The AspectJ™ 5 Development Kit Developer's Notebook

Table of Contents

by the AspectJ Team

This guide describes the changes to the AspectJ language in AspectJ 5. These include support for Java 5 features, support for an annotation-based development style for aspects and new reflection and tools APIs. If you are new to AspectJ, we recommend you start by reading the programming guide.

Join Point Signatures

Many of the extensions to the AspectJ language to address the new features of Java 5 are derived from a simple set of principles for join point matching. In this section, we outline these principles as a foundation for understanding the matching rules in the presence of annotations, generics, covariance, varargs, and autoboxing.

Join Point Matching

AspectJ supports 11 different kinds of join points. These are the `method call`, `method execution`, `constructor call`, `constructor execution`, `field get`, `field set`, `pre-initialization`, `initialization`, `static initialization`, `handler`, and `advice execution` join points.

The *kinded* pointcut designators match based on the kind of a join point. These are the `call`, `execution`, `get`, `set`, `preinitialization`, `initialization`, `staticinitialization`, `handler`, and `adviceexecution` designators.

A kinded pointcut is written using patterns, some of which match based on *signature*, and some of which match based on *modifiers*. For example, in the `call` pointcut designator:

```
call(ModifierPattern TypePattern TypePattern.IdPattern(TypePatternList) ThrowsPattern)
```

the modifiers matching patterns are `ModifierPattern` and `ThrowsPattern`, and the signature matching patterns are `TypePattern TypePattern.IdPattern(TypePatternList)`.

A join point has potentially multiple signatures, but only one set of modifiers. *A kinded primitive pointcut matches a particular join point if and only if*:

1. They are of the same kind
2. The signature pattern (exactly) matches at least one signature of the join point
3. The modifiers pattern matches the modifiers of the subject of the join point

These rules make it very easily to quickly determine whether a given pointcut matches a given join point. In the next two sections, we describe what the signature(s) of a join point are, and what the subjects of join points are.

Join Point Signatures

Call, execution, get, and set join points may potentially have multiple signatures. All other join points have exactly one signature. The following table summarizes the constituent parts of a join point signature for the different kinds of join point.

Join Point Kind	Return Type	Declaring Type	Id	Parameter Types	Field Type	Exception Type
Method call	+	+	+	+		

Join Point Kind	Return Type	Declaring Type	Id	Parameter Types	Field Type	Exception Type
Method execution	+	+	+	+		
Constructor call		+		+		
Constructor execution		+		+		
Field get		+	+		+	
Field set		+	+		+	
Pre-initialization		+		+		
Initialization		+		+		
Static initialization		+				
Handler						+
Advice execution		+		+		

Note that whilst an advice execution join point has a signature comprising the declaring type of the advice and the advice parameter types, the `adviceexecution` pointcut designator does not support matching based on this signature.

The signatures for most of the join point kinds should be self-explanatory, except for field get and set, and method call and execution join points, which can have multiple signatures. Each signature of a method call or execution join point has the same id and parameter types, but the declaring type and return type (with covariance) may vary. Each signature of a field get or set join point has the same id and field type, but the declaring type may vary.

The following sections examine signatures for these join points in more detail.

Method call join point signatures

For a call join point where a call is made to a method `m(parameter_types)` on a target type `T` (where `T` is the static type of the target):

```
T t = new T();
t.m("hello");  // <= call join point occurs when this line is executed
```

Then the signature `R(T) T.m(parameter_types)` is a signature of the call join point, where `R(T)` is the return type of `m` in `T`, and `parameter_types` are the parameter types of `m`. If `T` itself does not declare a definition of `m(parameter_types)`, then `R(T)` is the return type in the definition of `m` that `T` inherits. Given the call above, and the definition of `T.m`:

```
interface Q {
  R m(String s);
}

class P implements Q {
  R m(String s) {...}
}

class S extends P {
  R' m(String s) {...}
}

class T extends S {}
```

Then `R' T.m(String)` is a signature of the call join point for `t.m("hello")`.

For each ancestor (super-type) `A` of `T`, if `m(parameter_types)` is defined for that super-type, then `R(A) A.m(parameter_types)` is a signature of the call join point, where `R(A)` is the return type of `m(parameter_types)` as defined in `A`, or as inherited by `A` if `A` itself does not provide a definition of `m(parameter_types)`.

Continuing the example from above,we can deduce that

```
R' S.m(String)
R  P.m(String)
R  Q.m(String)
```

are all additional signatures for the call join point arising from the call `t.m("hello")`. Thus this call join point has four signatures in total. Every signature has the same id and parameter types, and a different declaring type.

Method execution join point signatures

Join point signatures for execution join points are defined in a similar manner to signatures for call join points. Given the hierarchy:

```
interface Q {
  R m(String s);
}

class P implements Q {
  R m(String s) {...}
}

class S extends P {
  R' m(String s) {...}
}
```

```
class T extends S { }

class U extends T {
  R' m(String s) {...}
}
```

Then the execution join point signatures arising as a result of the call to `u.m("hello")` are:

```
R' U.m(String)
R' S.m(String)
R  P.m(String)
R  Q.m(String)
```

Each signature has the same id and parameter types, and a different declaring type. There is one signature for each type that provides its own declaration of the method. Hence in this example there is no signature `R' T.m(String)` as `T` does not provide its own declaration of the method.

Field get and set join point signatures

For a field get join point where an access is made to a field `f` of type `F` on a object with declared type `T`, then `F T.f` is a signature of the get join point.

If `T` does not directly declare a member `f`, then for each super type `S` of `T`, up to and including the most specific super type of `T` that does declare the member `f`, `F S.f` is a signature of the join point. For example, given the hierarchy:

```
class P  {
  F f;
}

class S extends P {
  F f;
}

class T extends S { }
```

Then the join point signatures for a field get join point of the field `f` on an object with declared type `T` are:

```
F S.f
F T.f
```

The signatures for a field set join point are derived in an identical manner.

Join Point Modifiers

Every join point has a single set of modifiers - these include the standard Java modifiers such as `public`, `private`, `static`, `abstract` etc., any annotations, and the `throws` clauses of methods and constructors. These modifiers are the modifiers of the *subject* of the join point.

The following table defines the join point subject for each kind of join point.

Join Point Kind	Subject
Method call	The method picked out by Java as the static target of the method call.
Method execution	The method that is executing.
Constructor call	The constructor being called.
Constructor execution	The constructor executing.
Field get	The field being accessed.
Field set	The field being set.
Pre-initialization	The first constructor executing in this constructor chain.
Initialization	The first constructor executing in this constructor chain.
Static initialization	The type being initialized.
Handler	The declared type of the exception being handled.
Advice execution	The advice being executed.

For example, given the following types

```
public class X {
  @Foo
  protected void doIt() {...}
}

public class Y extends X {
  public void doIt() {...}
}
```

Then the modifiers for a call to `(Y y) y.doIt()` are simply `{ public }`. The modifiers for a call to `(X x) x.doIt()` are `{ @Foo, protected }`.

Summary of Join Point Matching

A join point has potentially multiple signatures, but only one set of modifiers. *A kinded primitive pointcut matches a particular join point if and only if*:

1. They are of the same kind
2. The signature pattern (exactly) matches at least one signature of the join point
3. The modifiers pattern matches the modifiers of the subject of the join point

Given the hierarchy

```
interface Q {
  R m(String s);
}

class P implements Q {
  @Foo
  public R m(String s) {...}
}

class S extends P {
  @Bar
  public R' m(String s) {...}
}

class T extends S {}
```

and the program fragment:

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

The the pointcut `call(@Foo R P.m(String))` matches the call `p.m("hello")` since both the signature and the modifiers match. It does not match the call `s.m("hello")` because even though the signature pattern matches one of the signatures of the join point, the modifiers pattern does not match the modifiers of the method m in S which is the static target of the call.

The pointcut `call(R' m(String))` matches the calls `t.m("hello")` and `s.m("hello")`. It does not match the call `p.m("hello")` since the signature pattern does not match any signature for the call join point of m in P.

Annotations

Annotations in Java 5

This section provides the essential information about annotations in Java 5 needed to understand how annotations are treated in AspectJ 5. For a full introduction to annotations in Java, please see the documentation for the Java 5 SDK.

Using Annotations

Java 5 introduces *annotation types* which can be used to express metadata relating to program members in the form of *annotations*. Annotations in Java 5 can be applied to package and type declarations (classes, interfaces, enums, and annotations), constructors, methods, fields, parameters, and variables. Annotations are specified in the program source by using the `@` symbol. For example, the following piece of code uses the `@Deprecated` annotation to indicate that the `obsoleteMethod()` has been deprecated:

```
@Deprecated
public void obsoleteMethod() { ... }
```

Annotations may be *marker annotations*, *single-valued annotations*, or *multi-valued annotations*. Annotation types with no members or that provide default values for all members may be used simply as marker annotations, as in the deprecation example above. Single-value annotation types have a single member, and the annotation may be written in one of two equivalent forms:

```
@SuppressWarnings({"unchecked"})
public void someMethod() {...}
```

or

```
@SuppressWarnings(value={"unchecked"})
public void someMethod() {...}
```

Multi-value annotations must use the `member-name=value ` syntax to specify annotation values. For example:

```
@Authenticated(role="supervisor",clearanceLevel=5)
public void someMethod() {...}
```

Retention Policies

Annotations can have one of three retention policies:

Source-file retention

Annotations with source-file retention are read by the compiler during the compilation process, but are not rendered in the generated `.class` files.

Class-file retention

This is the default retention policy. Annotations with class-file retention are read by the compiler and also retained in the generated `.class` files.

Runtime retention

Annotations with runtime retention are read by the compiler, retained in the generated `.class` files, and also made available at runtime.

Local variable annotations are not retained in class files (or at runtime) regardless of the retention policy set on the annotation type. See JLS 9.6.1.2.

Accessing Annotations at Runtime

Java 5 supports a new interface, `java.lang.reflect.AnnotatedElement`, that is implemented by the reflection classes in Java (`Class`, `Constructor`, `Field`, `Method`, and `Package`). This interface gives you access to annotations *that have runtime retention* via the `getAnnotation`, `getAnnotations`, and `isAnnotationPresent`. Because annotation types are just regular Java classes, the annotations returned by these methods can be queried just like any regular Java object.

Annotation Inheritance

It is important to understand the rules relating to inheritance of annotations, as these have a bearing on join point matching based on the presence or absence of annotations.

By default annotations are *not* inherited. Given the following program

```
@MyAnnotation
class Super {
  @Oneway public void foo() {}
}

class Sub extends Super {
  public void foo() {}
}
```

Then `Sub` *does not* have the `MyAnnotation` annotation, and `Sub.foo()` is not an `@Oneway` method, despite the fact that it overrides `Super.foo()` which is.

If an annotation type has the meta-annotation `@Inherited` then an annotation of that type on a *class* will cause the annotation to be inherited by sub-classes. So, in the example above, if the `MyAnnotation` type had the `@Inherited` attribute, then `Sub` would have the `MyAnnotation` annotation.

`@Inherited` annotations are not inherited when used to annotate anything other than a type. A type that implements one or more interfaces never inherits any annotations from the interfaces it

implements.

Annotating Aspects

AspectJ 5 supports annotations on aspects, and on method, field, constructor, advice, and inter-type declarations within aspects. Method and advice parameters may also be annotated. Annotations are not permitted on pointcut declarations or on `declare` statements.

The following example illustrates the use of annotations in aspects:

```
@AspectAnnotation
public abstract aspect ObserverProtocol {

    @InterfaceAnnotation
    interface Observer {}

    @InterfaceAnnotation
    interface Subject {}

    @ITDFieldAnnotation
    private List<Observer> Subject.observers;

    @ITDMethodAnnotation
    public void Subject.addObserver(Observer o) {
      observers.add(o);
    }

    @ITDMethodAnnotation
    public void Subject.removeObserver(Observer o) {
      observers.remove(o);
    }

    @MethodAnnotation
    private void notifyObservers(Subject subject) {
      for(Observer o : subject.observers)
        notifyObserver(o,subject);
    }

    /**
     * Delegate to concrete sub-aspect the actual form of
     * notification for a given type of Observer.
     */
    @MethodAnnotation
    protected abstract void notifyObserver(Observer o, Subject s);

    /* no annotations on pointcuts */
    protected abstract pointcut observedEvent(Subject subject);

    @AdviceAnnotation
    after(Subject subject) returning : observedEvent(subject) {
```

```
        notifyObservers(subject);
    }
}
```

An annotation on an aspect will be inherited by sub-aspects, iff it has the `@Inherited` meta-annotation.

AspectJ 5 supports a new XLint warning, "the pointcut associated with this advice does not match any join points". The warning is enabled by default and will be emitted by the compiler if the pointcut expression associated with an advice statement can be statically determined to not match any join points. The warning can be suppressed for an individual advice statement by using the `@SuppressAjWarnings({"adviceDidNotMatch"})` annotation. This works in the same way as the Java 5 SuppressWarnings annotation (See JLS 9.6.1.5), but has class file retention.

```
import org.aspectj.lang.annotation.SuppressAjWarnings;

public aspect AnAspect {

  pointcut anInterfaceOperation() : execution(* AnInterface.*(..));

  @SuppressAjWarnings // may not match if there are no implementers of the
interface...
  before() : anInterfaceOperation() {
     // do something...
  }

  @SuppressAjWarnings("adviceDidNotMatch") // alternate form
  after() returning : anInterfaceOperation() {
     // do something...
  }
}
```

Join Point Matching based on Annotations

This section discusses changes to type pattern and signature pattern matching in AspectJ 5 that support matching join points based on the presence or absence of annotations. We then discuss means of exposing annotation values within the body of advice.

