Java Programming

Annotation (JDK 1.5)

Introduction to Annotation

If you look inside a JDK package (says `java.lang`), besides classes, interfaces, exceptions and errors, JDK 1.5 introduces two new entities: enums and annotations.

Annotation gives you the ability to provide additional *metadata* alongside a Java entity (such as classes, interfaces, fields and methods). This additional metadata, called *annotation*, can be read and interrelated by the compiler or other utilities. They can also be stored in the class files. The runtime can discovered these *metadata* via the "reflection" API.

Prior to the introduction of this standardized annotation feature in JDK 1.5, Java provided only ad-hoc and non-standardized mechanism, including `Serializable` interface (a *tag* interface without a method) and `transient` modifier (not to serialize), Javadoc comments (used by the javadoc utility to generate documentation), `@deprecated` tag (indicates that the method should no longer be used).

One of the main reasons for adding annotation and metadata to the Java platform is to enable development and runtime tools to have a common infrastructure so as to reduce the effort required for development and deployment. A tool could use the metadata information to generate additional source code, or to provide additional information for debugging, among others. A typical application programmer does not have to define annotation.

Annotation is defined like a ordinary Java interface, but with an `'@'` preceding the interface keyword (i.e., `@interface`). You can declare methods inside an annotation definition (just like declaring abstract method inside an interface). These methods are called *elements* instead. The declarations must not have any formal parameters or a throw clause. Return types are restricted to primitives, `String`, `Class`, enum, annotation, and array of the preceding types. They can have default value.

Example: Annotation java.lang.Override

Let look at annotation type `java.lang.Override` API:

```
@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override
```

The purpose of annotation `Override` is to indicates that "a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message."

For example,

```
public class AnnotationOverrideTest {

   @Override public String toString() {
      return "Override the toString() of the superclass";
   }

   // Compilation Error because superclass Object does not have this method
   @Override public String toString123() {
      return "Override the toString123() of the superclass";
   }
}
```

```
AnnotationOverrideTest.java:7: method does not override or implement a method from a super-type
   @Override public String toString123() {
   ^
```

The following example shows how the `Override` annotation can save you many hours of fluctuation.

```
import java.awt.*;
import java.awt.event.*;
public class AnnotationOverrideDemo extends Frame {
   public AnnotationOverrideDemo() {
      this.addWindowListener(new WindowAdapter() {
         public void windowclosing(WindowEvent e) {
            System.exit(0);
         }
      });
      setSize(200, 100);
      setTitle("Annotation Override Demo");
      setVisible(true);
   }
```

```
    public static void main(String[] args) { new AnnotationOverrideDemo(); }
}
```

The above program look fine and cannot be compiled without error. However, clicking the window close button produces no effect. This is because `windowClosing()` is mis-spelled. Instead of overriding the event handler, it defines a new useless method.

Add annotation `@Override` to the `windowClosing()` as follows, whichs signal your intention, serves as documentation, and also allow the compiler to catch this error.

```
@Override
public void windowclosing(WindowEvent e) {
   System.exit(0);
}
```

The source code for `java.lang.Override` is as follows:

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

The package `java.lang.annotation` defines the following annotation types, that can be served as the *super-type* of the annotations:

- `@Retention`: specifies how long the annotation information is to be kept, takes value from enum `RetensionPolicy` of {`SOURCE`, `CLASS`, `RUNTIME`} and default to `RetensionPolicy.CLASS`.
- `@Target`: specifies the kinds of program element to which this annotation is applicable, takes value from enum `ElementType` of {`TYPE`, `FILED`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, `LOCAL_VARIABLE`, `ANNOTATION_TYPE`, `PACKAGE`}.

Annotation with *no element* is called a *marker annotation type*. For example,

```
public @interface UnderConstruction { }
```

You can annotate as follows:

```
@UnderConstruction
public class aMethod { ... }
```

If the annotation has a *single element*, it must be called `value()`.

```
public @interface Author {
    String value() default "Tan Ah Teck";
}
```

To use a single-element annotation, you can omit "value=":

```
@Author("Kevin Jones")
public class anotherMethod { ... }
```

[TODO] To be continued...

REFERENCES & RESOURCES

- TODO
- TODO

Latest version tested: JDK 1.6
Last modified: July 2008