntax can be compiled using a plain Java compiler.
- The Spring Framework uses the @AspectJ syntax within its proxy-based AOP framework without needing an AspectJ weaver.
- You have to add aspectjrt.jar to the project classpath in order to compile successfully.

@AspectJ syntax

- The most important reason for creating the @AspectJ syntax is to keep the code plain Java.
- Crosscutting information are kept in annotations on Java elements. It allows Java compilers and tools expecting plain Java code to work with aspects.
- Each crosscutting element is mapped to a Java element carrying a specific annotation.



@AspectJ syntax

- Any information that cannot be expressed in Java alone is expressed as parameters to the annotations.
- The idea behind the design of the @AspectJ syntax is to find a suitable Java element and annotate it to express the crosscutting characteristic.
- The principles behind the @AspectJ syntax are the natural mapping of crosscutting elements, compatibility with Java, and early error detection.
- Most elements find a direct mapping from the traditional syntax into the @AspectJ syntax (aspect->class, pointcut->method).
- A few elements (i.e., inter-type declarations) need some adjustment.
- A few features (i.e., the privileged aspect) cannot be mapped at all within the constraint of compiling using javac.

@AspectJ Crosscutting Elements

Feature	Mapped element
Aspects	Class with @Aspect annotation
Pointcuts	Method with @Pointcut annotation
Advice	Method with @Before, @After, @AfterReturning, @AfterThrowing, or @Around annotation
Declaring parents	Field with @DeclareMixin and @DeclareParents annotation
Declaring errors and warnings	Field with @DeclareError and @DeclareWarning annotation
Introducing data and methods	Not supported
Exception handling	Not supported
Privileged aspects	Not supported

Java compatibility

- Aspects expressed using the @AspectJ syntax can be compiled using regular javac.
- Any existing IDEs work better with such code, because they can always treat aspects as regular Java code.
- Although you can compile the code for @AspectJ aspects using javac, the Java compiler cannot understand the semantics associated with the @AspectJ annotations.
- The AspectJ weaver must still be used to perform binary weaving during build-time or load-time.
- If ajc is used to compile the aspects written using @AspectJ syntax, it performs weaving at build-time, obviating the need for a special weaving step.

Early error detection

- The @AspectJ syntax is designed to give as many errors and warnings as soon as possible even when you compile using the javac compiler.
- This design dictates using the Java constructs as much as possible instead of using only expressions in strings.
- There are, however, a few deviations from a straightforward mapping from the traditional syntax (eg. the if() pointcut).

```
@Aspect("public aspect SQLPeformanceMonitoring {"
+ " public pointcut monitored() : call(* java.sql.*.*(..)); "
+ " Object around() : monitored() {"
+ " ... "
+ " }"
+ "}")
public class SomeClass {
}
```

Mapping Aspects

- In AspectJ, an aspect is the unit of modularization for crosscutting concerns and consists of elements such as pointcuts and advice.
- In @AspectJ, a class marked with the @Aspect annotation represents an aspect. Such a class can contain other crosscutting elements such as pointcuts and advice, also expressed using the @AspectJ syntax.

```
@Aspect(["<perthis|pertarget|percflow|percflowbelow>(Pointcut)
   |[pertypewithin(TypePattern)]"])
[access specification] [abstract] [static] class <AspectName>
   [extends class-or-aspect-name] [implements interface-list] {
... aspect body
}
```

- One difference from a regular class is the annotation attached to the class that indicates to an AspectJ weaver that the class is to be treated as an aspect.

Mapping Aspects

@ AspectJ syntax	AspectJ syntax

`@Aspect`
`public abstract class Monitoring`
`{`
`...`
`}`
	`public abstract aspect Monitoring`
	`{`
	`...`
	`}`

`@Aspect`
`public class BankingMonitoring`
`  extends Monitoring {`
`...`
`}`
	`public aspect BankingMonitoring`
	`  extends Monitoring {`
	`...`
	`}`

Remark: Only classes can be marked with the @Aspect annotation (not other types such as interfaces and enums).