Annotation Patterns

For any kind of annotated element (type, method, constructor, package, etc.), an annotation pattern can be used to match against the set of annotations on the annotated element.An annotation pattern element has one of two basic forms:

- `@<qualified-name>`, for example, `@Foo`, or `@org.xyz.Foo`.
- `@(<type-pattern>)`, for example, `@(org.xyz..*)`, or `@(Foo || Boo)`

These simple elements may be negated using `!`, and combined by simple concatentation. The pattern `@Foo @Boo` matches an annotated element that has both an annotation of type `Foo` and an annotation of type `Boo`.

Some examples of annotation patterns follow:

`@Immutable`

Matches any annotated element which has an annotation of type `Immutable`.

`!@Persistent`

Matches any annotated element which does not have an annotation of type `Persistent`.

`@Foo @Goo`

Matches any annotated element which has both an annotation of type `Foo` and an annotation of type `Goo`.

`@(Foo || Goo)`

Matches any annotated element which has either an annotation of a type matching the type pattern `(Foo || Goo)`. In other words, an annotated element with either an annotation of type `Foo` or an annotation of type `Goo` (or both). (The parenthesis are required in this example).

`@(org.xyz..*)`

Matches any annotated element which has either an annotation of a type matching the type pattern `(org.xyz..*)`. In other words, an annotated element with an annotation that is declared in the org.xyz package or a sub-package. (The parenthesis are required in this example).

Type Patterns

AspectJ 1.5 extends type patterns to allow an optional `AnnotationPattern` prefix.

```
TypePattern := SimpleTypePattern |
               '!' TypePattern |
               '(' AnnotationPattern? TypePattern ')'
               TypePattern '&&' TypePattern |
               TypePattern '||' TypePattern

SimpleTypePattern := DottedNamePattern '+'? '[]'*

DottedNamePattern := FullyQualifiedName RestOfNamePattern? |
                     '*' NotStarNamePattern?

RestOfNamePattern := '..' DottedNamePattern |
                     '*' NotStarNamePattern?

NotStarNamePattern := FullyQualifiedName RestOfNamePattern? |
                      '..' DottedNamePattern

FullyQualifiedName := JavaIdentifierCharacter+ ('.' JavaIdentifierCharacter+)*
```

Note that in most cases when annotations are used as part of a type pattern, the parenthesis are required (as in `(@Foo Hello+)`). In some cases (such as a type pattern used within a `within` or `handler` pointcut expression), the parenthesis are optional:

```
OptionalParensTypePattern := AnnotationPattern? TypePattern
```

The following examples illustrate the use of annotations in type patterns:

`(@Immutable *)`

Matches any type with an `@Immutable` annotation.

`(!@Immutable *)`

Matches any type which does not have an `@Immutable` annotation.

`(@Immutable (org.xyz.* || org.abc.*))`

Matches any type in the `org.xyz` or `org.abc` packages with the `@Immutable` annotation.

`((@Immutable Foo+) || Goo)`

Matches a type `Foo` or any of its subtypes, which have the `@Immutable` annotation, or a type `Goo`.

`((@(Immutable || NonPersistent) org.xyz..*)`

Matches any type in a package beginning with the prefix `org.xyz`, which has either the `@Immutable` annotation or the `@NonPersistent` annotation.

`(@Immutable @NonPersistent org.xyz..*)`

Matches any type in a package beginning with the prefix `org.xyz`, which has both an `@Immutable` annotation and an `@NonPersistent` annotation.

`(@(@Inherited ) org.xyz..)`

Matches any type in a package beginning with the prefix `org.xyz`, which has an inheritable annotation. The annotation pattern `@(@Inherited *)` matches any annotation of a type matching the type pattern `@Inherited *`, which in turn matches any type with the `@Inherited` annotation.

Signature Patterns

Field Patterns

A `FieldPattern` can optionally specify an annotation-matching pattern as the first element:

```
FieldPattern :=
    AnnotationPattern? FieldModifiersPattern?
    TypePattern (TypePattern DotOrDotDot)? SimpleNamePattern

FieldModifiersPattern := '!'? FieldModifier FieldModifiersPattern*

FieldModifier := 'public' | 'private' | 'protected' | 'static' |
                 'transient' | 'final'
```

```
DotOrDotDot := '.' | '..'

SimpleNamePattern := JavaIdentifierChar+ ('*' SimpleNamePattern)?
```

If present, the `AnnotationPattern` restricts matches to fields with annotations that match the pattern. For example:

`@SensitiveData * *`

Matches a field of any type and any name, that has an annotation of type `@SensitiveData`

`@SensitiveData List org.xyz...`

Matches a member field of a type in a package with prefix `org.xzy`, where the field is of type `List`, and has an annotation of type `@SensitiveData`

`(@SensitiveData ) org.xyz...*`

Matches a member field of a type in a package with prefix `org.xzy`, where the field is of a type which has a `@SensitiveData` annotation.

`@Foo (@Goo ) (@Hoo *).`

Matches a field with an annotation `@Foo`, of a type with an annotation `@Goo`, declared in a type with annotation `@Hoo`.

`@Persisted @Classified * *`

Matches a field with an annotation `@Persisted` and an annotation `@Classified`.

Method and Constructor Patterns

A `MethodPattern` can optionally specify an annotation-matching pattern as the first element.

```
MethodPattern :=
    AnnotationPattern? MethodModifiersPattern? TypePattern
                       (TypePattern DotOrDotDot)? SimpleNamePattern
                       '(' FormalsPattern ')'ThrowsPattern?

MethodModifiersPattern := '!'? MethodModifier MethodModifiersPattern*

MethodModifier := 'public' | 'private' | 'protected' | 'static' |
                  'synchronized' | 'final'

FormalsPattern := '..' (',' FormalsPatternAfterDotDot)* |
                  OptionalParensTypePattern (',' FormalsPattern)* |
                  TypePattern '...'

FormalsPatternAfterDotDot :=
        OptionalParensTypePattern (',' FormalsPatternAfterDotDot)* |
        TypePattern '...'

ThrowsPattern := 'throws' TypePatternList
```

```
TypePatternList := TypePattern (',' TypePattern)*
```

A `ConstructorPattern` has the form

```
ConstructorPattern :=
    AnnotationPattern? ConstructorModifiersPattern?
                       (TypePattern DotOrDotDot)? 'new' '(' FormalsPattern ')'
                       ThrowsPattern?

ConstructorModifiersPattern := '!'? ConstructorModifier ConstructorModifiersPattern*

ConstructorModifier := 'public' | 'private' | 'protected'
```

The optional `AnnotationPattern` at the beginning of a method or constructor pattern restricts matches to methods/constructors with annotations that match the pattern. For example:

`@Oneway * *(..)`

Matches a method with any return type and any name, that has an annotation of type `@Oneway`.

`@Transaction * (@Persistent org.xyz..).(..)`

Matches a method with the `@Transaction` annotation, declared in a type with the `@Persistent` annotation, and in a package beginning with the `org.xyz` prefix.

`* .(@Immutable *,..)`

Matches any method taking at least one parameter, where the parameter type has an annotation `@Immutable`.

Example Pointcuts

`within(@Secure *)`

Matches any join point where the code executing is declared in a type with an `@Secure` annotation. The format of the `within` pointcut designator in AspectJ 5 is `'within' '(' OptionalParensTypePattern ')'`.

`staticinitialization(@Persistent *)`

Matches the staticinitialization join point of any type with the `@Persistent` annotation. The format of the `staticinitialization` pointcut designator in AspectJ 5 is `'staticinitialization' '(' OptionalParensTypePattern ')'`.

`call(@Oneway * *(..))`

Matches a call to a method with a `@Oneway` annotation.

`execution(public (@Immutable ) org.xyz...*(..))`

The execution of any public method in a package with prefix `org.xyz`, where the method returns an immutable result.

`set(@Cachable * *)`

Matches the set of any cachable field.

`handler(!@Catastrophic *)`

Matches the handler join point for the handling of any exception that is not `Catastrophic`. The format of the `handler` pointcut designator in AspectJ 5 is `'handler' '(' OptionalParensTypePattern ')'`.

Runtime type matching and context exposure

AspectJ 5 supports a set of "@" pointcut designators which can be used both to match based on the presence of an annotation at runtime, and to expose the annotation value as context in a pointcut or advice definition. These designators are `@args, @this, @target, @within, @withincode`, and `@annotation`

It is a compilation error to attempt to match on an annotation type that does not have runtime retention using `@this, @target` or `@args`. It is a compilation error to attempt to use any of these designators to expose an annotation value that does not have runtime retention.

The `this()`, `target()`, and `args()` pointcut designators allow matching based on the runtime type of an object, as opposed to the statically declared type. In AspectJ 5, these designators are supplemented with three new designators : `@this()` (read, "this annotation"), `@target()`, and `@args()`.

Like their counterparts, these pointcut designators can be used both for join point matching, and to expose context. The format of these new designators is:

```
AtThis := '@this' '(' AnnotationOrIdentifer ')'

AtTarget := '@target' '(' AnnotationOrIdentifier ')'

AnnotationOrIdentifier := FullyQualifiedName | Identifier

AtArgs := '@args' '(' AnnotationsOrIdentifiersPattern ')'

AnnotationsOrIdentifiersPattern :=
                  '..' (',' AnnotationsOrIdentifiersPatternAfterDotDot)? |
                  AnnotationOrIdentifier (',' AnnotationsOrIdentifiersPattern)* |
                  '*' (',' AnnotationsOrIdentifiersPattern)*

AnnotationsOrIdentifiersPatternAfterDotDot :=
              AnnotationOrIdentifier (',' AnnotationsOrIdentifiersPatternAfterDotDot)*
|
              '*' (',' AnnotationsOrIdentifiersPatternAfterDotDot)*
```

The forms of `@this()` and `@target()` that take a single annotation name are analogous to their counterparts that take a single type name. They match at join points where the object bound to `this` (or `target`, respectively) has an annotation of the specified type. For example:

`@this(Foo)`

Matches any join point where the object currently bound to 'this' has an annotation of type `Foo`.

`call(* *(..)) && @target(Classified)`

Matches a call to any object where the target of the call has a `@Classified` annotation.

Annotations can be exposed as context in the body of advice by using the forms of `@this()`, `@target()` and `@args()` that use bound variables in the place of annotation names. For example:

```
pointcut callToClassifiedObject(Classified classificationInfo) :
    call(* *(..)) && @target(classificationInfo);

pointcut txRequiredMethod(Tx transactionAnnotation) :
    execution(* *(..)) && @this(transactionAnnotation)
    && if(transactionAnnotation.policy() == TxPolicy.REQUIRED);
```

The `@args` pointcut designator behaves as its `args` counterpart, matching join points based on number and position of arguments, and supporting the `*` wildcard and at most one `..` wildcard. An annotation at a given position in an `@args` expression indicates that the runtime type of the argument in that position at a join point must have an annotation of the indicated type. For example:

```
/**
 * matches any join point with at least one argument, and where the
 * type of the first argument has the @Classified annotation
 */
pointcut classifiedArgument() : @args(Classified,..);

/**
 * matches any join point with three arguments, where the third
 * argument has an annotation of type @Untrusted.
 */
pointcut untrustedData(Untrusted untrustedDataSource) :
    @args(*,*,untrustedDataSource);
```

In addition to accessing annotation information at runtime through context binding, access to `AnnotatedElement` information is also available reflectively with the body of advice through the `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart` variables. To access annotations on the arguments, or object bound to this or target at a join point you can use the following code fragments:

```
Annotation[] thisAnnotations = thisJoinPoint.getThis().getClass().getAnnotations();
Annotation[] targetAnnotations = thisJoinPoint.getTarget().getClass().getAnnotations(
);
Annotation[] firstParamAnnotations = thisJoinPoint.getArgs()[0].getClass()
.getAnnotations();
```

The `@within` and `@withincode` pointcut designators match any join point where the executing code is defined within a type (`@within`), or a method/constructor (`@withincode`) that has an annotation of the specified type. The form of these designators is:

```
AtWithin := '@within' '(' AnnotationOrIdentifier ')'
AtWithinCode := '@withincode' '(' AnnotationOrIdentifier ')'
```

Some examples of using these designators follow:

`@within(Foo)`

> Matches any join point where the executing code is defined within a type which has an annotation of type `Foo`.

`pointcut insideCriticalMethod(Critical c) : @withincode(c);`

> Matches any join point where the executing code is defined in a method or constructor which has an annotation of type `@Critical`, and exposes the value of the annotation in the parameter `c`.

The `@annotation` pointcut designator matches any join point where the *subject* of the join point has an annotation of the given type. Like the other @pcds, it can also be used for context exposure.

```
AtAnnotation := '@annotation' '(' AnnotationOrIdentifier ')'
```

The subject of a join point is defined in the table in chapter one of this guide.

Access to annotation information on members at a matched join point is also available through the `getSignature` method of the `JoinPoint` and `JoinPoint.StaticPart` interfaces. The `Signature` interfaces are extended with additional operations that provide access to the `java.lang.reflect` `Method, Field` and `Constructor` objects on which annnotations can be queried. The following fragment illustrates an example use of this interface to access annotation information.

```
Signature sig = thisJoinPointStaticPart.getSignature();
AnnotatedElement declaringTypeAnnotationInfo = sig.getDeclaringType();
if (sig instanceof MethodSignature) {
  // this must be a call or execution join point
  Method method = ((MethodSignature)sig).getMethod();
}
```

Note again that it would be nicer to add the method getAnnotationInfo directly to MemberSignature, but this would once more couple the runtime library to Java 5.

The `@this,@target` and `@args` pointcut designators can only be used to match against annotations that have runtime retention. The `@within, @withincode` and `@annotation` pointcut designators can only be used to match against annotations that have at least class-file retention, and if used in the binding form the annotation must have runtime retention.

