Java SE Documentation

Annotations

Many APIs require a fair amount of boilerplate code. For example, in order to write a JAX-RPC web service, you must provide a paired interface and implementation. This boilerplate could be generated automatically by a tool if the program were "decorated" with annotations indicating which methods were remotely accessible.

Other APIs require "side files" to be maintained in parallel with programs. For example JavaBeans requires a `BeanInfo` class to be maintained in parallel with a bean, and Enterprise JavaBeans (EJB) requires a *deployment descriptor*. It would be more convenient and less error-prone if the information in these side files were maintained as annotations in the program itself.

The Java platform has always had various ad hoc annotation mechanisms. For example the `transient` modifier is an ad hoc annotation indicating that a field should be ignored by the serialization subsystem, and the `@deprecated` javadoc tag is an ad hoc annotation indicating that the method should no longer be used. The platform has a general purpose annotation (also known as *metadata*) facility that permits you to define and use your own annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, a class file representation for annotations, and annotation processing support provided by the `javac` tool.

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time.

Annotations complement javadoc tags. In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation.

Typical application programmers will never have to define an annotation type, but it is not hard to do so. 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. Method declarations must not have any parameters or a `throws` clause. Return types are restricted to primitives, `String`, `Class`, enums, annotations, and arrays of the preceding types. Methods can have *default values*. Here is an example annotation type declaration:

```
/**
 * 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]";
}
```

Once an annotation type is defined, you can use it to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used. By convention, annotations precede other modifiers. Annotations consist of an at-sign (`@`) followed by an annotation type and a parenthesized list of element-value pairs. The values must be compile-time constants. Here is a method declaration with an annotation corresponding to the annotation type declared above:

```
@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, for example:

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

It is permissible to omit the parentheses in marker annotations, as shown below:

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

In annotations with a single element, the element should be named `value`, as shown below:

```
/**
 * Associates a copyright notice with the annotated API element.
 */
public @interface Copyright {
    String value();
}
```

It is permissible to omit the element name and equals sign (`=`) in a single-element annotation whose element name is `value`, as shown below:

```
@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }
```

To tie it all together, we'll build a simple annotation-based test framework. First we need a marker annotation type to indicate that a method is a test method, and should be run by the testing tool:

```
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. Such annotations are called *meta-annotations*. The first (`@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. The second (`@Target(ElementType.METHOD)`) indicates that this annotation type can be used to annotate only method declarations.

Here is a sample program, some of whose methods are annotated with the above interface:

```
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() { }
}
```

Here is the testing tool:

```
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);
  }
}
```

The tool takes a class name as a command line argument and iterates over all the methods of the named class attempting to invoke each method that is annotated with the `Test` annotation type (defined above). The reflective query to find out if a method has a `Test` annotation is highlighted in green. If a test method invocation throws an exception, the test is deemed to have failed, and a failure report is printed. Finally, a summary is printed showing the number of tests that passed and failed. Here is how it looks when you run the testing tool on the `Foo` program (above):

```
$ 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
```

While this testing tool is clearly a toy, it demonstrates the power of annotations and could easily be extended to overcome its limitations.