Mapping Aspects

- There are a few restrictions on the class declaration to match the aspect semantics:
  - Include a default public constructor if the class defines a constructor.
  - Not extend a concrete class carrying the @Aspect annotation. It matches the restriction that an aspect cannot extend another concrete aspect.
  - Not declare a generic parameter unless it is an abstract aspect. It matches the restriction that only abstract aspects are allowed to declare generic type parameters.
  - Be marked static if it is a nested class. It matches the restriction that an inner aspect must be static (not bound to an instance of the enclosing type).
- The @Aspect annotation has an optional property that you can use to specify aspect association.

Accessing the aspect instance

- With traditional-style aspects, you can access an aspect instance using the automatically added **`aspectOf()`** and **`hasAspect()`** method:

  `ProfileAspect.aspectOf()`
- @AspectJ syntax provides an alternative using the **`org.aspectj.lang.Aspects`** class, which contains **`aspectOf`**() and **`hasAspect`**() methods.
- There are two differences.
- With @AspectJ you do the following:
  - Invoke the method on the **`Aspects`** class instead of on the aspect.
  - Pass the aspect's class as the first argument to each method.

Eg. If you need an instance of the **`ProfileAspect`** aspect, you use

`Aspects.aspectOf(ProfileAspect.class).`

Declaring aspect precedence

- The traditional syntax offers the **`declare precedence`** construct to control precedence between aspects.
- @AspectJ syntax offers the **`@DeclarePrecedence`** annotation :

```
@Aspect
@DeclarePrecedence("ajia.HomeSecurityAspect,
   ajia.SaveEnergyAspect")
public class HomeSystemCoordinationAspect {
}
```

- The @DeclarePrecedence annotation must be attached only to an aspect (a class with the @Aspect annotation).
- You set the annotation's value attribute to aspect type patterns separated by a comma, following the same rules as the traditional syntax.

Declaring aspect precedence

Remarks:

- You must use fully qualified aspect types unless the aspect resides in the same package as the aspect declaring precedence.
- You can declare only one precedence per aspect.
- In the traditional syntax, an aspect may include any number of declare precedence statements.
- Equivalent functionality in the @AspectJ syntax requires using multiple aspects, with each declaring a single precedence.
- In a future version of AspectJ, the attribute type may change to `String[]`.

Mapping pointcuts

- The @AspectJ syntax uses a method with a @Pointcut annotation to represent a pointcut.
- The value parameter of the annotation represents the pointcut expression.
- All other pointcut characteristics such as the name, access specification, abstractness, and parameters match with the corresponding characteristics of the method representing it.
- The pointcut expression used is the same as in the traditional syntax except for two differences:
  - the use of fully qualified type names
  - a special treatment of the if() pointcut.

Mapping abstract pointcuts

- Abstract pointcuts allow you to write reusable aspects by letting subaspects provide a definition for those pointcuts.
- The @AspectJ syntax maps abstract pointcuts to an abstract method with the @Pointcut annotation without any annotation parameters (using the default value, instead).

```
@Pointcut
[access specifier] abstract void pointcut-name([args]);
```

- Because an abstract pointcut does not need a pointcut expression, the default value for the pointcut annotation suffices.
- The method representing the abstract pointcut must be declared to return void.
- Method parameters define the pointcut parameters that a concrete pointcut must collect as the join point context.

Mapping Abstract Pointcuts

@ AspectJ syntax	AspectJ syntax
`@Pointcut` `public abstract void readOperation();`	`public abstract pointcut readOperation();`
`@Pointcut` `public abstract void accountOperation(Account account, float amount);`	`public abstract pointcut accountOperation(Account account, float amount);`

Remark: The method-access specification can be public, package (default), or protected, but not private.