Package and Parameter Annotations

Matching on package annotations is not supported in AspectJ. Support for this capability may be considered in a future release.

Parameter annotation matching is being added in AspectJ1.6. Initially only matching is supported but binding will be implemented at some point. Whether the annotation specified in a pointcut should be considered to be an annotation on the parameter type or an annotation on the parameter itself is determined through the use of parentheses around the parameter type. Consider the following:

```
@SomeAnnotation
class AnnotatedType {}

class C {
  public void foo(AnnotatedType a) {}
  public void goo(@SomeAnnotation String s) {}
}
```

The method foo has a parameter of an annotated type, and can be matched by this pointcut:

```
pointcut p(): execution(* *(@SomeAnnotation *));
```

When there is a single annotation specified like this, it is considered to be part of the type pattern in the match against the parameter: 'a parameter of any type that has the annotation @SomeAnnotation'.

To match the parameter annotation case, the method goo, this is the pointcut:

```
pointcut p(): execution(* *(@SomeAnnotation (*)));
```

The use of parentheses around the wildcard is effectively indicating that the annotation should be considered separately to the type pattern for the parameter type: 'a parameter of any type that has a parameter annotation of @SomeAnnotation'.

To match when there is a parameter annotation and an annotation on the type as well:

```
pointcut p(): execution(* *(@SomeAnnotation (@SomeOtherAnnotation *)));
```

The parentheses are grouping @SomeOtherAnnotation with the * to form the type pattern for the parameter, then the type @SomeAnnotation will be treated as a parameter annotation pattern.

Annotation Inheritance and pointcut matching

According to the Java 5 specification, non-type annotations are not inherited, and annotations on types are only inherited if they have the `@Inherited` meta-annotation. Given the following program:

```
class C1 {
  @SomeAnnotation
  public void aMethod() {...}
}

class C2 extends C1 {
  public void aMethod() {...}
}

class Main {
  public static void main(String[] args) {
    C1 c1 = new C1();
    C2 c2 = new C2();
    c1.aMethod();
    c2.aMethod();
  }
}

aspect X {
  pointcut annotatedC2MethodCall() :
    call(@SomeAnnotation * C2.aMethod());

  pointcut annotatedMethodCall() :
    call(@SomeAnnotation * aMethod());
}
```

The pointcut `annotatedC2MethodCall` will not match anything since the definition of `aMethod` in `C2` does not have the annotation.

The pointcut `annotatedMethodCall` matches `c1.aMethod()` but not `c2.aMethod()`. The call to `c2.aMethod` is not matched because join point matching for modifiers (the visibility modifiers, annotations, and throws clause) is based on the subject of the join point (the method actually being called).

Matching based on annotation values

The `if` pointcut designator can be used to write pointcuts that match based on the values annotation members. For example:

```
pointcut txRequiredMethod(Tx transactionAnnotation) :
    execution(* *(..)) && @this(transactionAnnotation)
    && if(transactionAnnotation.policy() == TxPolicy.REQUIRED);
```

Using Annotations with declare statements

Declare error and declare warning

Since pointcut expressions in AspectJ 5 support join point matching based on annotations, this

facility can be exploited when writing `declare warning` and `declare error` statements. For example:

```
declare warning : withincode(@PerformanceCritical * *(..)) &&
                  call(@ExpensiveOperation * *(..))
                : "Expensive operation called from within performance critical
section";
```

```
declare error : call(* org.xyz.model.*.*(..)) &&
                !@within(Trusted)
                : "Untrusted code should not call the model classes directly";
```

declare parents

The general form of a `declare parents` statement is:

```
declare parents : TypePattern extends Type;
declare parents : TypePattern implements TypeList;
```

Since AspectJ 5 supports annotations as part of a type pattern specification, it is now possible to match types based on the presence of annotations *with either class-file or runtime retention*. For example:

`declare parents : (@Secured *) implements SecuredObject;`

All types with the `@Secured` annotation implement the `SecuredObject` inteface.

`declare parents : (@Secured BankAccount+) implements SecuredObject;`

The subset of types drawn from the `BankAccount` type and any subtype of `BankAccount`, where the `@Secured` annotation is present, implement the `SecuredObject` interface.

An annotation type may not be used as the target of a declare parents statement. If an annotation type is named explicitly as the target of a declare parents statement, a compilation error will result. If an annotation type is matched by a non-explicit type pattern used in a declare parents statement it will be ignored (and an XLint warning issued).

declare precedence

The general form of a declare precedence statement is:

```
declare precedence : TypePatList;
```

AspectJ 5 allows the type patterns in the list to include annotation information as part of the pattern specification. For example:

`declare precedence : (@Security ),;`

All aspects with the `@Security` annotation take precedence over any other aspects in the system.

(Or, more informally, all security-related aspects take precedence).

Declare Annotation

AspectJ 5 supports a new kind of declare statement, `declare annotation`. This takes different forms according to the recipient of the annotation: `declare @type` for types, `declare @method` for methods, `declare @constructor` for constructors, and `declare @field` for fields. `declare @package` may be supported in a future release.

The general form is:

```
declare @<kind> : ElementPattern : Annotation ;
```

Where annotation is a regular annotation expression as defined in the Java 5 language. If the annotation has the `@Target` meta-annotation, then the elements matched by `ElementPattern` must be of the kind specified by the `@Target` annotation.

`ElementPattern` is defined as follows:

```
ElementPattern := TypePattern |
                  MethodPattern |
                  ConstructorPattern |
                  FieldPattern
```

The following examples illustrate the use of `declare annotation`.

`declare @type : org.xyz.model..* : @BusinessDomain ;`

All types defined in a package with the prefix `org.xyz.model` have the `@BusinessDomain` annotation. declare @method : public * BankAccount+.*(..) :

`@Secured(role="supervisor")`

All public methods in `BankAccount` and its subtypes have the annotation `@Secured(role="supervisor")`. declare @constructor : BankAccount+.new(..) :

`@Secured(role="supervisor")`

All constructors in `BankAccount` and its subtypes have the annotation `@Secured(role="supervisor")`.

`declare @field : * DAO+.* : @Persisted;`

All fields defined in `DAO` or its subtypes have the `@Persisted` annotation.

Inter-type Declarations

An annotation type may not be the target of an inter-type declaration.

Generics

Generics in Java 5

This section provides the essential information about generics in Java 5 needed to understand how generics are treated in AspectJ 5. For a full introduction to generics in Java, please see the documentation for the Java 5 SDK.

Declaring Generic Types

A generic type is declared with one or more type parameters following the type name. By convention formal type parameters are named using a single letter, though this is not required. A simple generic list type (that can contain elements of any type `E`) could be declared:

```
interface List<E> {
   Iterator<E> iterator();
   void add(E anItem);
   E remove(E anItem);
}
```

It is important to understand that unlike template mechanisms there will only be one type, and one class file, corresponding to the `List` interface, regardless of how many different instantiations of the `List` interface a program has (each potentially providing a different value for the type parameter `E`). A consequence of this is that you cannot refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable.

A *parameterized type* is an invocation of a generic type with concrete values supplied for all of its type parameters (for example, `List<String>` or `List<Food>`).

A generic type may be declared with multiple type parameters. In addition to simple type parameter names, type parameter declarations can also constrain the set of types allowed by using the `extends` keyword. Some examples follow:

`class Foo<T> {…}`

A class `Foo` with one type parameter, `T`.

`class Foo<T,S> {…}`

A class `Foo` with two type parameters, `T` and `S`.

`class Foo<T extends Number> {…}`

A class `Foo` with one type parameter `T`, where `T` must be instantiated as the type `Number` or a subtype of `Number`.

`class Foo<T, S extends T> {…}`

A class `Foo` with two type parameters, `T` and `S`. `Foo` must be instantiated with a type `S` that is a subtype of the type specified for parameter `T`.

`class Foo<T extends Number & Comparable> {…}`

A class `Foo` with one type parameter, `T`. `Foo` must be instantiated with a type that is a subtype of `Number` and that implements `Comparable`.

Using Generic and Parameterized Types

You declare a variable (or a method/constructor argument) of a parameterized type by specifying a concrete type specfication for each type parameter in the generic type. The following example declares a list of strings and a list of numbers:

```
List<String> strings;
List<Number> numbers;
```

It is also possible to declare a variable of a generic type without specifying any values for the type parameters (a *raw* type). For example, `List strings`. In this case, unchecked warnings may be issued by the compiler when the referenced object is passed as a parameter to a method expecting a parameterized type such as a `List<String>`. New code written in the Java 5 language would not be expected to use raw types.

Parameterized types are instantiated by specifying type parameter values in the constructor call expression as in the following examples:

```
List<String> strings = new MyListImpl<String>();
List<Number> numbers = new MyListImpl<Number>();
```

When declaring parameterized types, the `?` wildcard may be used, which stands for "some type". The `extends` and `super` keywords may be used in conjunction with the wildcard to provide upper and lower bounds on the types that may satisfy the type constraints. For example:

`List<?>`

A list containing elements of some type, the type of the elements in the list is unknown.

`List<? extends Number>`

A list containing elements of some type that extends Number, the exact type of the elements in the list is unknown.

`List<? super Double>`

A list containing elements of some type that is a super-type of Double, the exact type of the elements in the list is unknown.

A generic type may be extended as any other type. Given a generic type `Foo<T>` then a subtype `Goo` may be declared in one of the following ways:

`class Goo extends Foo`

Here `Foo` is used as a raw type, and the appropriate warning messages will be issued by the compiler on attempting to invoke methods in `Foo`.

`class Goo<E> extends Foo`

`Goo` is a generic type, but the super-type `Foo` is used as a raw type and the appropriate warning messages will be issued by the compiler on attempting to invoke methods defined by `Foo`.

`class Goo<E> extends Foo<E>`

This is the most usual form. `Goo` is a generic type with one parameter that extends the generic type `Foo` with that same parameter. So `Goo<String<` is a subclass of `Foo<String>`.

`class Goo<E,F> extends Foo<E>`

`Goo` is a generic type with two parameters that extends the generic type `Foo` with the first type parameter of `Goo` being used to parameterize `Foo`. So `Goo<String,Integer<` is a subclass of `Foo<String>`.

`class Goo extends Foo<String>`

`Goo` is a type that extends the parameterized type `Foo<String>`.

A generic type may implement one or more generic interfaces, following the type binding rules given above. A type may also implement one or more parameterized interfaces (for example, `class X implements List<String>`, however a type may not at the same time be a subtype of two interface types which are different parameterizations of the same interface.

Subtypes, Supertypes, and Assignability

The supertype of a generic type `C` is the type given in the extends clause of `C`, or `Object` if no extends clause is present. Given the type declaration

```
public interface List<E> extends Collection<E> {... }
```

then the supertype of `List<E>` is `Collection<E>`.

The supertype of a parameterized type `P` is the type given in the extends clause of `P`, or `Object` if no extends clause is present. Any type parameters in the supertype are substituted in accordance with the parameterization of `P`. An example will make this much clearer: Given the type `List<Double>` and the definition of the `List` given above, the direct supertype is `Collection<Double>`. `List<Double>` is *not* considered to be a subtype of `List<Number>`.

An instance of a parameterized type `P<T1,T2,⋯Tn>`may be assigned to a variable of the same type or a supertype without casting. In addition it may be assigned to a variable `R<S1,S2,⋯Sm>` where `R` is a supertype of `P` (the supertype relationship is reflexive), `m` ⇐ `n`, and for all type parameters `S1..m`, `Tm` equals `Sm` *or* `Sm` is a wildcard type specification and `Tm` falls within the bounds of the wildcard. For example, `List<String>` can be assigned to a variable of type `Collection<?>`, and `List<Double>` can be assigned to a variable of type `List<? extends Number>`.

Generic Methods and Constructors

A static method may be declared with one or more type parameters as in the following declaration:

```
static <T> T first(List<T> ts) { ... }
```

Such a definition can appear in any type, the type parameter `T` does not need to be declared as a type parameter of the enclosing type.

Non-static methods may also be declared with one or more type parameters in a similar fashion:

```
<T extends Number> T max(T t1, T t2) { ... }
```

The same technique can be used to declare a generic constructor.

Erasure

Generics in Java are implemented using a technique called *erasure*. All type parameter information is erased from the run-time type system. Asking an object of a parameterized type for its class will return the class object for the raw type (eg. `List` for an object declared to be of type `List<String>`. A consequence of this is that you cannot at runtime ask if an object is an `instanceof` a parameterized type.

Generics in AspectJ 5

AspectJ 5 provides full support for all of the Java 5 language features, including generics. Any legal Java 5 program is a legal AspectJ 5 progam. In addition, AspectJ 5 provides support for generic and parameterized types in pointcuts, inter-type declarations, and declare statements. Parameterized types may freely be used within aspect members, and support is also provided for generic *abstract* aspects.

Matching generic and parameterized types in pointcut expressions

The simplest way to work with generic and parameterized types in pointcut expressions and type patterns is simply to use the raw type name. For example, the type pattern `List` will match the generic type `List<E>` and any parameterization of that type (`List<String>, List<?>, List<? extends Number>` and so on. This ensures that pointcuts written in existing code that is not generics-aware will continue to work as expected in AspectJ 5. It is also the recommended way to match against generic and parameterized types in AspectJ 5 unless you explicitly wish to narrow matches to certain parameterizations of a generic type.

