Chapter 13

Annotations

NOTE: This article is an excerpt from Cay Horstmann and Gary Cornell, *Core Java*, 7th ed., vol. 2, ch. 13.

According to Graham Hamilton, Sun fellow and lead architect for J2SE 5.0, annotations are the single most important feature of the new Java release. Annotations are tags that you insert into your source code so that they can be processed by tools.

The Java compiler understands a couple of annotations, but to go any further, you need to build your own processing tool or obtain a tool from a third party.

There is a wide range of possible uses for annotations, and that generality can be initially confusing. Here are some possible uses for annotations:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes.
- Automatic generation of code for testing, logging, transaction semantics, and so on.

EJB 3.0 users will see annotations in a big way. Quite a bit of the repetitive coding in EJB will be automated by annotations.

In this article, we start out with the basic concepts and put them to use in a concrete example: we mark methods as event listeners for AWT components and show you an annotation processor that analyzes the annotations and hooks up the listeners. We then briefly look at advanced annotation processing.

Adding Metadata to Programs

Metadata are data about data. In the context of computer programs, metadata are data about the code. A good example for metadata is Javadoc comments. The comments describe the code, but they do not modify its meaning.

Starting with JDK 5.0, you can use *annotations* to insert arbitrary data into your source code. Here is an example of a simple annotation:

```
public class MyClass
{
   . . .
   @TestCase public void checkRandomInsertions()
}
```

The annotation `@TestCase` annotates the `checkRandomInsertions` method.

In the Java programming language, an annotation is used like a *modifier*, and it is placed before the annotated item, *without a semicolon*. (A modifier is a keyword such as `public` or `static`.) The name of each annotation is preceded by an `@` symbol, similar to Javadoc comments. However, Javadoc comments occur inside `/** . . . */` delimiters, whereas annotations are part of the code.

By itself, the `@TestCase` annotation does not do anything. It needs a tool to be useful. For example, a testing tool might call all methods that are labeled as `@TestCase` when testing a class. Another tool might remove all test methods from a class file so that they are not shipped with the program after it has been tested.

Annotations can be defined to have *elements*, such as

```
@TestCase(id="3352627")
```

These elements can be processed by the tools that read the annotations. Other forms of elements are possible; we discuss them later in this chapter.

Besides methods, you can annotate classes, fields, and local variables—an annotation can be anywhere you could put a modifier such as `public` or `static`.

Each annotation must be defined by an *annotation interface*. The methods of the interface correspond to the elements of the annotation. For example, a `TestCase` annotation could be defined by the following interface:

```
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestCase
{
   String id() default "[none]";
}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. A tool would call the `id` method to retrieve the `id` element of a particular `TestCase` annotation.

The `Target` and `Retention` annotations are *meta-annotations*. They annotate the `TestCase` annotation, marking it as an annotation that can be applied to methods only and that is retained when the class file is loaded into the virtual machine.

You have now seen the basic concepts of program metadata and annotations. In the next section, we walk through a concrete example of annotation processing.

An Example: Annotating Event Handlers

One of the more boring tasks in user interface programming is the wiring of listeners to event sources. Many listeners are of the form

```
myButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         doSomething();
      }
   });
```

In this section, we design an annotation to avoid this drudgery. The annotation has the form

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

The programmer no longer has to make calls to `addActionListener`. Instead, each method is simply tagged with an annotation. Example 13-1 shows a simple program that makes use of these annotations.

We also need to define an annotation interface. The code is in Example 13-2.

Example 13-1: ButtonTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. public class ButtonTest
6. {
7.    public static void main(String[] args)
8.    {
9.       ButtonFrame frame = new ButtonFrame();
10.      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11.      frame.setVisible(true);
12.   }
13. }
14.
15. /**
16.    A frame with a button panel
17. */
18. class ButtonFrame extends JFrame
19. {
20.   public ButtonFrame()
21.   {
22.      setTitle("ButtonTest");
23.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24.
25.      panel = new JPanel();
26.      add(panel);
27.
28.      // create buttons
29.
30.      yellowButton = new JButton("Yellow");
31.      blueButton = new JButton("Blue");
32.      redButton = new JButton("Red");
33.
```

```
34.       // add buttons to panel
35.
36.       panel.add(yellowButton);
37.       panel.add(blueButton);
38.       panel.add(redButton);
39.
40.       ActionListenerInstaller.processAnnotations(this);
41.    }
42.
43.
44.    @ActionListenerFor(source="yellowButton")
45.    public void yellowBackground()
46.    {
47.       panel.setBackground(Color.YELLOW);
48.    }
49.
50.    @ActionListenerFor(source="blueButton")
51.    public void blueBackground()
52.    {
53.       panel.setBackground(Color.BLUE);
54.    }
55.
56.    @ActionListenerFor(source="redButton")
57.    public void redBackground()
58.    {
59.       panel.setBackground(Color.RED);
60.    }
61.
62.    public static final int DEFAULT_WIDTH = 300;
63.    public static final int DEFAULT_HEIGHT = 200;
64.
65.    private JPanel panel;
66.    private JButton yellowButton;
67.    private JButton blueButton;
68.    private JButton redButton;
69. }
```