Mapping concrete pointcuts

- A concrete pointcut uses a pointcut expression to specify a join point selection criterion and collect join point context.
- The @AspectJ syntax maps concrete pointcuts to a concrete method with the @Pointcut annotation, which specifies the pointcut expression.
- The method body for a concrete pointcut is empty, because the method is a placeholder without any significance for the code inside it (except if())

```
@Pointcut("<pointcut-definition>")
[access specifier] void <pointcut-name>([args]) {}
```

Eg. //@AspectJ

```
@Pointcut("execution(public void set*(*))")
public void setter() {}
```

//AspectJ

```
public pointcut setter() : execution(public void set*(*));
```

Requirements For Pointcut Expressions

- The pointcut expression for a concrete pointcut is the same as in the traditional syntax except for two differences:
  - Type names, if any, must be fully qualified, unless the type resides in the same package as the aspect or belongs to the java.lang package. You cannot use imported type names.
  - If you use an if() pointcut, it must be of the if(true), if(false), or if() form. If you use the last form, the method body must specify the selection criteria for the pointcut.
- The use of fully qualified type names is required because import statements are not retained in compiled byte code.
- It makes it impossible for the AspectJ weaver to deduce the type from the pointcut expression string.

Requirements For Pointcut Expressions

//@AspectJ

```
@Pointcut("call(* java.sql.Connection.*(..))")
public void connectionOperation() {}
```

//AspectJ

```
public pointcut connectionOperation()
: call(* java.sql.Connection.*(..))
```

//or AspectJ

```
import java.sql.*;
...
public pointcut connectionOperation ()
: call(* Connection.*(..))
```

Weaving and error detection

- Although the @AspectJ syntax promotes early error detection, because pointcut expressions are strings, errors in them are not reported until those strings are parsed.
- If ajc is used to compile the aspects, it parses pointcuts immediately and issues any errors.
- If AJDT is used, it also reports any errors immediately.
- If the binary or load-time weaver are used, parsing and error detection only occur at that time.

Requirements For Compilation

- Pointcut parameters expressed as part of the method parameters must be bound using the same pointcut expression.

```
@Pointcut("execution(public * aija.banking.domain.Account.*(float))"
+ " && this(account) && args(amount)")
public void accountOperation(Account account, float amount) {}
```

- If you use javac to compile code that uses method parameters, you must use either the `-g:vars` (or `-g`, which is a superset of `-g:vars`) flag or the `argNames` parameter to the @Pointcut annotation.
- It is required because the Java compiler does not preserve argument names in the compiled code.
- The AspectJ weaver cannot determine the correspondence between the method parameter names (which are lost in the byte code) and identifiers used in the pointcut expression (which, being part of the annotation value, are preserved).

Requirements For Compilation

- Using `-g:vars` instructs the compiler to preserve argument names in the compiled byte code. The use of this flag leads to slightly larger class files; but no degradation occurs in the runtime performance of such a class.
- If you do not use the `-g:vars` flag, you need to use the `argNames` parameter to preserve the same information.
- The value of the `argNames` parameter is a comma-separated list of the method parameter names in the same order as defined by the method.
- Because `argNames` duplicates parameter names, there is a chance that it may become inconsistent (e.g., method parameters are rearranged without rearranging the `argNames` parameters value).

```
@Pointcut(value="execution(public *
   ajia.banking.domain.Account.*(float)) && this(account) &&
   args(amount)", argNames="account, amount")
public void accountOperation(Account account, float amount) {}
```

The if() Pointcut

- In the traditional syntax, you include the conditional statement in the if() pointcut definition.
- With @AspectJ, you must provide the same statement in a method.
- It allows more checks to be performed at compile-time even when you use a Java compiler.

//AspectJ syntax

```
pointcut debugEnabled() : if(logLevel >= DEBUG);
```

//@AspectJ syntax

```
@Pointcut("if()")
public static boolean debugEnabled() {
return logLevel >= DEBUG;
}
```

The if() Pointcut

- The pointcut expression in the @AspectJ syntax takes "if()" as part of the annotation value.
- The method declares that it returns a boolean and the body is the expression used in the traditional syntax.
- The method used for the if() pointcut must be public and static.
- You can then use this named pointcut as a part of other pointcuts

```
@Pointcut("traced() && debugEnabled()")
public void debugTraced() { }
```