Generic methods and constructors, and members defined in generic types, may use type variables as part of their signature. For example:

```
public class Utils {

  /** static generic method */
  static <T> T first(List<T> ts) { ... }

  /** instance generic method */
```

```
  <T extends Number> T max(T t1, T t2) { ... }

}

public class G<T> {

   // field with parameterized type
   T myData;

   // method with parameterized return type
   public List<T> getAllDataItems() {...}

}
```

AspectJ 5 does not allow the use of type variables in pointcut expressions and type patterns. Instead, members that use type parameters as part of their signature are matched by their *erasure*. Java 5 defines the rules for determing the erasure of a type as follows.

Let `|T|` represent the erasure of some type `T`. Then:

- The erasure of a parameterized type `T<T1,⋯,Tn>` is `|T|`. For example, the erasure of `List<String>` is `List`.
- The erasure of a nested type `T.C` is `|T|.C`. For example, the erasure of the nested type `Foo<T>.Bar` is `Foo.Bar`.
- The erasure of an array type `T[]` is `|T|[]`. For example, the erasure of `List<String>[]` is `List[]`.
- The erasure of a type variable is its leftmost bound. For example, the erasure of a type variable `P` is `Object`, and the erasure of a type variable `N extends Number` is `Number`.
- The erasure of every other type is the type itself.

Applying these rules to the earlier examples, we find that the methods defined in `Utils` can be matched by a signature pattern matching `static Object Utils.first(List)` and `Number Utils.max(Number, Number)` respectively. The members of the generic type `G` can be matched by a signature pattern matching `Object G.myData` and `public List G.getAllDataItems()` respectively.

Restricting matching using parameterized types

Pointcut matching can be further restricted to match only given parameterizations of parameter types (methods and constructors), return types (methods) and field types (fields). This is achieved by specifying a parameterized type pattern at the appropriate point in the signature pattern. For example, given the class `Foo`:

```
public class Foo {

  List<String> myStrings;
  List<Float>  myFloats;

  public List<String> getStrings() { return myStrings; }
  public List<Float> getFloats() { return myFloats; }
```

```
  public void addStrings(List<String> evenMoreStrings) {
     myStrings.addAll(evenMoreStrings);
  }

}
```

Then a `get` join point for the field `myStrings` can be matched by the pointcut `get(List Foo.myStrings)` and by the pointcut `get(List<String> Foo.myStrings)`, but *not* by the pointcut `get(List<Number> *)`.

A `get` join point for the field `myFloats` can be matched by the pointcut `get(List Foo.myFloats)`, the pointcut `get(List<Float> *)`, and the pointcut `get(List<Number+> *)`. This last example shows how AspectJ type patterns can be used to match type parameters types just like any other type. The pointcut `get(List<Double> *)` does *not* match.

The execution of the methods `getStrings` and `getFloats` can be matched by the pointcut expression `execution(List get*(..))`, and the pointcut expression `execution(List<*> get*(..))`, but only `getStrings` is matched by `execution(List<String> get*(..))` and only `getFloats` is matched by `execution(List<Number+> get*(..))`

A call to the method `addStrings` can be matched by the pointcut expression `call(* addStrings(List))` and by the expression `call(* addStrings(List<String>))`, but *not* by the expression `call(* addStrings(List<Number>))`.

Remember that any type variable reference in a generic member is *always* matched by its erasure. Thus given the following example:

```
class G<T> {
    List<T> foo(List<String> ls) { return null; }
}
```

The execution of `foo` can be matched by `execution(List foo(List))`, `execution(List foo(List<String>>))`, and `execution(* foo(List<String<))`but not by `execution(List<Object> foo(List<String>>)` since the erasure of `List<T>` is `List` and not `List<Object>`.

Generic wildcards and signature matching

When it comes to signature matching, a type parameterized using a generic wildcard is a distinct type. For example, `List<?>` is a very different type to `List<String>`, even though a variable of type `List<String>` can be assigned to a variable of type `List<?>`. Given the methods:

```
class C {
  public void foo(List<? extends Number> listOfSomeNumberType) {}
  public void bar(List<?> listOfSomeType) {}
  public void goo(List<Double> listOfDoubles) {}
}
```

`execution(* C.*(List))`

Matches an execution join point for any of the three methods.

`execution(* C.*(List<? extends Number>))`

matches only the execution of `foo`, and *not* the execution of `goo` since `List<? extends Number>` and `List<Double>` are distinct types.

`execution(* C.*(List<?>))`

matches only the execution of `bar`.

`execution(* C.*(List<? extends Object+>))`

matches both the execution of `foo` and the execution of `bar` since the upper bound of `List<?>` is implicitly `Object`.

Treatment of bridge methods

Under certain circumstances a Java 5 compiler is required to create *bridge methods* that support the compilation of programs using raw types. Consider the types

```
class Generic<T> {
  public T foo(T someObject) {
    return someObject;
  }
}

class SubGeneric<N extends Number> extends Generic<N> {
  public N foo(N someNumber) {
    return someNumber;
  }
}
```

The class `SubGeneric` extends `Generic` and overrides the method `foo`. Since the upper bound of the type variable `N` in `SubGeneric` is different to the upper bound of the type variable `T` in `Generic`, the method `foo` in `SubGeneric` has a different erasure to the method `foo` in `Generic`. This is an example of a case where a Java 5 compiler will create a *bridge method* in `SubGeneric`. Although you never see it, the bridge method will look something like this:

```
public Object foo(Object arg) {
  Number n = (Number) arg; // "bridge" to the signature defined in this type
return foo(n);
}
```

Bridge methods are synthetic artefacts generated as a result of a particular compilation strategy and have no execution join points in AspectJ 5. So the pointcut `execution(Object SubGeneric.foo(Object))` does not match anything. (The pointcut `execution(Object Generic.foo(Object))` matches the execution of `foo` in both `Generic` and `SubGeneric` since both are implementations of `Generic.foo`).

It *is* possible to *call* a bridge method as the following short code snippet demonstrates. Such a call *does* result in a call join point for the call to the method.

```
SubGeneric rawType = new SubGeneric();
rawType.foo("hi");  // call to bridge method (will result in a runtime failure in this
case)
Object n = new Integer(5);
rawType.foo(n);     // call to bridge method that would succeed at runtime
```

Runtime type matching with this(), target() and args()

The `this()`, `target()`, and `args()` pointcut expressions all match based on the runtime type of their arguments. Because Java 5 implements generics using erasure, it is not possible to ask at runtime whether an object is an instance of a given parameterization of a type (only whether or not it is an instance of the erasure of that parameterized type). Therefore AspectJ 5 does not support the use of parameterized types with the `this()` and `target()` pointcuts. Parameterized types may however be used in conjunction with `args()`. Consider the following class

```
public class C {
  public void foo(List<String> listOfStrings) {}

  public void bar(List<Double> listOfDoubles) {}

  public void goo(List<? extends Number> listOfSomeNumberType) {}
}
```

`args(List)`

will match an execution or call join point for any of these methods

`args(List<String>)`

will match an execution or call join point for `foo`.

`args(List<Double>)`

matches an execution or call join point for `bar`, and *may* match at an execution or call join point for `goo` since it is legitimate to pass an object of type `List<Double>` to a method expecting a `List<? extends Number>`.

In this situation, a runtime test would normally be applied to ascertain whether or not the argument was indeed an instance of the required type. However, in the case of parameterized types such a test is not possible and therefore AspectJ 5 considers this a match, but issues an *unchecked* warning. For example, compiling the aspect `A` below with the class `C` produces the compilation warning: `unchecked match of List<Double> with List<? extends Number> when argument is an instance of List at join point method-execution(void C.goo(List<? extends Number>)) [Xlint:uncheckedArgument]`;

```
public aspect A {
   before(List<Double> listOfDoubles) : execution(* C.*(..)) && args(listOfDoubles) {
```

```
      for (Double d : listOfDoubles) {
         // do something
      }
   }
}
```

Like all Lint messages, the `uncheckedArgument` warning can be configured in severity from the default warning level to error or even ignore if preferred. In addition, AspectJ 5 offers the annotation `@SuppressAjWarnings` which is the AspectJ equivalent of Java's `@SuppressWarnings` annotation. If the advice is annotated with `@SuppressWarnings` then *all* lint warnings issued during matching of pointcut associated with the advice will be suppressed. To suppress just an `uncheckedArgument` warning, use the annotation `@SuppressWarnings("uncheckedArgument")` as in the following examples:

```
import org.aspectj.lang.annotation.SuppressAjWarnings
public aspect A {
   @SuppressAjWarnings   // will not see *any* lint warnings for this advice
   before(List<Double> listOfDoubles) : execution(* C.*(..)) && args(listOfDoubles) {
      for (Double d : listOfDoubles) {
         // do something
      }
   }

   @SuppressAjWarnings("uncheckedArgument")   // will not see *any* lint warnings for
this advice
   before(List<Double> listOfDoubles) : execution(* C.*(..)) && args(listOfDoubles) {
      for (Double d : listOfDoubles) {
         // do something
      }
   }
}
```

The safest way to deal with `uncheckedArgument` warnings however is to restrict the pointcut to match only at those join points where the argument is guaranteed to match. This is achieved by combining `args` with a `call` or `execution` signature matching pointcut. In the following example the advice will match the execution of `bar` but not of `goo` since the signature of `goo` is not matched by the execution pointcut expression.

```
public aspect A {
   before(List<Double> listOfDoubles) : execution(* C.*(List<Double>)) && args
(listOfDoubles) {
      for (Double d : listOfDoubles) {
         // do something
      }
   }
}
```

Generic wildcards can be used in args type patterns, and matching follows regular Java 5 assignability rules. For example, `args(List<?>)` will match a list argument of any type, and `args(List<? extends Number>)` will match an argument of type `List<Number>`, `List<Double>`, `List<Float>` and so on. Where a match cannot be fully statically determined, the compiler will once more issue an `uncheckedArgument` warning.

Consider the following program:

```
public class C {
   public static void main(String[] args) {
      C c = new C();
      List<String> ls = new ArrayList<String>();
      List<Double> ld = new ArrayList<Double>();
      c.foo("hi");
      c.foo(ls);
      c.foo(ld);
   }

   public void foo(Object anObject) {}
}

aspect A {
    before(List<? extends Number> aListOfSomeNumberType)
      : call(* foo(..)) && args(aListOfSomeNumberType) {
       // process list...
    }
}
```

From the signature of `foo` all we know is that the runtime argument will be an instance of `Object`.Compiling this program gives the unchecked argument warning: `unchecked match of List<? extends Number> with List when argument is an instance of List at join point method-execution(void C.foo(Object)) [Xlint:uncheckedArgument]`. The advice will not execute at the call join point for `c.foo("hi")` since `String` is not an instance of `List`. The advice *will* execute at the call join points for `c.foo(ls)` and `c.foo(ld)` since in both cases the argument is an instance of `List`.

Combine a wildcard argument type with a signature pattern to avoid unchecked argument matches. In the example below we use the signature pattern `List<Number+>` to match a call to any method taking a `List<Number>`, `List<Double>`, `List<Float>` and so on. In addition the signature pattern `List<? extends Number+>` can be used to match a call to a method declared to take a `List<? extends Number>`, `List<? extends Double>` and so on. Taken together, these restrict matching to only those join points at which the argument is guaranteed to be an instance of `List<? extends Number>`.

```
aspect A {
    before(List<? extends Number> aListOfSomeNumberType)
      : (call(* foo(List<Number+>)) || call(* foo(List<? extends Number+>)))
        && args(aListOfSomeNumberType) {
        // process list...
    }
```

```
}
```

Binding return values in after returning advice

After returning advice can be used to bind the return value from a matched join point. AspectJ 5 supports the use of a parameterized type in the returning clause, with matching following the same rules as described for args. For example, the following aspect matches the execution of any method returning a `List`, and makes the returned list available to the body of the advice.

```
public aspect A {
  pointcut executionOfAnyMethodReturningAList() : execution(List *(..));

  after() returning(List<?> listOfSomeType) : executionOfAnyMethodReturningAList() {
    for (Object element : listOfSomeType) {
       // process element...
    }
  }
}
```

The pointcut uses the raw type pattern `List`, and hence it matches methods returning any kind of list (`List<String>`, `List<Double>`, and so on). We've chosen to bind the returned list as the parameterized type `List<?>` in the advice since Java's type checking will now ensure that we only perform safe operations on the list.

Given the class

```
public class C {
  public List<String> foo(List<String> listOfStrings) {...}
  public List<Double> bar(List<Double> listOfDoubles) {...}
  public List<? extends Number> goo(List<? extends Number> listOfSomeNumberType) {...}
}
```

The advice in the aspect below will run after the execution of `bar` and bind the return value. It will also run after the execution of `goo` and bind the return value, but gives an `uncheckedArgument` warning during compilation. It does *not* run after the execution of `foo`.

```
public aspect Returning {
  after() returning(List<Double> listOfDoubles) : execution(* C.*(..)) {
     for(Double d : listOfDoubles) {
        // process double...
     }
  }
}
```

As with `args` you can guarantee that after returning advice only executes on lists *statically determinable* to be of the right type by specifying a return type pattern in the associated pointcut.

The `@SuppressAjWarnings` annotation can also be used if desired.

Declaring pointcuts inside generic types

Pointcuts can be declared in both classes and aspects. A pointcut declared in a generic type may use the type variables of the type in which it is declared. All references to a pointcut declared in a generic type from outside of that type must be via a parameterized type reference, and not a raw type reference.