Example 13-2: ActionListenerFor.java

```
1. import java.lang.annotation.*;
2.
3. @Target(ElementType.METHOD)
4. @Retention(RetentionPolicy.RUNTIME)
5. public @interface ActionListenerFor
6. {
7.    String source();
8. }
```

Of course, the annotations don't do anything by themselves. They sit in the source file. The compiler places them in the class file, and the virtual machine loads them. We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

The static `processAnnotations` method enumerates all methods of the object that it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

```
Class cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
   ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
   if (a != null) . . .
}
```

Here, we use the `getAnnotation` method that is defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface.

The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

For each annotated method, we construct a proxy object that implements the `ActionListener` interface and whose `actionPerformed` method calls the annotated method. The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method.

In Example 13-3, the annotations were processed at run time. It would also have been possible to process them at the source level. A source code generator might have produced the code for adding the listeners. Alternatively, the annotations might have been processed at the bytecode level. A bytecode editor might have injected the calls to `addActionListener` into the frame constructor.

Example 13-3: ActionListenerInstaller.java

```
1. import java.awt.event.*;
2. import java.lang.annotation.*;
3. import java.lang.reflect.*;
4.
5. public class ActionListenerInstaller
6. {
7.    /**
8.       Processes all ActionListenerFor annotations in the given object.
9.       @param obj an object whose methods may have ActionListenerFor annotations
10.   */
11.   public static void processAnnotations(Object obj)
12.   {
13.      try
14.      {
15.         Class cl = obj.getClass();
16.         for (Method m : cl.getDeclaredMethods())
17.         {
18.            ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
19.            if (a != null)
20.            {
21.               Field f = cl.getDeclaredField(a.source());
22.               f.setAccessible(true);
23.               addListener(f.get(obj), obj, m);
24.            }
25.         }
26.      }
27.      catch (Exception e)
28.      {
29.         e.printStackTrace();
```

```
30.       }
31.    }
32.
33.    /**
34.       Adds an action listener that calls a given method.
35.       @param source the event source to which an action listener is added
36.       @param param the implicit parameter of the method that the listener calls
37.       @param m the method that the listener calls
38.    */
39.    public static void addListener(Object source, final Object param, final Method m)
40.       throws NoSuchMethodException, IllegalAccessException, InvocationTargetException
41.    {
42.       InvocationHandler handler = new
43.          InvocationHandler()
44.          {
45.             public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
46.             {
47.                return m.invoke(param);
48.             }
49.          };
50.
51.       Object listener = Proxy.newProxyInstance(null,
52.          new Class[] { java.awt.event.ActionListener.class },
53.          handler);
54.       Method adder = source.getClass().getMethod("addActionListener", ActionListener.class);
55.       adder.invoke(source, listener);
56.    }
57. }
```

Annotation Syntax

In this section, we cover everything you need to know about the annotation syntax.

An annotation is defined by an annotation interface:

modifiers `@interface` *AnnotationName*
`{`
element declaration$_1$
element declaration$_2$
`. . .`
`}`

Each element declaration has the form

type elementName`();`

or

type elementName`()` `default` *value*`;`

For example, the following annotation has two elements, `assignedTo` and `severity`.

```
public @interface BugReport
{
   String assignedTo() default "[none]";
   int severity() = 0;
}
```

Each annotation has the format

`@`*AnnotationName*`(`*elementName*$_1$`=`*value*$_1$`,` *elementName*$_2$`=`*value*$_2$`, . . .)`

For example,

```
@BugReport(assignedTo="Harry", severity=10)
```

The order of the elements does not matter. The annotation

```
@BugReport(severity=10, assignedTo="Harry")
```

is identical to the preceding one.

The default value of the declaration is used if an element value is not specified. For example, consider the annotation

```
@BugReport(severity=10)
```

The value of the `assignedTo` element is the string `"[none]"`.

Two special shortcuts can simplify annotations.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, then you don't need to use parentheses. For example,

```
@BugReport
```

is the same as

```
@BugReport(assignedTo="[none]", severity=0)
```

Such an annotation is called a *marker annotation*.