- You can also rewrite the same pointcut to avoid the explicit **`debugEnabled()`** pointcut

```
@Pointcut("traced() && if()")
public static boolean debugTraced() {
     return logLevel >= DEBUG;
}
```

Mapping dynamic crosscutting constructs

- An advice is represented with a method.
- The method representing an advice carries a @Before, @After, @AfterReturning, @AfterThrowing, or @Around annotation to denote the kind of advice the method is implementing.
- The value attribute of the annotation denotes the associated pointcut (named or anonymous), whereas the method body denotes the advice to be executed.
- Because a method has a name, that name is also the advice's name.
- In the traditional syntax, advice does not have an inherent name (it may be annotated with @AdviceName to assign it a name).

Mapping dynamic crosscutting constructs

- The methods that stand in for advice:
  - Should not return a value (they must return `void`), except for the around advice. It matches the traditional syntax, where advice may not return a value unless it is an around advice.
  - May declare that it throws an exception similar to the traditional syntax.
  - Should not be declared static. It matches the traditional syntax, where advice behaves like an instance method, because it has access to the `this` variable that points to the aspect instance.
- Although all advice constructs follow common ideas, each construct has a few particularities.

The before advice

- The before advice is created using a method with the @Before annotation.

```
@Before("<pointcut>")
public void <advice-name>([arguments]) {
   ... advice body
}
```

- The value of the @Before annotation specifies the pointcut associated with the advice.
- The pointcut may be named (referring to a pointcut defined in the same or a different aspect) or may be a pointcut expression.
- The method used as advice must be public and must return void.
- The advice name does not matter from AspectJ's point of view, however it should be meaningful to represent the logic carried by the advice.
- The optional arguments to the method represent the join point context.

Advising With Anonymous Pointcuts

```
package ajia.monitoring;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class SystemHealthMonitor {
  HeartBeatListener heartBeatListener= new HeartBeatListener();

  @Before("execution(* *(..)) && !within(ajia.monitoring.*)")
  public void beatHeart() {
     heartBeatListener.beat();
  }
}
```

Remark: The pointcut `execution(* *(..)) && !within(ajia.monitoring.*)` will not select the `beatHeart()` method that stands in for an advice.

Although `beatHeart()` is a method, from AspectJ's perspective, it really is an advice. The join point corresponds to its advice-execution join point and not a method execution.

Advising With Named Pointcuts

```
package ajia.monitoring;
import org.aspectj.lang.annotation.*;

@Aspect
public class SystemHealthMonitor {
  HeartBeatListener heartBeatListener= new HeartBeatListener();

 @Pointcut("execution(* *.*(..)) && !within(ajia.monitoring.*)")
  public void aliveOperation() {}

  @Before("aliveOperation()")
  public void beatHeart() {
    heartBeatListener.beat();
   }
}
```

Using A Reflective Join Point Context

- In @AspectJ styled aspects, you can not use the special variables `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart` .
- AspectJ requires the advice to declare method parameters of the corresponding type and use those parameters inside advice.
- The weaver ensures that those parameters are appropriately passed when advice is executed.

Using A Reflective Join Point Context

```
package ajia.tracing;
import ...

@Aspect
public class Tracing {

  @Pointcut("execution(* *.*(..))")
  public void traced() {}

  @Before("traced()")
  public void trace(JoinPoint jp) {
    System.out.println("Entering " + jp);
  }
}
```

Using A Reflective Join Point Context

- In the advice, you get reflective access to the dynamic context of the current join point by declaring a method parameter of type `JoinPoint`. When the advice is executed,the information in this variable is exactly the same as you get in `thisJoinPoint` in the traditional syntax.

```
//AspectJ syntax
before() : traced() {
   System.out.println("Entering " + thisJoinPoint);
}
```

- You can similarly declare a parameter of type `JoinPoint.StaticPart` to obtain static information associated with the current join point, logically the same as the `thisJoinPointStaticPart`.
- Because the type of both `thisJoinPointStaticPart` and `thisEnclosingJoinPointStaticPart` is `JoinPoint.StaticPart`, the weaver cannot determine the objects you intend if it sees two variables of the same type.
- If the advice needs to obtain the enclosing join point context, it must declare a variable of type `JoinPoint.EnclosingStaticPart` (a subtype of `JoinPoint.StaticPart`).