Consider the generic type `Generic` with a pointcut `foo`:

```
public class Generic<T> {
   /**
    * matches the execution of any implementation of a method defined for T
    */
   public pointcut foo() : execution(* T.*(..));
}
```

Such a pointcut must be refered to using a parameterized reference as shown below.

```
public aspect A {
  // runs before the execution of any implementation of a method defined for MyClass
  before() : Generic<MyClass>.foo() {
     // ...
  }

  // runs before the execution of any implementation of a method defined for YourClass
  before() : Generic<YourClass>.foo() {
      // ...
  }

  // results in a compilation error - raw type reference
  before() : Generic.foo() { }
}
```

Inter-type Declarations

AspectJ 5 supports the inter-type declaration of generic methods, and of members on generic types. For generic methods, the syntax is exactly as for a regular method declaration, with the addition of the target type specification:

`<T extends Number> T Utils.max(T first, T second) {…}`

Declares a generic instance method `max` on the class `Util`. The `max` method takes two arguments, `first` and `second` which must both be of the same type (and that type must be `Number` or a subtype of `Number`) and returns an instance of that type.

`static <E> E Utils.first(List<E> elements) {…}`

Declares a static generic method `first` on the class `Util`. The `first` method takes a list of

elements of some type, and returns an instance of that type.

<T> Sorter.new(List<T> elements,Comparator<? super T> comparator) {⋯}

Declares a constructor on the class `Sorter`. The constructor takes a list of elements of some type, and a comparator that can compare instances of the element type.

A generic type may be the target of an inter-type declaration, used either in its raw form or with type parameters specified. If type parameters are specified, then the number of type parameters given must match the number of type parameters in the generic type declaration. Type parameter *names* do not have to match. For example, given the generic type `Foo<T,S extends Number>` then:

`String Foo.getName() {⋯}`

Declares a `getName` method on behalf of the type `Foo`. It is not possible to refer to the type parameters of Foo in such a declaration.

`public R Foo<Q, R>.getMagnitude() {⋯}`

Declares a method `getMagnitude` on the generic class `Foo`. The method returns an instance of the type substituted for the second type parameter in an invocation of `Foo` If `Foo` is declared as `Foo<T,N extends Number> {⋯}` then this inter-type declaration is equivalent to the declaration of a method `public N getMagnitude()` within the body of `Foo`.

`R Foo<Q, R extends Number>.getMagnitude() {⋯}`

Results in a compilation error since a bounds specification is not allowed in this form of an inter-type declaration (the bounds are determined from the declaration of the target type).

A parameterized type may not be the target of an inter-type declaration. This is because there is only one type (the generic type) regardless of how many different invocations (parameterizations) of that generic type are made in a program. Therefore it does not make sense to try and declare a member on behalf of (say) `Bar<String>`, you can only declare members on the generic type `Bar<T>`.

Declare Parents

Both generic and parameterized types can be used as the parent type in a `declare parents` statement (as long as the resulting type hierarchy would be well-formed in accordance with Java's sub-typing rules). Generic types may also be used as the target type of a `declare parents` statement.

`declare parents: Foo implements List<String>`

The `Foo` type implements the `List<String>` interface. If `Foo` already implements some other parameterization of the `List` interface (for example, `List<Integer>` then a compilation error will result since a type cannot implement multiple parameterizations of the same generic interface type.

Declare Soft

It is an error to use a generic or parameterized type as the softened exception type in a declare soft statement. Java 5 does not permit a generic class to be a direct or indirect subtype of `Throwable` (JLS 8.1.2).

Generic Aspects

AspectJ 5 allows an *abstract* aspect to be declared as a generic type. Any concrete aspect extending a generic abstract aspect must extend a parameterized version of the abstract aspect. Wildcards are not permitted in this parameterization.

Given the aspect declaration:

```
public abstract aspect ParentChildRelationship<P,C> {
    // ...
}
```

then

`public aspect FilesInFolders extends ParentChildRelationship<Folder,File> {⋯`

declares a concrete sub-aspect, `FilesInFolders` which extends the parameterized abstract aspect `ParentChildRelationship<Folder,File>`.

`public aspect FilesInFolders extends ParentChildRelationship {⋯`

results in a compilation error since the `ParentChildRelationship` aspect must be fully parameterized.

`public aspect ThingsInFolders<T> extends ParentChildRelationship<Folder,T>`

results in a compilation error since concrete aspects may not have type parameters.

`public abstract aspect ThingsInFolders<T> extends ParentChildRelationship<Folder,T>`

declares a sub-aspect of `ParentChildRelationship` in which `Folder` plays the role of parent (is bound to the type variable `P`).

The type parameter variables from a generic aspect declaration may be used in place of a type within any member of the aspect, *except for within inter-type declarations*. For example, we can declare a `ParentChildRelationship` aspect to manage the bi-directional relationship between parent and child nodes as follows:

```
/**
 * a generic aspect, we've used descriptive role names for the type variables
 * (Parent and Child) but you could use anything of course
 */
public abstract aspect ParentChildRelationship<Parent,Child> {

  /** generic interface implemented by parents */
  interface ParentHasChildren<C extends ChildHasParent>{
    List<C> getChildren();
    void addChild(C child);
    void removeChild(C child);
  }

  /** generic interface implemented by children */
```

```
interface ChildHasParent<P extends ParentHasChildren>{
  P getParent();
  void setParent(P parent);
}

/** ensure the parent type implements ParentHasChildren<child type> */
declare parents: Parent implements ParentHasChildren<Child>;

/** ensure the child type implements ChildHasParent<parent type> */
declare parents: Child implements ChildHasParent<Parent>;

// Inter-type declarations made on the *generic* interface types to provide
// default implementations.

/** list of children maintained by parent */
private List<C> ParentHasChildren<C>.children = new ArrayList<C>();

/** reference to parent maintained by child */
private P ChildHasParent<P>.parent;

/** Default implementation of getChildren for the generic type ParentHasChildren */
public List<C> ParentHasChildren<C>.getChildren() {
      return Collections.unmodifiableList(children);
}

/** Default implementation of getParent for the generic type ChildHasParent */
public P ChildHasParent<P>.getParent() {
     return parent;
}

/**
  * Default implementation of addChild, ensures that parent of child is
  * also updated.
  */
public void ParentHasChildren<C>.addChild(C child) {
     if (child.parent != null) {
       child.parent.removeChild(child);
     }
     children.add(child);
     child.parent = this;
  }

 /**
   * Default implementation of removeChild, ensures that parent of
   * child is also updated.
   */
 public void ParentHasChildren<C>.removeChild(C child) {
     if (children.remove(child)) {
       child.parent = null;
     }
  }
```

```
    /**
      * Default implementation of setParent for the generic type ChildHasParent.
      * Ensures that this child is added to the children of the parent too.
      */
    public void ChildHasParent<P>.setParent(P parent) {
       parent.addChild(this);
    }

    /**
      * Matches at an addChild join point for the parent type P and child type C
      */
    public pointcut addingChild(Parent p, Child c) :
      execution(* ParentHasChildren.addChild(ChildHasParent)) && this(p) && args(c);

    /**
      * Matches at a removeChild join point for the parent type P and child type C
      */
    public pointcut removingChild(Parent p, Child c) :
      execution(* ParentHasChildren.removeChild(ChildHasParent)) && this(p) && args(c
);

}
```

The example aspect captures the protocol for managing a bi-directional parent-child relationship between any two types playing the role of parent and child. In a compiler implementation managing an abstract syntax tree (AST) in which AST nodes may contain other AST nodes we could declare the concrete aspect:

```
public aspect ASTNodeContainment extends ParentChildRelationship<ASTNode,ASTNode> {
    before(ASTNode parent, ASTNode child) : addingChild(parent, child) {
      // ...
    }
}
```

As a result of this declaration, `ASTNode` gains members:

- `List<ASTNode> children`
- `ASTNode parent`
- `List<ASTNode>getChildren()`
- `ASTNode getParent()`
- `void addChild(ASTNode child)`
- `void removeChild(ASTNode child)`
- `void setParent(ASTNode parent)`

In a system managing orders, we could declare the concrete aspect:

```
public aspect OrderItemsInOrders extends ParentChildRelationship<Order, OrderItem> {}
```

As a result of this declaration, `Order` gains members:

- `List<OrderItem> children`
- `List<OrderItem> getChildren()`
- `void addChild(OrderItem child)`
- `void removeChild(OrderItem child)`

and `OrderItem` gains members:

- `Order parent`
- `Order getParent()`
- `void setParent(Order parent)`

A second example of an abstract aspect, this time for handling exceptions in a uniform manner, is shown below:

```
abstract aspect ExceptionHandling<T extends Throwable> {

  /**
   * method to be implemented by sub-aspects to handle thrown exceptions
   */
  protected abstract void onException(T anException);

  /**
   * to be defined by sub-aspects to specify the scope of exception handling
   */
  protected abstract pointcut inExceptionHandlingScope();

  /**
   * soften T within the scope of the aspect
   */
  declare soft: T : inExceptionHandlingScope();

  /**
   * bind an exception thrown in scope and pass it to the handler
   */
  after() throwing (T anException) : inExceptionHandlingScope() {
    onException(anException);
  }

}
```

Notice how the type variable `T extends Throwable` allows the components of the aspect to be designed to work together in a type-safe manner. The following concrete sub-aspect shows how the

abstract aspect might be extended to handle `IOExceptions`.

```
public aspect IOExceptionHandling extends ExceptionHandling<IOException>{

  protected pointcut inExceptionHandlingScope() :
    call(* doIO*(..)) && within(org.xyz..*);

  /**
   * called whenever an IOException is thrown in scope.
   */
  protected void onException(IOException ex) {
    System.err.println("handled exception: " + ex.getMessage());
    throw new MyDomainException(ex);
  }
}
```

Autoboxing and Unboxing

Autoboxing and Unboxing in Java 5

Java 5 (and hence AspectJ 1.5) supports automatic conversion of primitive types (`int`, `float`, `double` etc.) to their object equivalents (`Integer`, `Float`, `Double` etc.) in assignments and method and constructor invocations. This conversion is know as autoboxing.

Java 5 also supports automatic unboxing, where wrapper types are automatically converted into their primitive equivalents if needed for assignments or method or constructor invocations.

For example:

```
int i = 0;
i = new Integer(5);  // auto-unboxing
Integer i2 = 5;      // autoboxing
```

Autoboxing and Join Point matching in AspectJ 5

Most of the pointcut designators match based on signatures, and hence are unaffected by autoboxing. For example, a call to a method

```
public void foo(Integer i);
```

is *not* matched by a pointcut `call(void foo(int))` since the signature declares a single `Integer` parameter, not an `int`.

The `args` pointcut designator is affected by autoboxing since it matches based on the runtime type of the arguments. AspectJ 5 applies autoboxing and unboxing in determining argument matching. In other words, `args(Integer)` will match any join point at which there is a single argument of type `Integer` or of type `int`.

- `args(Integer)` and `args(int)` are equivalent
- `args(Float)` and `args(float)` are equivalent
- `args(Double)` and `args(double)` are equivalent
- `args(Short)` and `args(short)` are equivalent
- `args(Byte)` and `args(byte)` are equivalent
- `args(Long)` and `args(long)` are equivalent
- `args(Boolean)` and `args(boolean)` are equivalent

Autoboxing and unboxing are also applied when binding pointcut or advice parameters, for example:

```
pointcut foo(int i) : args(i);

before(Integer i) : foo(i) {
  // ...
}
```

Inter-type method declarations and method dispatch

Autoboxing, unboxing, and also varargs all affect the method dispatch algorithm used in Java 5. In AspectJ 5, the target method of a call is selected according to the following algorithm:

1. Attempt to locate a matching method or inter-type declared method without considering autoboxing, unboxing, or vararg invocations.
2. If no match is found, try again considering autoboxing and unboxing.
3. Finally try again considering both autoboxing, unboxing, and varargs.

One consequence is that a directly matching inter-type declared method will take precedence over a method declared locally in the target class but that only matches via autoboxing.

Covariance

Covariance in Java 5

Java 5 (and hence AspectJ 5) allows you to narrow the return type in an overriding method. For example:

```
class A {
  public A whoAreYou() {...}
}

class B extends A {
  // override A.whoAreYou *and* narrow the return type.
  public B whoAreYou() {...}
}
```

Covariant methods and Join Point matching

The join point matching rules for `call` and `execution` pointcut designators are extended to match against covariant methods.

Given the classes `A` and `B` as defined in the previous section, and the program fragment

```
A a = new A();
B b = new B();
a.whoAreYou();
b.whoAreYou();
```

The signatures for the call join point `a.whoAreYou()` are simply:

```
A A.whoAreYou()
```

The signatures for the call join point `b.whoAreYou()` are:

```
A A.whoAreYou()
B B.whoAreYou()
```

Following the join point matching rules given in Join Point Signatures.

`call(* whoAreYou())`
: Matches both calls, (since each call join point has at least one matching signature).

`call(* A.whoAreYou())`
: Matches both calls, (since each call join point has at least one matching signature).

`call(A whoAreYou())`

Matches both calls, (since each call join point has at least one matching signature).

`call(A B.whoAreYou())`

Does not match anything - neither of the call join points has a signature matched by this pattern. A lint warning is given for the call `a.whoAreYou()` ("does not match because declaring type is `A`, if match required use `target(B)`").

`call(A+ B.whoAreYou())`

Matches the call to `b.whoAreYou()` since the signature pattern matches the signature `B B.whoAreYou()`. A lint warning is given for the call `a.whoAreYou()` ("does not match because declaring type is `A`, if match required use `target(B)`").

`call(B A.whoAreYou())`

Does not match anything since neither join point has a signature matched by this pattern.

`call(B whoAreYou())`

Matches the call to `b.whoAreYou()` only.

`call(B B.whoAreYou())`

Matches the call to `b.whoAreYou()` only.

The rule for signature matching at call and execution join points is unchanged from AspectJ 1.2: a call or execution pointcut matches if the signature pattern matches at least one of the signatures of the join point, and if the modifiers of the method or constructor are matched by any modifier pattern or annotation pattern that may be present.