The other shortcut is the *single value annotation*. If an element has the special name `value` and no other element is specified, then you can omit the element name and the `=` symbol. For example, had we defined the `ActionListenerFor` annotation interface of the preceding section as

```
public @interface ActionListenerFor
{
   String value();
}
```

then we could have written the annotations as

```
@ActionListenerFor("yellowButton")
```

instead of

```
@ActionListenerFor(value="yellowButton")
```

The type of an annotation element is one of the following:

- A primitive type (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, or `boolean`)
- `String`
- `Class` (or a parameterized type such as `Class<? extends MyClass>`)
- An `enum` type
- An annotation type
- An array of the preceding types

Here are examples for valid element declarations:

```
public @interface BugReport
{
   enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
   boolean showStopper() default false;
   String assignedTo() default "[none]";
   Class<? extends Testable> testCase() default Testable.class;
   Status status() default Status.UNCONFIRMED;
   TestCase testCase();
   String[] reportedBy();
}
```

CAUTION: An annotation element can never be set to `null`. This can be rather inconvenient in practice. You will need to find other defaults, such as "" or `Void.class`.

If an element value is an array, you enclose its values in braces, like this:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

Since an annotation element can be another annotation, you can build arbitrarily complex annotations. For example,

```
@BugReport(testCase=@TestCase(id="3352627"), . . .)
```

You can add annotations to the following items:

- Packages
- Classes (including `enum`)
- Interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance fields (including `enum` constants)
- Local variables
- Parameter variables

Standard Annotations

JDK 5.0 defines seven annotation interfaces. Three of them are regular annotations that you can use to annotate items in your source code. The other four are meta-annotations that describe the behavior of annotation interfaces. Table 13–1 shows these annotations. We discuss them in detail in the following two sections.

Table 13–1: The Standard Annotations

Annotation Interface	Applicable To	Purpose
`Deprecated`	All	Marks item as deprecated
`SuppressWarnings`	All but packages and annotations	Suppresses warnings of the given type
`Override`	Methods	Checks that this method overrides a superclass method
`Target`	Annotations	Specifies the items to which this annotation can be applied
`Retention`	Annotations	Specifies how long this annotation is retained
`Documented`	Annotations	Specifies that this annotation should be included in the documentation of annotated items
`Inherited`	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses.

Regular Annotations

The `@Deprecated` annotation can be attached to any items whose use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` Javadoc tag.

The `@SuppressWarnings` annotation tells the compiler to suppress warnings of a particular type, for example,

```
@SuppressWarnings("unchecked cast")
```

The initial release of the JDK 5.0 compiler does not support this annotation.

The `@Override` annotation applies only to methods. The compiler checks that a method with this annotation really overrides a method from the superclass. For example, if you declare

```
public MyClass
{
   @Override public boolean equals(MyClass other);
   . . .
}
```

then the compiler will report an error. After all, the `equals` method does *not* override the `equals` method of the `Object` class. That method has a parameter of type `Object`, not `MyClass`.

Meta-Annotations

The `@Target` meta-annotation is applied to an annotation, restricting the items to which the annotation applies. For example,

