

@AspectJ Annotations

Raffi Khatchadourian

Contents

- Annotations in Java 1.5
- `@AspectJ`
 - In Declarations
 - In Pointcuts

Java 1.5 Annotations

- Annotations are meta-data tags added to code.
- Can apply to package declarations, type declarations, methods, fields, variables, etc.
- Relieves developers of cumbersome, external configuration files.
- Annotation Processing Tool (APT) API
 - Can *reflect* on annotations.

- Annotation type declarations are similar to normal interface declarations.
- An at-sign (@) precedes the interface keyword.

- Each method declaration defines an element of the annotation type.
- No params, no throw clause.
- Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types.
- Methods can have default values.

```
/**
 * Describes the Request-For-Enhancement(RFE) that led
 * to the presence of the annotated API element.
 */
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();      default "[unimplemented]";
}
```

- An annotation is a special kind of modifier.
- can be used anywhere that other modifiers (such as public, static, or final) can be used.

```
@RequestForEnhancement(  
    id          = 2868724,  
    synopsis   = "Enable time-travel",  
    engineer    = "Mr. Peabody",  
    date       = "4/1/3007"  
)  
public static void travelThroughTime(Date destination)  
{ ... }
```

- An annotation type with no elements is termed a marker annotation type

```
/**  
 * Indicates that the specification of the annotated API element  
 * is preliminary and subject to change.  
 */  
public @interface Preliminary { }
```

```
@Preliminary public class TimeTravel { ... }
```

```
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * This annotation should be used only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }
```

- Note that the annotation type declaration is itself annotated!
- *meta-annotations*
- `@Retention(RetentionPolicy.RUNTIME)` indicates that annotations with this type are to be retained by the VM so they can be read reflectively at run-time.
- `@Target(ElementType.METHOD)` indicates that this annotation type can be used to annotate only method declarations.

```
public class Foo {
    @Test public static void m1() { }

    public static void m2() { }

    @Test public static void m3() {
        throw new RuntimeException("Boom");}

    public static void m4() { }

    @Test public static void m5() { }

    public static void m6() { }

    @Test public static void m7() {
        throw new RuntimeException("Crash");}

    public static void m8() { }
}
```

```
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int passed = 0, failed = 0;
        for (Method m : Class.forName(args[0]).getMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                try {
                    m.invoke(null); passed++;
                } catch (Throwable ex) {
                    System.out.printf
                        ("Test %s failed: %s %n", m, ex.getCause());
                    failed++;
                }
            }
        }
        System.out.printf
            ("Passed: %d, Failed %d%n", passed, failed);
    }
}
```

```
$ java RunTests Foo
Test public static void Foo.m3() failed:
java.lang.RuntimeException: Boom

Test public static void Foo.m7() failed:
java.lang.RuntimeException: Crash

Passed: 2, Failed 2
```

Annotations in AspectJ

- `@AspectJ` introduced in AspectJ 5
- Allows:
 - aspects and their members to be specified using either the code style or the annotation style.
 - PCD's for annotations.
 - Dynamic advice reflecting on annotation values.

@AspectJ Declarations

- Enables compilation of AJ source by regular Java 1.5 compiler.
 - Subsequently woven by the AspectJ weaver
 - additional build stage.
 - class load-time.
- True *plug-n-play* of aspect subsystem.
- Privileged aspects are not supported.

Declaring Pointcuts

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

Is equivalent to:

```
public aspect Foo {}
```

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 someCall(int i, Foo callee) {}
```

is equivalent to...

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

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() {}

}
```

```
@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);
```

Declaring 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 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);  
}
```

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);
    }
}
```

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.

```
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 pertyewithin */
    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 pertyewithin */
    public static boolean hasAspect(Object anAspect, Class forType) {...}
}
```