Varargs

Variable-length Argument Lists in Java 5

Java 5 (and hence AspectJ 5) allows you to specify methods that take a variable number of arguments of a specified type. This is achieved using an ellipsis (…) in the method signature as shown:

```
public void foo(int i, String... strings) {}
```

A method or constructor may take at most one variable length argument, and this must always be the last declared argument in the signature.

Calling Methods and Constructors with variable-length arguments

A *varargs* method may be called with zero or more arguments in the variable argument position. For example, given the definition of `foo` above, the following calls are all legal:

```
foo(5);
foo(5,"One String");
foo(7,"One String","Two Strings");
foo(3,"One String","Two Strings","Three Strings");
```

A *varargs* parameter is treated as an array within the defining member. So in the body of `foo` we could write for example:

```
public void foo(int i, String... strings) {
  String[] someStrings = strings;
  // rest of method body
}
```

One consequence of this treatment of a varargs parameter as an array is that you can also call a varargs method with an array:

```
foo(7,new String[] {"One String","Two Strings"});
```

Using Variable-length arguments in advice and pointcut expressions

AspectJ 5 allows variable-length arguments to be used for methods declared within aspects, and for inter-type declared methods and constructors, in accordance with the rules outlined in the previous section.

AspectJ 5 also allows variable length arguments to be matched by pointcut expressions and bound as formals in advice.

Matching signatures based on variable length argument types

Recall from the definition of signature patterns given in the chapter on annotations (Signature Patterns), that `MethodPattern` and `ConstructorPattern` are extended to allow a `varargs` pattern in the last argument position of a method or constructor signature.

```
FormalsPattern :=
  '..' (',' FormalsPatternAfterDotDot)? |
  OptionalParensTypePattern (',' FormalsPattern)* |
  TypePattern '...'

FormalsPatternAfterDotDot :=
  OptionalParensTypePattern (',' FormalsPatternAfterDotDot)* |
  TypePattern '...'
```

Method and constructor patterns are used in the `call`, `execution`, `initialization`, `preinitialization`, and `withincode` pointcut designators. Some examples of usage follow:

`call(* org.xyz..(int, String…))`

Matches a call join point for a call to a method defined in the `org.xyz` package, taking an `int` and a `String` *vararg*.

`execution(* org.xyz..(Integer…))`

Matches an execution join point for the execution of a method defined in the `org.xyz` package, taking an `Integer` *vararg*.

`initialization(org.xyz.*.newFoo || Goo)…`

Matches the initialization join point for the construction of an object in the `org.xyz` package via a constructor taking either a variable number of `Foo` parameters or a variable number of `Goo` parameters. (This example illustrating the use of a type pattern with `…`).

A variable argument parameter and an array parameter are treated as distinct signature elements, so given the method definitions:

```
void foo(String...);
void bar(String[]);
```

The pointcut `execution(* .(String…))` matches the execution join point for `foo`, but not `bar`. The pointcut `execution(* .(String[]))` matches the execution join point for `bar` but not `foo`.

Exposing variable-length arguments as context in pointcuts and advice

When a varargs parameter is used within the body of a method, it has an array type, as discussed in the introduction to this section. We follow the same convention when binding a varargs parameter

via the `args` pointcut designator. Given a method

```
public void foo(int i, String... strings) {}
```

The call or execution join points for `foo` will be matched by the pointcut `args(int,String[])`. It is not permitted to use the varargs syntax within an args pointcut designator - so you *cannot* write `args(int,String⋯)`.

Binding of a varargs parameter in an advice statement is straightforward:

```
before(int i, String[] ss) : call(* foo(int,String...)) && args(i,ss) {
  // varargs String... argument is accessible in advice body through ss
  // ...
}
```

Since you cannot use the varargs syntax in the `args` pointcut designator, you also cannot use the varargs syntax to declare advice parameters.

Note: the proposal in this section does not allow you to distinguish between a join point with a signature `(int, String⋯)` and a join point with a signature `(int, String[])` based *solely* on the use of the `args` pointcut designator. If this distinction is required, `args` can always be coupled with `call` or `execution`.

Enumerated Types

Enumerated Types in Java 5

Java 5 (and hence AspectJ 5) provides explicit support for enumerated types. In the simplest case, you can declare an enumerated type as follows:

```
public enum ProgrammingLanguages {
  COBOL, C, JAVA, ASPECTJ
}
```

Enumerated types are just classes, and they can contain method and field declarations, and may implement interfaces. Enums may only have private constructors, and may not be extended.

Enumerated types in Java 5 all implicitly extend the type `java.lang.Enum`. It is illegal to explicitly declare a subtype of this class.

Enumerated Types in AspectJ 5

AspectJ 5 supports the declaration of enumerated types just as Java 5 does. Because of the special restrictions Java 5 places around enumerated types, AspectJ makes the following additional restrictions:

- You cannot use declare parents to change the super type of an enum.
- You cannot use declare parents to declare `java.lang.Enum` as the parent of any type.
- You cannot make inter-type constructor declarations on an enum.
- You cannot extend the set of values in an enum via any ITD-like construct.
- You cannot make inter-type method or field declarations on an enum.
- You cannot use declare parents to make an enum type implement an interface.

In theory, the last of these two items *could* be supported. However, AspectJ 5 follows the simple rule that *an enum type cannot be the target of an inter-type declaration or declare parents statement*. This position may be relaxed in a future version of AspectJ.

If an enum is named explicitly as the target of a declare parents statement, a compilation error will result. If an enumerated type is matched by a non-explicit type pattern used in a declare parents statement it will be ignored (and an XLint warning issued).

The `pertypewithin` Aspect Instantiation Model

AspectJ 5 defines a new per-clause type for aspect instantiation: `pertypewithin`. Unlike the other per-clauses, `pertypewithin` takes a type pattern:

```
PerTypeWithin := 'pertypewithin' '(' OptionalParensTypePattern ')'
```

When an aspect is declared using the `pertypewithin` instantiation model, one new aspect instance will be created for each type matched by the associated type pattern.

Pertypewithin aspects have `aspectOf` and `hasAspect` methods with the following signatures:

```
/**
 * return true if this aspect has an instance associated with
 * the given type.
 */
public static boolean hasAspect(Class clazz)

/**
 * return the instance associated with the given type.
 * Throws NoAspectBoundException if there is no such
 * aspect.
 */
public static P aspectOf(Class clazz)
```

Where `P` is the type of the `pertypewithin` aspect.

In addition, `pertypewithin` aspects have a `getWithinTypeName` method that can be called to return the package qualified name of the type for which the aspect instance has been created.

```
/**
 * return the package qualified name (eg. com.foo.MyClass) of the type
 * for which the aspect instance has been instantiated.
 */
public String getWithinTypeName()
```

In common with the other per-clause instantiation models, the execution of any advice declared within a `pertypewithin` aspect is conditional upon an implicit pointcut condition. In this case, that any join point be `within` the type that the executing aspect is an `aspectOf`. For example, given the aspect definition

```
import java.util.*;

public aspect InstanceTracking pertypewithin(org.xyz..*) {

  // use WeakHashMap for auto-garbage collection of keys
```

```
  private Map<Object,Boolean> instances = new WeakHashMap<Object,Boolean>();

  after(Object o) returning() : execution(new(..)) && this(o) {
    instances.put(o,true);
  }

  public Set<?> getInstances() {
    return instances.keySet();
  }

}
```

Then one aspect instance will be created for each type within `org.xyz...` For each aspect instance, the after returning advice will match only the execution of constructors within the matched per-type-within type. The net result is that the aspect tracks all known instances of each type within `org.xyz..`. To get access to the instances, a programmer can simply write `InstanceTracking.aspectOf(org.xyz.SomeType.class).getInstances()`.

The `pertypewithin` aspect instantiation model should be used when the implementation of a crosscutting concern requires that some state be maintained for each type in a set of types. To maintain state for a single type, it is easier to use a static inter-type declared field. Examples of usage include instance tracking, profiling, and the implementation of a common tracing idiom that uses one Logger per traced class.

An Annotation Based Development Style

Introduction

In addition to the familiar AspectJ code-based style of aspect declaration, AspectJ 5 also supports an annotation-based style of aspect declaration. We informally call the set of annotations that support this development style the "@AspectJ" annotations.

AspectJ 5 allows aspects and their members to be specified using either the code style or the annotation style. Whichever style you use, the AspectJ weaver ensures that your program has exactly the same semantics. It is, to quote a famous advertising campaign, "a choice, not a compromise". The two styles can be mixed within a single application, and even within a single source file, though we doubt this latter mix will be recommended in practice.

The use of the @AspectJ annotations means that there are large classes of AspectJ applications that can be compiled by a regular Java 5 compiler, and subsequently woven by the AspectJ weaver (for example, as an additional build stage, or as late as class load-time). In this chapter we introduce the @AspectJ annotations and show how they can be used to declare aspects and aspect members.

Aspect Declarations

Aspect declarations are supported by the `org.aspectj.lang.annotation.Aspect` annotation. The declaration:

```
@Aspect
public class Foo {}
```

Is equivalent to:

```
public aspect Foo {}
```

To specify an aspect an aspect instantiation model (the default is singleton), provide the perclause as the `@Aspect` value. For example:

```
@Aspect("perthis(execution(* abc..*(..)))")
public class Foo {}
```

is equivalent to...

```
public aspect Foo perthis(execution(* abc..*(..))) {}
```

Limitations

Privileged aspects are not supported by the annotation style.

Pointcuts and Advice

Pointcut and advice declarations can be made using the `Pointcut`, `Before`, `After`, `AfterReturning`, `AfterThrowing`, and `Around` annotations.

Pointcuts

Pointcuts are specified using the `org.aspectj.lang.annotation.Pointcut` annotation on a method declaration. The method should have a `void` return type. The parameters of the method correspond to the parameters of the pointcut. The modifiers of the method correspond to the modifiers of the pointcut.

As a general rule, the `@Pointcut` annotated method must have an empty method body and must not have any `throws` clause. If formal are bound (using `args()`, `target()`, `this()`, `@args()`, `@target()`, `@this()`, `@annotation()`) in the pointcut, then they must appear in the method signature.

The `if()` pointcut is treated specially and is discussed in a later section.

Here is a simple example of a pointcut declaration in both code and @AspectJ styles:

```
@Pointcut("call(* *.*(..))")
void anyCall() {}
```

is equivalent to...

```
pointcut anyCall() : call(* *.*(..));
```

When binding arguments, simply declare the arguments as normal in the annotated method:

```
@Pointcut("call(* *.*(int)) && args(i) && target(callee)")
void anyCall(int i, Foo callee) {}
```

is equivalent to...

```
pointcut anyCall(int i, Foo callee) : call(* *.*(int)) && args(i) && target(callee);
```

An example with modifiers (Remember that Java 5 annotations are not inherited, so the `@Pointcut` annotation must be present on the extending aspect's pointcut declaration too):

```
@Pointcut("")
```

```
protected abstract void anyCall();
```

is equivalent to...

```
protected abstract pointcut anyCall();
```

Type references inside @AspectJ annotations

Using the code style, types referenced in pointcut expressions are resolved with respect to the imported types in the compilation unit. When using the annotation style, types referenced in pointcut expressions are resolved in the absence of any imports and so have to be fully qualified if they are not by default visible to the declaring type (outside of the declaring package and `java.lang`). This does not apply to type patterns with wildcards, which are always resolved in a global scope.

Consider the following compilation unit:

```
package org.aspectprogrammer.examples;

import java.util.List;

public aspect Foo {
  pointcut listOperation() : call(* List.*(..));
  pointcut anyUtilityCall() : call(* java.util..*(..));
}
```

Using the annotation style this would be written as:

```
package org.aspectprogrammer.examples;

import java.util.List; // redundant but harmless

@Aspect
public class Foo {
  @Pointcut("call(* java.util.List.*(..))") // must qualify
  void listOperation() {}

  @Pointcut("call(* java.util..*(..))")
  void anyUtilityCall() {}
}
```

if() pointcut expressions

In code style, it is possible to use the `if(…)` poincut to define a conditional pointcut expression which will be evaluated at runtime for each candidate join point. The `if(…)` body can be any valid Java boolean expression, and can use any exposed formal, as well as the join point forms `thisJoinPoint, thisJoinPointStaticPart and thisJoinPointEnclosingStaticPart` .

When using the annotation style, it is not possible to write a full Java expression within the annotation value so the syntax differs slightly, whilst providing the very same semantics and runtime behaviour. An `if()` pointcut expression can be declared in an `@Pointcut` , but must have either an empty body (`if()`, or be one of the expression forms `if(true)` or `if(false)` . The annotated method must be public, static, and return a boolean. The body of the method contains the condition to be evaluated. For example:

```
@Pointcut("call(* *.*(int)) && args(i) && if()")
public static boolean someCallWithIfTest(int i) {
    return i > 0;
}
```

is equivalent to...

```
pointcut someCallWithIfTest(int i) :
  call(* *.*(int)) && args(i) && if(i > 0);
```

and the following is also a valid form:

```
static int COUNT = 0;

@Pointcut("call(* *.*(int)) && args(i) && if()")
public static boolean someCallWithIfTest(int i, JoinPoint jp, JoinPoint
.EnclosingStaticPart esjp) {
   // any legal Java expression...
   return i > 0
          && jp.getSignature().getName.startsWith("doo")
          && esjp.getSignature().getName().startsWith("test")
          && COUNT++ < 10;
}

@Before("someCallWithIfTest(anInt, jp, enc)")
public void beforeAdviceWithRuntimeTest(int anInt, JoinPoint jp, JoinPoint
.EnclosingStaticPart enc) {
   //...
}

// Note that the following is NOT valid
/*
@Before("call(* *.*(int)) && args(i) && if()")
public void advice(int i) {
   // so you were writing an advice or an if body ?
}
*/
```

It is thus possible with the annotation style to use the `if()` pointcut only within an `@Pointcut` expression. The `if()` must not contain any body. The annotated `@Pointcut` method must then be of

the form `public static boolean` and can use formal bindings as usual. Extra *implicit* arguments of type JoinPoint, JoinPoint.StaticPart and JoinPoint.EnclosingStaticPart can also be used (this is not permitted for regular annotated pointcuts not using the `if()` form).