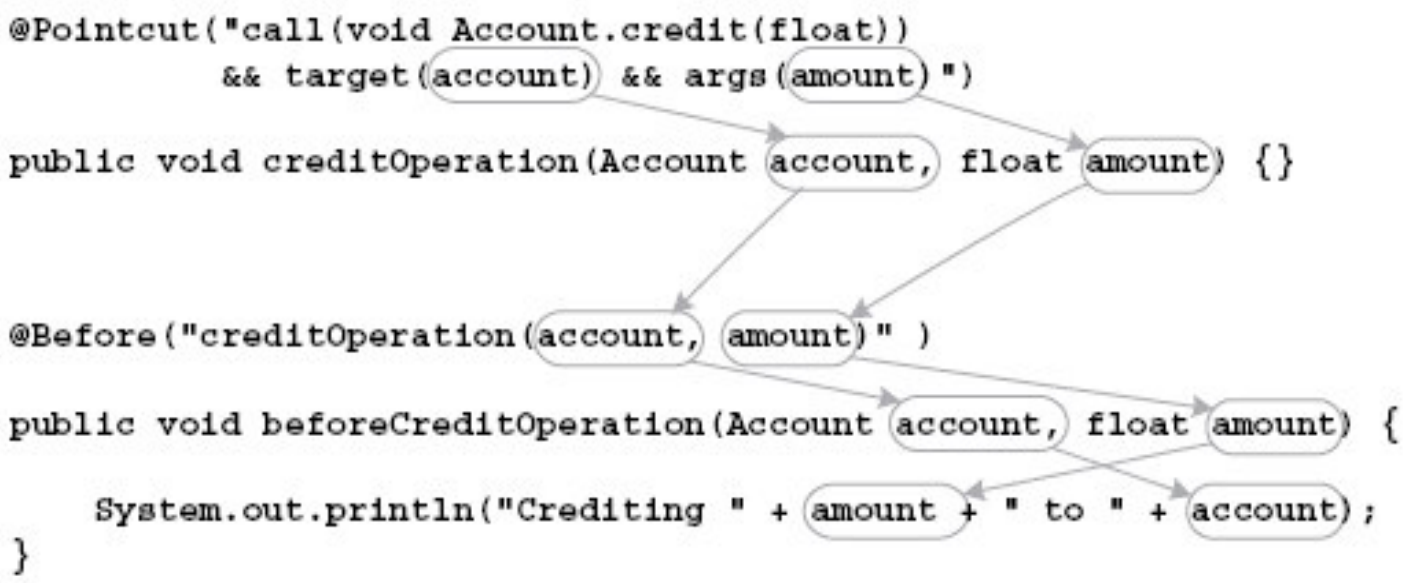
Using A Typed Join Point Context

- The @AspectJ syntax requires that the needed join point context be declared as method parameters the same way a traditional advice declares advice parameters. The pointcut must then use the same parameter names to collect context in pointcut.
- For anonymous pointcuts: the method parameter declares the needed context, and the context-collecting pointcut binds those parameters to the join point context.

```
@Before("call(void Account.credit(float)) && target( account ) && args( amount )")

public void beforeAccountOperations(Account account, float amount ) {

     System.out.println("Crediting " + amount  + " to " + account );

}
```

Using A Typed Join Point Context

- For context collection using named pointcuts: the pointcut method parameter declares the needed context, and the context-collecting pointcut binds those parameters to the join point context.
- The advice then maps that context:



Using A Typed Join Point Context

- You can mix reflective access to join point information with that obtained using context-binding pointcuts by including one or more variables of appropriate reflective access types.

```
package ajia.monitoring;
import ...
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) && target(connection)")
  public void connectionOperation(Connection connection) {}

  @Before("connectionOperation(connection)")
  public void monitorUse(JoinPoint.StaticPart jpsp,
   JoinPoint.EnclosingStaticPart jpesp, Connection connection) {
    System.out.println("About to use " + connection + " to perform " +
   jpsp.toShortString() + " called from " + jpesp.toShortString());
   }
}
```

The after advice

- In AspectJ, the after advice has three variations: after (finally), after returning, and after throwing.
- The after (finally) advice, which executes regardless of how the join point execution completes, is similar to before advice from the code point of view. The only difference is the annotation used: @After.