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface TestCase
```

Table 13–2 shows all possible values. They belong to the enumerated type `ElementType`. You can specify any number of element types, enclosed in braces.

Table 13–2: Element Types for the `@Target` Annotation

Element Type	Annotation Applies To
`ANNOTATION_TYPE`	Annotation type declarations
`PACKAGE`	Packages
`TYPE`	Classes (including `enum`) and interfaces (including annotation types)
`METHOD`	Methods
`CONSTRUCTOR`	Constructors
`FIELD`	Fields (including `enum` constants)
`PARAMETER`	Method or constructor parameters
`LOCAL_VARIABLE`	Local variables

An annotation without an `@Target` restriction can be applied to any item. The compiler checks that you apply an annotation only to a permitted item. For example, if you apply `@TestCase` to a field, a compile-time error results.

The `@Retention` meta-annotation specifies how long an annotation is retained. You specify at most one of the values in Table 13–3. The default is `RetentionPolicy.CLASS`.

In Example 13-2, the `@ActionListenerFor` annotation was declared with `RetentionPolicy.RUNTIME` because we used reflection to process annotations. In the following two sections, you will see examples of processing annotations at the source and class file levels.

Table 13–3: Retention Policies for the `@Retention` Annotation

Retention Policy	Description
`SOURCE`	Annotations are not included in class files.
`CLASS`	Annotations are included in class files, but the virtual machine need not load them.
`RUNTIME`	Annotations are included in class files and loaded by the virtual machine. They are available through the reflection API.

The `@Documented` meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers, such as `protected` or `static`, for documentation purposes. The use of other annotations is not included in the documentation. For example, suppose we declare `@ActionListenerFor` as a documented annotation:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

Now the documentation of each annotated method contains the annotation, as shown in Figure 13–1.

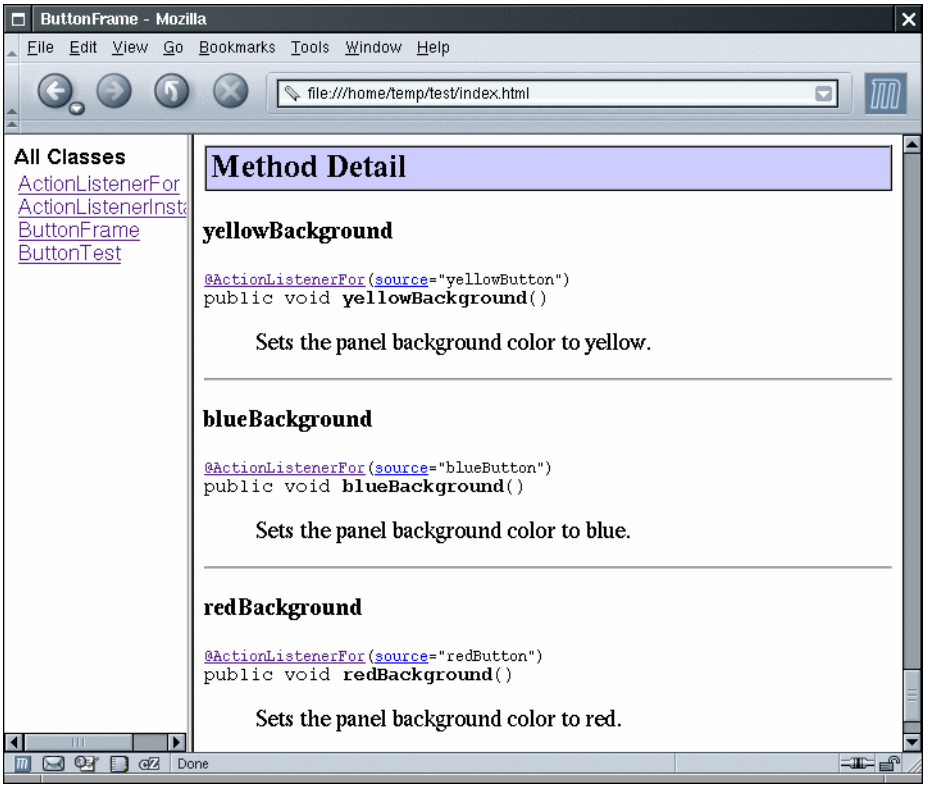

Figure 13–1: Documented annotations

NOTE: It is legal to apply an annotation to itself. For example, the `@Documented` annotation is itself annotated as `@Documented`. Therefore, the Javadoc documentation for annotations shows whether they are documented.

The `@Inherited` meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. Suppose you define an inherited annotation, `@Persistent`, to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // also @Persistent
```

When the persistence mechanism searches for objects to store in the database, it will detect both `Employee` and `Manager` objects.

Advanced Annotation Processing

Annotations can be processed at three levels:

- In the source code
- In bytecode files
- At run time, in the virtual machine

Our introductory examples processed annotations at run time. To write tools for source code and bytecode processing, you need to work harder since you need to parse the source code or bytecodes.

You can find detailed examples in the full Core Java chapter. Here is a brief synopsis.

You process annotations at the source level if you want to produce other source files, such as bean information classes, remote interfaces, XML descriptors, or other "side files." Since you probably don't want to write your own Java parser, the JDK contains an annotation processing tool called `apt`. You can find documentation for `apt` at `http://java.sun.com/j2se/5.0/docs/guide/apt/`. The `apt` tool processes annotations in source files, feeding them to *annotation processors* that you supply.

Your annotation processors can take arbitrary actions. The most common action will be to write a file that contains Java source or XML descriptors.

In the Core Java chapter, we show you how to automatically produce a `BeanInfo` file from the metadata in a JavaBeans class.

NOTE: Some people have suggested using annotations to remove a major drudgery of programming in Java. Wouldn't it be nice if trivial getters and setters were generated automatically? For example, the annotation

```
@Property private String title;
```

could produce the methods

```
public String getTitle() {return title;}
public void setTitle(String title) { this.title = title; }
```

However, those methods need to be added to the *same class*. This requires editing a source file, not just generating another file, and is beyond the capabilities of `apt`. It would be possible to build another tool for this purpose, but such a tool would probably go beyond the mission of annotations. An annotation is intended as a description *about* a code item, not a directive for code editing.