The special forms `if(true)` and `if(false)` can be used in a more general way and don't imply that the pointcut method must have a body. You can thus write `@Before("somePoincut() && if(false)")` .

Advice

In this section we first discuss the use of annotations for simple advice declarations. Then we show how `thisJoinPoint` and its siblings are handled in the body of advice and discuss the treatment of `proceed` in around advice.

Using the annotation style, an advice declaration is written as a regular Java method with one of the `Before, After, AfterReturning, AfterThrowing,` or `Around` annotations. Except in the case of around advice, the method should return void. The method should be declared public.

A method that has an advice annotation is treated exactly as an advice declaration by AspectJ's weaver. This includes the join points that arise when the advice is executed (an adviceexecution join point, not a method execution join point).

The following example shows a simple before advice declaration in both styles:

```
@Before("call(* org.aspectprogrammer..*(..)) && this(Foo)")
public void callFromFoo() {
  System.out.println("Call from Foo");
}
```

is equivalent to...

```
before() : call(* org.aspectprogrammer..*(..)) && this(Foo) {
  System.out.println("Call from Foo");
}
```

If the advice body needs to know which particular `Foo` instance is making the call, just add a parameter to the advice declaration.

```
before(Foo foo) : call(* org.aspectprogrammer..*(..)) && this(foo) {
  System.out.println("Call from Foo: " + foo);
}
```

can be written as:

```
@Before("call(* org.aspectprogrammer..*(..)) && this(foo)")
public void callFromFoo(Foo foo) {
  System.out.println("Call from Foo: " + foo);
```

```
}
```

If the advice body needs access to `thisJoinPoint`, `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart` then these need to be declared as additional method parameters when using the annotation style.

```
@Before("call(* org.aspectprogrammer..*(..)) && this(foo)")
public void callFromFoo(JoinPoint thisJoinPoint, Foo foo) {
  System.out.println("Call from Foo: " + foo + " at " + thisJoinPoint);
}
```

is equivalent to...

```
before(Foo foo) : call(* org.aspectprogrammer..*(..)) && this(foo) {
  System.out.println("Call from Foo: " + foo + " at " + thisJoinPoint);
}
```

Advice that needs all three variables would be declared:

```
@Before("call(* org.aspectprogrammer..*(..)) && this(Foo)")
public void callFromFoo(
  JoinPoint thisJoinPoint,
  JoinPoint.StaticPart thisJoinPointStaticPart,
  JoinPoint.EnclosingStaticPart thisEnclosingJoinPointStaticPart
) {
    // ...
}
```

`JoinPoint.EnclosingStaticPart` is a new (empty) sub-interface of `JoinPoint.StaticPart` which allows the AspectJ weaver to distinguish based on type which of `thisJoinPointStaticPart` and `thisEnclosingJoinPointStaticPart` should be passed in a given parameter position.

`After` advice declarations take exactly the same form as `Before`, as do the forms of `AfterReturning` and `AfterThrowing` that do not expose the return type or thrown exception respectively.

To expose a return value with after returning advice simply declare the returning parameter as a parameter in the method body and bind it with the "returning" attribute:

```
@AfterReturning("criticalOperation()")
public void phew() {
  System.out.println("phew");
}

@AfterReturning(pointcut="call(Foo+.new(..))",returning="f")
public void itsAFoo(Foo f) {
  System.out.println("It's a Foo: " + f);
```

```
}
```

is equivalent to...

```
after() returning : criticalOperation() {
  System.out.println("phew");
}

after() returning(Foo f) : call(Foo+.new(..)) {
  System.out.println("It's a Foo: " + f);
}
```

(Note the use of the `pointcut=` prefix in front of the pointcut expression in the returning case).

After throwing advice works in a similar fashion, using the `throwing` attribute when needing to expose a thrown exception.

For around advice, we have to tackle the problem of `proceed` . One of the design goals for the annotation style is that a large class of AspectJ applications should be compilable with a standard Java 5 compiler. A straight call to `proceed` inside a method body:

```
@Around("call(* org.aspectprogrammer..*(..))")
public Object doNothing() {
  return proceed(); // CE on this line
}
```

will result in a "No such method" compilation error. For this reason AspectJ 5 defines a new sub-interface of `JoinPoint` , `ProceedingJoinPoint` .

```
public interface ProceedingJoinPoint extends JoinPoint {
  public Object proceed(Object[] args);
}
```

The around advice given above can now be written as:

```
@Around("call(* org.aspectprogrammer..*(..))")
public Object doNothing(ProceedingJoinPoint thisJoinPoint) {
  return thisJoinPoint.proceed();
}
```

Here's an example that uses parameters for the proceed call:

```
@Aspect
public class ProceedAspect {
```

```
  @Pointcut("call(* setAge(..)) && args(i)")
  void setAge(int i) {}

  @Around("setAge(i)")
  public Object twiceAsOld(ProceedingJoinPoint thisJoinPoint, int i) {
    return thisJoinPoint.proceed(new Object[]{i*2}); //using Java 5 autoboxing
  }

}
```

is equivalent to:

```
public aspect ProceedAspect {
  pointcut setAge(int i): call(* setAge(..)) && args(i);

  Object around(int i): setAge(i) {
    return proceed(i*2);
  }
}
```

Note that the ProceedingJoinPoint does not need to be passed to the `proceed(..)` arguments.

In code style, the proceed method has the same signature as the advice, any reordering of actual arguments to the joinpoint that is done in the advice signature must be respected. Annotation style is different. The `proceed(..)` call takes, in this order:

- If `this()` was used in the pointcut for binding, it must be passed first in `proceed(..)`.
- If `target()` was used in the pointcut for binding, it must be passed next in `proceed(..)` - it will be the first argument to `proceed(..)` if `this()` was not used for binding.
- Finally come all the arguments expected at the join point, in the order they are supplied at the join point. Effectively the advice signature is ignored - it doesn't matter if a subset of arguments were bound or the ordering was changed in the advice signature, the `proceed(..)` calls takes all of them in the right order for the join point.

Since `proceed(..)` in this case takes an `Object` array, AspectJ cannot do as much compile time checking as it can for code style. If the rules above aren't obeyed, then it will unfortunately manifest as a runtime error.

Inter-type Declarations

Inter-type declarations are challenging to support using an annotation style. For code style aspects compiled with the *ajc* compiler, the entire type system can be made aware of inter-type declarations (new supertypes, new methods, new fields) and the completeness and correctness of it can be guaranteed. Achieving this with an annotation style is hard because the source code may simply be compiled with javac where the type system cannot be influenced and what is compiled must be 'pure Java'.

AspectJ 1.5.0 introduced `@DeclareParents`, an attempt to offer something like that which is achievable with code style declare parents and the other intertype declarations (fields, methods, constructors). However, it has proved too challenging to get close to the expressiveness and capabilities of code style in this area and effectively `@DeclareParents` is offering just a mixin strategy. The definition of mixin I am using here is that when some `interface I` is mixed into some target type `T` then this means that all the methods from `I` are created in `T` and their implementations are simple forwarding methods that call a delegate which that provides an implementation of `I`.

The next section covers `@DeclareParents` but AspectJ 1.6.4 introduces `@DeclareMixin` - an improved approach to defining a mixin and the choice of a different name for the annotation will hopefully alleviate some of the confusion about why `@DeclareParents` just doesn't offer the same semantics as the code style variant. Offering `@DeclareMixin` also gives code style developers a new tool for a simple mixin whereas previously they would have avoided `@DeclareParents`, thinking what it could only do was already achievable with code style syntax.

The `defaultImpl` attribute of `@DeclareParents` may become deprecated if `@DeclareMixin` proves popular, leaving `@DeclareParents` purely as a way to introduce a marker interface.

@DeclareParents

Consider the following aspect:

```
public aspect MoodIndicator {

   public interface Moody {};

   private Mood Moody.mood = Mood.HAPPY;

   public Mood Moody.getMood() {
     return mood;
   }

   declare parents : org.xyz..* implements Moody;

   before(Moody m) : execution(* *.*(..)) && this(m) {
      System.out.println("I'm feeling " + m.getMood());
   }
}
```

This declares an interface `Moody` , and then makes two inter-type declarations on the interface - a field that is private to the aspect, and a method that returns the mood. Within the body of the inter-type declared method `getMoody` , the type of `this` is `Moody` (the target type of the inter-type declaration).

Using the annotation style this aspect can be written:

```
@Aspect
public class MoodIndicator {
```

```
   // this interface can be outside of the aspect
   public interface Moody {
     Mood getMood();
   };

   // this implementation can be outside of the aspect
   public static class MoodyImpl implements Moody {
      private Mood mood = Mood.HAPPY;

      public Mood getMood() {
        return mood;
      }
   }

   // the field type must be the introduced interface. It can't be a class.
   @DeclareParents(value="org.xzy..*",defaultImpl=MoodyImpl.class)
   private Moody implementedInterface;

   @Before("execution(* *.*(..)) && this(m)")
   void feelingMoody(Moody m) {
      System.out.println("I'm feeling " + m.getMood());
   }
}
```

This is very similar to the mixin mechanism supported by AspectWerkz. The effect of the `@DeclareParents` annotation is equivalent to a declare parents statement that all types matching the type pattern implement the given interface (in this case `Moody`). Each method declared in the interface is treated as an inter-type declaration. Note how this scheme operates within the constraints of Java type checking and ensures that `this` has access to the exact same set of members as in the code style example.

Note that it is illegal to use the `@DeclareParents` annotation on an aspect' field of a non-interface type. The interface type is the inter-type declaration contract that dictates which methods are declared on the target type.

```
// this type will be affected by the inter-type declaration as the type pattern
matches
package org.xyz;
public class MoodTest {

   public void test() {
       // see here the cast to the introduced interface (required)
       Mood mood = ((Moody)this).getMood();
       ...
   }
}
```

The `@DeclareParents` annotation can also be used without specifying a `defaultImpl` value (for

example, `@DeclareParents("org.xyz..*")`). This is equivalent to a `declare parents ⋯ implements` clause, and does *not* make any inter-type declarations for default implementation of the interface methods.

Consider the following aspect:

```
public aspect SerializableMarker {
   declare parents : org.xyz..* implements Serializable;
}
```

Using the annotation style this aspect can be written:

```
@Aspect
public class SerializableMarker {
   @DeclareParents("org.xyz..*")
   Serializable implementedInterface;
}
```

If the interface defines one or more operations, and these are not implemented by the target type, an error will be issued during weaving.

@DeclareMixin

Consider the following aspect:

```
public aspect MoodIndicator {

   public interface Moody {};

   private Mood Moody.mood = Mood.HAPPY;

   public Mood Moody.getMood() {
     return mood;
   }

   declare parents : org.xyz..* implements Moody;

   before(Moody m) : execution(* *.*(..)) && this(m) {
      System.out.println("I'm feeling " + m.getMood());
   }
}
```

This declares an interface `Moody`, and then makes two inter-type declarations on the interface - a field that is private to the aspect, and a method that returns the mood. Within the body of the inter-type declared method `getMoody`, the type of `this` is `Moody` (the target type of the inter-type declaration).

Using the annotation style, this aspect can be written:

```
@Aspect
public class MoodIndicator {

   // this interface can be outside of the aspect
   public interface Moody {
     Mood getMood();
   };

   // this implementation can be outside of the aspect
   public static class MoodyImpl implements Moody {
      private Mood mood = Mood.HAPPY;

      public Mood getMood() {
        return mood;
      }
   }

   // The DeclareMixin annotation is attached to a factory method that can return
instances of the delegate
   // which offers an implementation of the mixin interface.  The interface that is
mixed in is the
   // return type of the method.
   @DeclareMixin("org.xyz..*")
   public static Moody createMoodyImplementation() {
     return new MoodyImpl();
   }

   @Before("execution(* *.*(..)) && this(m)")
   void feelingMoody(Moody m) {
      System.out.println("I'm feeling " + m.getMood());
   }
}
```

Basically, the `@DeclareMixin` annotation is attached to a factory method. The factory method specifies the interface to mixin as its return type, and calling the method should create an instance of a delegate that implements the interface. This is the interface which will be delegated to from any target matching the specified type pattern.

Exploiting this syntax requires the user to obey the rules of pure Java. So references to any targeted type as if it were affected by the Mixin must be made through a cast, like this:

```
// this type will be affected by the inter-type declaration as the type pattern
matches
package org.xyz;
public class MoodTest {

   public void test() {
```

```
        // see here the cast to the introduced interface (required)
        Mood mood = ((Moody)this).getMood();
        ...
    }
}
```

Sometimes the delegate instance may want to perform differently depending upon the type/instance for which it is behaving as a delegate. To support this it is possible for the factory method to specify a parameter. If it does, then when the factory method is called the parameter will be the object instance for which a delegate should be created:

```
@Aspect
public class Foo {

  @DeclareMixin("org.xyz..*")
  public static SomeInterface createDelegate(Object instance) {
    return new SomeImplementation(instance);
  }
}
```

It is also possible to make the factory method non-static - and in this case it can then exploit the local state in the surrounding aspect instance, but this is only supported for singleton aspects:

```
@Aspect
public class Foo {
  public int maxLimit=35;

  @DeclareMixin("org.xyz..*")
  public SomeInterface createDelegate(Object instance) {
    return new SomeImplementation(instance,maxLimit);
  }
}
```

Although the interface type is usually determined purely from the return type of the factory method, it can be specified in the annotation if necessary. In this example the return type of the method extends multiple other interfaces and only a couple of them (`I` and `J`) should be mixed into any matching targets:

```
// interfaces is an array of interface classes that should be mixed in
@DeclareMixin(value="org.xyz..*",interfaces={I.class,J.class})
public static InterfaceExtendingLotsOfInterfaces createMoodyImplementation() {
  return new MoodyImpl();
}
```

There are clearly similarities between `@DeclareMixin` and `@DeclareParents` but `@DeclareMixin` is not pretending to offer more than a simple mixin strategy. The flexibility in being able to provide the

factory method instead of requiring a no-arg constructor for the implementation also enables delegate instances to make decisions based upon the type for which they are the delegate.