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) &&
   target(connection)")
  public void connectionOperation(Connection connection) {}

  @After("connectionOperation(connection)")
  public void monitorUse(Connection connection) {
    System.out.println("Just used " + connection);
  }
}
```

The after advice

- The other two variations of after advice (after returning and after throwing) come in two sub-variations.
- The first variation uses the fact that the join point executes normally or by throwing an exception.
- The second variation collects either the returned object (for the after-returning advice) or the thrown exception (for the after throwing advice).
- The first variation requires a trivial change: instead of using @After, you use @AfterReturning or @AfterThrowing,

The after advice

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) && target(connection)")
  public void connectionOperation(Connection connection) {}

  @AfterReturning("connectionOperation(connection)")
   public void monitorSuccessfulUse(Connection connection) {
     System.out.println("Just used " + connection + " successfully");
   }

   @AfterThrowing("connectionOperation(connection)")
    public void monitorFailedUse(Connection connection) {
       System.out.println("Just used " + connection+ " but met with a
   failure");
   }
}
```

Collecting The Return Value And Thrown Exception

- The second variation of the after advice, the one that needs access to the returned object or the thrown exception, must be expressed differently.
- For the after-returning advice: you can specify an additional parameter to the method and use the parameter name as the value of the returning attribute of the annotation to that parameter name.
- The after-throwing advice: you specify an exception type as the method parameter and use the name of the parameter as the value of the throwing attribute.

Collecting The Return Value And Thrown Exception

```
package ajia.monitoring;
import ....
@Aspect
public class ConnectionMonitor {
  @Pointcut("call(* java.sql.Connection.*(..)) && target(connection)")
  public void connectionOperation(Connection connection) {}

  @AfterReturning(value="connectionOperation(connection)",
   returning="ret")
  public void monitorSuccessfulUse(Connection connection, Object ret) {
    //...
  }

  @AfterThrowing(value="connectionOperation(connection)",throwing="ex")
  public void monitorFailedUse(Connection connection, Exception ex) {
    //...
 }
}
```

Collecting The Return Value And Thrown Exception

- You must explicitly use the value attribute, because the annotation specification in Java lets you omit the attribute name only if a single value is specified and the attribute name is value.
- The pointcut attribute is also available, which you can use in place of value.
- If both pointcut and value attributes are specified, the pointcut attribute takes precedence.
- The type of the return value or the exception specified limits the advice applicability to where the return object or the thrown exception is assignable to the specified type.

Example:

Instead of specifying Object as the return value type, if you specify Statement, only methods that return Statement or its subtype will be selected. This behavior is identical to that seen with the traditional syntax.

The around advice

- An around advice in @AspectJ is represented by a method with an @Around annotation.
- The method must be public and may return a value.
- The rules governing the return value are the same as the rules in the traditional syntax.
- The return value must be compatible with all matching join points. The method may also return Object, where the matching join points may return any type and the weaver takes care of necessary unboxing and casting.

Proceeding With The Original Join Point Execution

- In the traditional syntax, if you want to execute the original join point you use the special keyword `proceed()` in around advice.
- No such special keyword exists in @AspectJ styled around advice.
- You must take the same approach as with `thisJoinPoint` and related variables.
- If you need to proceed with the original join point, you must declare the method to take a parameter of the `ProceedingJoinPoint` type (which extends `JoinPoint`).
- The `ProceedingJoinPoint` interface provides two methods: `proceed()` and `proceed(Object[])`.
- The no-argument `proceed()` proceeds with the original join point with unaltered join point context (the execution object, method arguments, and so on).