The annotation facility gives programmers a lot of power, and with that power comes the potential for abuse. If annotations are used without restraint, they can result in code that becomes incomprehensible to programmers and development tools alike. (C and C++ programmers have had the same experience with unrestrained use of the C preprocessor.)

We offer the following rule of thumb for the responsible use of annotations: Your source files should compile without errors even if all annotations are removed. This rule precludes "meta-source" that is only turned into valid source by an annotation processor.

When annotations are compiled at the bytecode level, you need to analyze class files. The class file format is documented (see `http://java.sun.com/docs/books/vmspec`). The format is rather complex, and it would be challenging to process class files without special libraries. One such library is BCEL, the Bytecode Engineering Library, available at `http://jakarta.apache.org/bcel`.

In the Core Java chapter, we use BCEL to add logging messages to annotated methods. If a method is annotated with

```
@LogEntry(logger=loggerName)
```

then we add the bytecodes for the following statement at the beginning of the method:

```
Logger.getLogger(loggerName).entering(className, methodName);
```

Inserting these bytecodes sounds tricky, but BCEL makes it fairly straightforward.

You can run a bytecode engineering tool in two modes.You can simply write a tool that reads a class file, processes the annotations, injects the bytecodes, and writes the modified class file. Alternatively, you can defer the bytecode engineering until *load time*, when the class loader loads the class.

Before JDK 5.0, you had to write a custom classloader to achieve this task. Now, the *instrumentation API* has a hook for installing a bytecode transformer. The transformer must be installed before the `main` method of the program is called. You handle this requirement by defining an *agent*, a library that is loaded to monitor a program in some way. The agent code can carry out initializations in a `premain` method.

Here are the steps required to build an agent:

- Implement a class with a method:

  ```
  public static void premain(String arg, Instrumentation instr)
  ```

  This method is called when the agent is loaded. The agent can get a single command-line argument, which is passed in the `arg` parameter. The `instr` parameter can be used to install various hooks.
- Make a manifest file that sets the `Premain-Class` attribute, for example:

  ```
  Premain-Class: EntryLoggingAgent
  ```
- Package the agent code and the manifest into a JAR file, for example:

  ```
  jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf *.class
  ```

To launch a Java program together with the agent, use the following command-line options:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

Example 13-4 shows the agent code. The agent installs a class file transformer. The transformer first checks whether the class name matches the agent argument. If so, it uses the `EntryLogger` class from the preceding section to modify the bytecodes. However, the modified bytecodes are not saved to a file. Instead, the transformer returns them for loading into the virtual machine. In other words, this technique carries out "just in time" modification of the bytecodes.

Example 13-4: EntryLoggingAgent.java

```
1. import java.lang.instrument.*;
2. import java.io.*;
3. import java.security.*;
4.
5. import org.apache.bcel.classfile.*;
6. import org.apache.bcel.generic.*;
7.
8. public class EntryLoggingAgent
9. {
10.    public static void premain(final String arg, Instrumentation instr)
11.    {
12.       System.out.println(instr);
13.       instr.addTransformer(new
14.          ClassFileTransformer()
15.          {
16.             public byte[] transform(ClassLoader loader, String className, Class cl,
17.                ProtectionDomain pd, byte[] data)
18.             {
19.                if (!className.equals(arg)) return null;
20.                try
21.                {
22.                   Attribute.addAttributeReader("RuntimeVisibleAnnotations",
23.                      new AnnotationsAttributeReader());
24.                   ClassParser parser = new ClassParser(
25.                      new ByteArrayInputStream(data), className + ".java");
26.                   JavaClass jc = parser.parse();
27.                   ClassGen cg = new ClassGen(jc);
28.                   EntryLogger el = new EntryLogger(cg);
29.                   el.convert();
30.                   return cg.getJavaClass().getBytes();
31.                }
32.                catch (Exception e)
33.                {
34.                   e.printStackTrace();
35.                   return null;
36.                }
37.             }
38.          });
39.    }
40. }
```

Summary

In this article, you have learned

- how to add annotations to Java programs,
- how to design your own annotation interfaces, and
- how to implement tools that make use of the annotations.

It is easy to use annotations that someone else designed. It is also easy to design an annotation interface. But annotations are useless without tools. Building annotation tools is undeniably complex. The examples that we presented in this article give you an idea of the possibilities and perhaps has piqued your interest in developing your own tools.