Any annotations defined on the interface methods are also put upon the delegate forwarding methods created in the matched target type.

Declare statements

The previous section on inter-type declarations covered the case of `declare parents ⋯ implements`. The 1.5.0 release of AspectJ 5 does not support annotation style declarations for `declare parents ⋯ extends` and `declare soft` (programs with these declarations would not in general be compilable by a regular Java 5 compiler, reducing the priority of their implementation). These may be supported in a future release.

Declare annotation is also not supported in the 1.5.0 release of AspectJ 5.

Declare precedence *is* supported. For declare precedence, use the `@DeclarePrecedence` annotation as in the following example:

```
public aspect SystemArchitecture {
  declare precedence : Security*, TransactionSupport, Persistence;
  // ...
}
```

can be written as:

```
@Aspect
@DeclarePrecedence("Security*,org.xyz.TransactionSupport,org.xyz.Persistence")
public class SystemArchitecture {
  // ...
}
```

We also support annotation style declarations for declare warning and declare error - any corresponding warnings and errors will be emitted at weave time, not when the aspects containing the declarations are compiled. (This is the same behaviour as when using declare warning or error with the code style). Declare warning and error declarations are made by annotating a string constant whose value is the message to be issued.

Note that the String must be a literal and not the result of the invocation of a static method for example.

```
declare warning : call(* javax.sql..*(..)) && !within(org.xyz.daos..*)
                : "Only DAOs should be calling JDBC.";

declare error : execution(* IFoo+.*(..)) && !within(org.foo..*)
              : "Only foo types can implement IFoo";
```

can be written as...

```
@DeclareWarning("call(* javax.sql..*(..)) && !within(org.xyz.daos..*)")
static final String aMessage = "Only DAOs should be calling JDBC.";

@DeclareError("execution(* IFoo+.*(..)) && !within(org.foo..*)")
static final String badIFooImplementors = "Only foo types can implement IFoo";

// the following is not valid since the message is not a String literal
@DeclareError("execution(* IFoo+.*(..)) && !within(org.foo..*)")
static final String badIFooImplementorsCorrupted = getMessage();
static String getMessage() {
  return "Only foo types can implement IFoo " + System.currentTimeMillis();
}
```

`aspectOf()` and `hasAspect()` methods

A central part of AspectJ's programming model is that aspects written using the code style and compiled using ajc support `aspectOf` and `hasAspect` static methods. When developing an aspect using the annotation style and compiling using a regular Java 5 compiler, these methods will not be visible to the compiler and will result in a compilation error if another part of the program tries to call them.

To provide equivalent support for AspectJ applications compiled with a standard Java 5 compiler, AspectJ 5 defines the `Aspects` utility class:

```
public class Aspects {

  /* variation used for singleton, percflow, percflowbelow */
  static<T> public static T aspectOf(T aspectType) {...}

  /* variation used for perthis, pertarget */
  static<T> public static T aspectOf(T aspectType, Object forObject) {...}

  /* variation used for pertypewithin */
  static<T> public static T aspectOf(T aspectType, Class forType) {...}

  /* variation used for singleton, percflow, percflowbelow */
  public static boolean hasAspect(Object anAspect) {...}

  /* variation used for perthis, pertarget */
  public static boolean hasAspect(Object anAspect, Object forObject) {...}

  /* variation used for pertypewithin */
  public static boolean hasAspect(Object anAspect, Class forType) {...}
}
```

New Reflection Interfaces

AspectJ 5 provides a full set of reflection APIs analogous to the `java.lang.reflect` package, but fully aware of the AspectJ type system. See the javadoc for the runtime and tools APIs for the full details. The reflection APIs are only supported when running under Java 5 and for code compiled by the AspectJ 5 compiler at target level 1.5.

Using `AjTypeSystem`

The starting point for using the reflection apis is `org.aspectj.lang.reflect.AjTypeSystem` which provides the method `getAjType(Class)` which will return the `AjType` corresponding to a given Java class. The `AjType` interface corresponds to `java.lang.Class` and gives you access to all of the method, field, constructor, and also pointcut, advice, declare statement and inter-type declaration members in the type.

Other Changes in AspectJ 5

Pointcuts

AspectJ 5 is more liberal than AspectJ 1.2.1 in accepting pointcut expressions that bind context variables in more than one location. For example, AspectJ 1.2.1 does not allow:

```
pointcut foo(Foo foo) :
  (execution(* *(..)) && this(foo) ) ||
  (set(* *) && target(foo));
```

whereas this expression is permitted in AspectJ 5. Each context variable must be bound exactly once in each branch of a disjunction, and the disjunctive branches must be mutually exclusive. In the above example for instance, no join point can be both an execution join point and a set join point so the two branches are mutually exclusive.

Declare Soft

The semantics of the `declare soft` statement have been refined in AspectJ 5 to only soften exceptions that are not already runtime exceptions. If the exception type specified in a declare soft statement is `RuntimeException` or a subtype of `RuntimeException` then a new XLint warning will be issued:

```
declare soft : SomeRuntimeException : execution(* *(..));

// "SomeRuntimeException will not be softened as it is already a
// RuntimeException" [XLint:runtimeExceptionNotSoftened]
```

This XLint message can be controlled by setting the `runtimeExceptionNotSoftened` XLint parameter.

If the exception type specified in a declare soft statement is a super type of `RuntimeException` (such as `Exception` for example) then any *checked* exception thrown at a matched join point, where the exception is an instance of the softened exception, will be softened to an `org.aspectj.lang.SoftException`.

```
public aspect SoftenExample {
  declare soft : Exception : execution(* Foo.*(..));
}

class Foo {
  public static void main(String[] args) {
    Foo foo = new Foo();
    foo.foo();
    foo.bar();
  }
```

```
  void foo() throws Exception {
    throw new Exception();        // this will be converted to a SoftException
  }

  void bar() throws Exception {
    throw new RuntimeException();  // this will remain a RuntimeException
  }
}
```

Load-Time Weaving

Introduction

See Developer's Guide for information on load-time weaving support in AspectJ 5.

A Grammar for the AspectJ 5 Language

```
== type patterns ===

TypePattern :=
    SimpleTypePattern |
    '!' TypePattern |
    '(' AnnotationPattern? TypePattern ')'
    TypePattern '&&' TypePattern |
    TypePattern '||' TypePattern

SimpleTypePattern := DottedNamePattern '+'? '[]'*

DottedNamePattern :=
    FullyQualifiedName RestOfNamePattern? |
    '*' NotStarNamePattern?

RestOfNamePattern := '..' DottedNamePattern | '*' NotStarNamePattern?

NotStarNamePattern :=
    FullyQualifiedName RestOfNamePattern? |
    '..' DottedNamePattern

FullyQualifiedName := JavaIdentifierCharacter+ ('.' JavaIdentifierCharacter+)*

== annotation patterns ===

AnnotationPattern := '!'? '@' AnnotationTypePattern AnnotationPattern*

AnnotationTypePattern := FullyQualifiedName | '(' TypePattern ')'

== signature patterns ===

-- field --

FieldPattern :=
    AnnotationPattern? FieldModifiersPattern?
    TypePattern (TypePattern DotOrDotDot)? SimpleNamePattern

FieldModifiersPattern := '!'? FieldModifier FieldModifiersPattern*

FieldModifier :=
    'public' | 'private' | 'protected' | 'static' |
    'transient' | 'final'

DotOrDotDot := '.' | '..'

SimpleNamePattern := JavaIdentifierChar+ ('*' SimpleNamePattern)?
```

```
-- method --

MethodPattern :=
    AnnotationPattern? MethodModifiersPattern? TypePattern
    (TypePattern DotOrDotDot)? SimpleNamePattern
    '(' FormalsPattern ')' ThrowsPattern?

MethodModifiersPattern := '!'? MethodModifier MethodModifiersPattern*

MethodModifier :=
    'public' | 'private' | 'protected' | 'static' |
    'synchronized' | 'final'

FormalsPattern :=
    '..' (',' FormalsPatternAfterDotDot)? |
    OptionalParensTypePattern (',' FormalsPattern)* |
    TypePattern '...'

FormalsPatternAfterDotDot :=
    OptionalParensTypePattern (',' FormalsPatternAfterDotDot)* |
    TypePattern '...'

ThrowsPattern := 'throws' TypePatternList

TypePatternList := TypePattern (',' TypePattern)*

-- constructor --

ConstructorPattern :=
    AnnotationPattern? ConstructorModifiersPattern?
    (TypePattern DotOrDotDot)? 'new' '(' FormalsPattern ')'
    ThrowsPattern?

ConstructorModifiersPattern :=
    '!'? ConstructorModifier ConstructorModifiersPattern*

ConstructorModifier := 'public' | 'private' | 'protected'

== Pointcuts ===

PointcutPrimitive :=
    Call | Execution | Get | Set | Handler |
    Initialization | PreInitialization |
    StaticInitialization | AdviceExecution |
    This | Target | Args | CFlow | CFlowBelow |
    Within | WithinCode | If |
    AnnotationPointcut

AnnotationPointcut :=
    AtAnnotation | AtThis | AtTarget |
    AtWithin | AtWithinCode | AtArgs
```

```
Call := 'call' '(' MethodOrConstructorPattern ')'

MethodOrConstructorPattern := MethodPattern | ConstructorPattern

Execution := 'execution' '(' MethodOrConstructorPattern ')'

Get := 'get' '(' FieldPattern ')'
Set := 'set' '(' FieldPattern ')'
Handler := 'handler' '(' OptionalParensTypePattern ')'
Initialization := 'initialization' '(' ConstructorPattern ')'
PreInitialization := 'preinitialization' '(' ConstructorPattern ')'
StaticInitialization := 'staticinitialization' '(' OptionalParensTypePattern ')'
AdviceExecution := 'adviceexecution' '(' ')'
This := 'this' '(' TypeOrIdentifier ')'
Target := 'target' '(' TypeOrIdentifier ')'
Args := 'args' '(' FormalsOrIdentifiersPattern ')'
CFlow := 'cflow' '(' Pointcut ')'
CFlowBelow := 'cflowbelow' '(' Pointcut ')'
Within := 'within' '(' OptionalParensTypePattern ')'
WithinCode := 'withincode' '(' OptionalParensTypePattern ')'
If := 'if' '(' BooleanJavaExpression ')'

TypeOrIdentifier := FullyQualifiedName ('[' ']')* | Identifier
Identifier := JavaIdentifierChar+

FormalsOrIdentifiersPattern :=
    '..' (',' FormalsOrIdentifiersPatternAfterDotDot)? |
    TypeOrIdentifier (',' FormalsOrIdentifiersPattern)* |
    '*' (',' FormalsOrIdentifiersPattern)*

FormalsOrIdentifiersPatternAfterDotDot :=
    TypeOrIdentifier (',' FormalsOrIdentifiersPatternAfterDotDot)* |
    '*' (',' FormalsOrIdentifiersPatternAfterDotDot)*

AtAnnotation := '@annotation' '(' AnnotationOrIdentifier ')'
AtThis := '@this' '(' AnnotationOrIdentifer ')'
AtTarget := '@target' '(' AnnotationOrIdentifier ')'
AtWithin := '@within' '(' AnnotationOrIdentifier ')'
AtWithinCode := '@withincode' '(' AnnotationOrIdentifier ')'

AnnotationOrIdentifier := FullyQualifiedName | Identifier

AtArgs := '@args' '(' AnnotationsOrIdentifiersPattern ')'

AnnotationsOrIdentifiersPattern :=
    '..' (',' AnnotationsOrIdentifiersPatternAfterDotDot)? |
    AnnotationOrIdentifier (',' AnnotationsOrIdentifiersPattern)* |
    '*' (',' AnnotationsOrIdentifiersPattern)*

AnnotationsOrIdentifiersPatternAfterDotDot :=
```

```
    AnnotationOrIdentifier (',' AnnotationsOrIdentifiersPatternAfterDotDot)* |
    '*' (',' AnnotationsOrIdentifiersPatternAfterDotDot)*

PointcutDeclaration :=
    PointcutModifiers? 'pointcut' Identifier Formals ':' PointcutExpression

PointcutModifiers := PointcutModifier*

PointcutModifier := 'public' | 'private' | 'protected' | 'abstract'

Formals := '(' ParamList? ')'
ParamList := FullyQualifiedName Identifier (',' ParamList)*

ReferencePointcut := (FullyQualifiedName '.')? Identifier Formals

PointcutExpression :=
    (PointcutPrimitive | ReferencePointcut) |
    '!' PointcutExpression |
    '(' PointcutExpression ')' |
    PointcutExpression '&&' PointcutExpression |
    PointcutExpression '||' PointcutExpression

== Advice ===

to be written...

== Inter-type Declarations ===

to be written...

== Declare Statements ===

to be written...

== Aspects ===

to be written...
```