Proceeding With The Original Join Point Execution

```
package ajia.monitoring;
import ...
@Aspect
public abstract class Monitoring {
  @Pointcut
  public abstract void monitored();

  @Around("monitored()")
  public Object measureTime(ProceedingJoinPoint pjp) throws
   Throwable{
    long startTime = System.nanoTime();
    Object ret = pjp.proceed();
    long endTime = System.nanoTime();
    System.out.println("Method "+
   pjp.getSignature().toShortString()+ " took " + (endTime-
   startTime));
    return ret;
  }
}
```

Altering The Join Point Context

- In some cases of the around advice, the original join point must be invoked with an altered context.
- In these cases, the form `proceed(Object[])` must be used to pass the new context.
- The array passed to the call must contain elements in the following order:
  - The `this` object or its replacement (only if you collected context using `this()`).
  - The `target` object or its replacement (only if you collected context using `target()`).
  - The join point's arguments or its replacements, in the same order as needed by the join point (even if you did not use the `args()` pointcut to collect them).
  - A simple way to pass all arguments is to use the `getArgs()` method on the `ProceedingJoinPoint` object.

Altering The Join Point Context

- Eg. Utility method for altering the context.
- The method expects that the caller passes in the objects to be used as this, target, and arguments. If you do not collect either of the first two, the caller passes a null for them:

```
public static Object[] formProceedArguments(Object thiz, Object
   target, Object[] arguments) {
  int argumentsOffset = 0;
  if(thiz != null) { argumentsOffset++; }
  if(target != null) { argumentsOffset++; }
  Object[] jpContext = new Object[arguments.length + argumentsOffset];
  int currentIndex = 0;
  if(thiz != null) { jpContext[currentIndex++] = thiz; }
  if(target != null) { jpContext[currentIndex++] = target; }
  System.arraycopy(arguments, 0, jpContext, argumentsOffset,
   arguments.length);
  return jpContext;
}
```

Altering The Join Point Context

- This scheme of proceeding with altered context is different from that used in the traditional style, where you pass objects matching the collected context.
- The @AspectJ style is especially weak when you need to alter the context collected by arguments, because you are forced to know the exact position of the argument you want to alter, in turn forcing the advice to know too much about the join point.

Mapping static crosscutting

- Static crosscutting comes in many forms:
  - introducing members,
  - declaring new types as parent type,
  - attaching annotations,
  - compile-time errors and warnings,
  - and exception softening.
- Static crosscutting significantly relies on compile-time behavior.
- Not all features can be implemented in @AspectJ.

Mapping weave-time declarations

- You declare weave-time errors and warnings in @AspectJ by declaring a static final member of the String type annotated with a @DeclareError or @DeclareWarning annotation.
- The value of the string is the message emitted by the weaver upon detecting the occurrence of a matching join point; the annotation attribute specifies the pointcut for which errors and warning are to be emitted.

```
@DeclareError("callToUnsafeCode()")
static final String unsafeCodeUsageError= "This third-
   party...";

@DeclareWarning("callToBlockingOperations()")
static final String blockingCallFromAWTWarning
= "Please ensure you are not calling this from the AWT thread";
```

Mapping weave-time declarations

- AspectJ requires you to mark the field associated with the @DeclareError or @DeclareWarning as static and final.
- It ensures that the variable remains unmodified during program execution.
- Confusion may exist between the message in the source file and its runtime value.
- The string specified must be a literal and not a result of a call to a static method. This constraint ensures that the weaver can access the message string without executing any methods.

//Error

```
@DeclareWarning("callToBlockingOperations()")
static final String blockingCallFromAWTWarning=
   Warnings.blockCallWarning();
```

Mapping declare parents

- The @AspectJ syntax offers limited but useful support for static-crosscutting constructs aimed at type modification.
- With the traditional syntax, it is possible to use declare parents statements to add new parent types (class or interface) to a set of existing types.
- An AspectJ weaver can then take into account the effect of such statements during code compilation.
- The support for declaring parents comes through the @DeclareMixin annotation.

Mapping declare parents

- This annotation mixes an interface into a matching set of types and delegates the implementation to a specified object.
- To mix in a set of types, you annotate a factory method with this annotation and specify a type-pattern to select types being mixed-in as its value attribute.
- The return type of the method (which must be an interface) is mixed in with the matched types. The AspectJ weaver uses the object returned by the method to delegate the implementation for the mixed-in interface.
- @AspectJ syntax also supports the @DeclareParents annotation.
- The @DeclareParents cannot implement the exact equivalent of the declare parents construct and may mislead developers into believing they are equivalent.
- @DeclareMixin is the preferred approach.
- In the future, @DeclareParents may be deprecated.

Declaring Parents For Marker Interfaces

- Declaring parents for marker interfaces (interfaces without any methods) requires that the factory method return a null (you may return any other object, but since there is nothing to delegate to, the object will remain unused).
- Eg. It declares all the types in the ajia.banking.domain package to implement java.io.Serializable:

```
@DeclareMixin("ajia.banking.domain.*")
public Serializable serializableMixin() {
   return null;
}
```

- The declaration is equivalent to

```
declare parents: ajia.banking.domain.* implements Serializable;
```

Declaring Parents For Non-marker Interfaces

- For a non marker interface, either the classes selected by the type pattern must already implement the methods for the interface or the factory method must return an object to delegate the implementations for the interface methods.

```
public interface Identifiable {
  public Long getId();
  public void setId(Long id);
}
```

- All classes in `ajia.banking.domain.*` must already implement getId and setId.

```
@Aspect
public class AccountTracking {
  @DeclareMixin("ajia.banking.domain.*")  //
   public Identifiable identifiableMixin() {
    return null;
  }
   ... advice
}
```

Requiring classes to already include an implementation is not useful in most situations.

Declaring Parents For Non-marker Interfaces

- The method annotated with @DeclareMixin annotation must return an object that is used as the delegate.

```
@Aspect
public class AccountTracking {
  @DeclareMixin("ajia.banking.domain.*")
   public Identifiable identifiableMixin() {
     return new IdentifiableDefaultImpl();
   }
}
```

- The AspectJ weaver uses the return object to delegate implementation of the interface to the types specified.
- The annotated factory method is public and may be declared static.
- If it is an instance method, it can use aspect instance members while creating the delegate object.

Declaring Parents For Non-marker Interfaces

- The method annotated with the @DeclareMixin annotation can declare a single parameter.
- AspectJ passes the object being mixed in as that parameter.
- You can pass this parameter to the delegate object being created if it needs access to the object being mixed in.

```
@DeclareMixin("ajia.banking.domain.*")
public Auditor auditorMixin(Object mixedIn) {
   return new AuditorImpl(mixedIn);
}
```

Using the Introduced Type

- With a @DeclareMixin statement, the weaver modifies the byte code for the child types to make them implement the parent type.
- The Java compiler does not know this. You will get compilation errors.
- You can fix these errors by specifying a typecast.

```
import ...
public class Main {
public static void main(String[] args) {
  Account account = new Account(1);
  Identifiable identifiableAccount = (Identifiable)account;
  identifiableAccount.setId(6L);
  System.out.println("Id is " + identifiableAccount.getId());
}
```

Using The Introduced Type

- Usually, an advice will use the new parent type.
- You can avoid typecasts by using an appropriate context-collecting pointcut.

```
import ...
@Aspect
public class AccountTracking {
  @DeclareMixin("ajia.banking.domain.*")
  public Identifiable identifiableMixin() {
    return new IdentifiableDefaultImpl();
  }
  @AfterReturning("execution(* ajia.banking.domain.*.*(..))"
   + " && this(identif)")
  public void track(Identifiable identif) {
    System.out.println("Object with id " + identif.getId());
  }
}
```

Features not implemented in @AspectJ

- A few features:
  - introducing data and methods,
  - softening exceptions,
  - and privileged aspects

  will never be implemented due to the fundamental constraints of compiling code using a plain Java compiler.
- Associating annotations is not implemented in the current version.
- There is not an elegant mapping that allows compilation using javac and performs as much compile-time checking as reasonably possible.
- In a future version, this feature may be implemented.