Java Annotations

Robert J Stroud,

School of Computing Science,

University of Newcastle upon Tyne

JICC'11 Conference

26th Jan 2007

Introduction

- J2SE 5.0 added a lot of new language features to Java:

  `Generics, Enums, Auto-Boxing, Varargs, Annotations`
- Amongst all these features, Annotations or Metadata are perhaps the least well known
- And yet, according to the lead architect for J2SE 5.0, annotations were the single most important feature of the new release
- Adding annotations to Java makes it possible to adopt a more declarative style of programming
- For example, annotations have greatly simplified the Java EE 5 programming model
- But annotations will also have an impact on desktop programming:
- Increasingly, new APIs and tools will make use of annotations to simplify their programming model

Annotations and metadata

- An annotation is simply a tag in the source code that can be processed by a tool
- Javadoc comments are an example of such metadata
- However, the idea behind this language extension is to generalise the mechanism to allow arbitrary code annotations
- These can then be used by tools to extract information, enforce checks, or generate boiler-plate code
- Automatic generation of code for Web Services is an obvious application, as demonstrated by C# and some existing Java tools
- However, there are many other possible applications, limited only by the imagination of the tool writer
- J2SE 5.0 provided basic support for declaring annotations and chcking them for syntactic correctness
- J2SE 6.0 adds compiler support for processing annotations

Possible applications of annotations

- Object-relational mapping - support for persistent objects

  `@Entity, @Table, @Id, @OneToMany, etc.`
- Web services:

  `@WebService, @WebMethod, etc.`
- Remote objects (EJB)

  `@EJB, @Remote`
- Event Handling

  `@Action,@ActionListenerFor`
- Unit testing

  `@Test`
- Concurrency

  `@ThreadSafe, @GuardedBy`
- Static analysis

  `@NonNull, @PreCondition`

Example - persistence

- Most applications need access to some form of persistent storage
- Typically, the data layer of an application is provided by a database
- However, programs work with objects rather than relations
- Hence, there is a need to convert program objects into database objects
- Java provides access to databases via JDBC (Java Database Connectivity)
- This requires the results of database queries to be converted into objects
- A better approach is to use an Object Relation Mapping layer that takes care of this transformation automatically
- This is supported by the new Java Persistence API (JPA)

Example - book database

- Suppose we have a database containing information about books
- Books have properties such as author, title, and ISBN
- The database contains a table holding information about each book
- Each row of the table corresponds to a particular book object, and each column of the table corresponds to a particular book property
- Consider a query such as

```
select * from books where title like '%Java%'
```

- How do we convert the results of such a query into a list of book objects that can be manipulated by the program?
- Ideally, we would like to issue a database query and get back a list of book objects
- Instead, we issue a database query, and get back a database result set that has to be converted into a list of book objects

Example - Book class

```
class Book
{
   String author;
   String title;
   String isbn;

   void setAuthor(String author)
   {
       this.author = author;
   }

   String getAuthor()
   {
       return this.author;
   }

   …
}
```

Example - JDBC query

```
List getBooks( DataSource db, String query )
   throws SQLException
{
   Connection c = db.getConnection();
   Statement stmt = c.getStatement();
   ResultSet r = stmt.executeQuery(query);

   List list = new ArrayList();
   while ( r.hasNext() )
   {
       Book b = new Book();
       b.setAuthor(r.getString("author"));
       b.setTitle(r.getString("title"));
       b.setISBN(r.getString("isbn"));
       list.add(b);
   }
   return list;
}
```

Annotated Book class

```
@Entity
@Table(name = "books")
class Book
{
   @Id
   String isbn;
   String author;
   String title;

   void setAuthor(String author)
   {
       this.author = author;
   }

   String getAuthor()
   {
       return this.author;
   }

   …
}
```

Example - JPA query

```
List getBooks( EntityManager em, String query )
{
   Query q = em.createQuery(query);
   return q.list();
}
```

How Java Persistence works

- Class definitions for persistent objects are annotated with information about the corresponding database representation
- This makes it possible to turn "Plain Old Java Objects" (POJOs) into persistent objects by adding an `@Entity` annotation
- At run-time, the Java Persistence framework reads these annotations and creates the necessary object-relational mapping
- An `EntityManager` is used to manage the persistent objects
- The `EntityManager` is linked to a particular persistence context, and associated with a transaction
- Queries can be created and executed via the `EntityManager` object
- JPA supports its own query language, which is similar to SQL

Example - JPA application

```
// Obtain EntityManager
@PersistenceContext
EntityManager em;

// Start unit of work
EntityTransaction tx = em.getTransaction();
tx.begin();

// Do something
List<Book> books = em.createQuery
   ("select b from Book b order by b.title asc")
   .getResultList();
System.out.println( books.size() + " book(s) found" );
for (Book b : books)
{
   System.out.println(b.getTitle());
}

// End unit of work
tx.commit();
```

Example - Resource Injection

- Suppose a Java EE application needs to access a resource such as a database
- In J2EE 1.4, the programmer had to look up the resource explicitly using JNDI:

```
import javax.naming.*;

DataSource db = null;
try
{
   InitialContext ic = new InitialContext();
   db = ic.lookup("java:/comp/env/jdbc/myDB");
}
catch ( Exception e ) {… }
```

- In J2EE 5, resources are injected automatically using annotations:

```
@Resource
DataSource myDB;
```

Example - defining web services (JAX-WS)

- Use annotations to specify that a class implements a web service:

```
package helloservice;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
class Hello
{
   @WebMethod
   String sayHello(String name)
   {
     return "Hello " + name + ".";
   }
}
```

Example - defining web services (JAX-RPC)

- First define an interface that extends Remote:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface HelloIF extends Remote
{
   String sayHello(String s) throws RemoteException;
}
```

- Now define a class that implements this interface:

```
class HelloImpl implements HelloIF
{
   String sayHello(String s)
   {
          return "Hello " + s + ".";
   }
}
```

Example - calling a web service (JAX-RPC)

```
import javax.xml.rpc.Stub;

public class HelloClient
{
   public static void main(String[] args)
   {
      try
      {
         // Implementation specific…
         Stub stub = (Stub)
            (new MyHelloService_Impl().getHelloIFPort());
         stub._setProperty(ENDPOINT_ADDRESS_PROPERTY, "…");

         HelloIF hello = (HelloIF)stub;
         hello.sayHello("Robert"));
      }
      catch (Exception ex) { … }
   }
}
```

Example - calling a web service (JAX-WS)

```
import javax.xml.ws.WebServiceRef;
import helloservice.endpoint.HelloService;
import helloservice.endpoint.Hello;

public class HelloClient
{
   @WebServiceRef(wsdlLocation="…")
   static HelloService service;

   public static void main(String[] args)
   {
      try
      {
         Hello hello = service.getHelloPort();
         hello.sayHello("Robert");
      }
      catch(Exception e) { … }
   }
}
```

Example - defining actions in Swing

- Actions are a convenient of packaging up common event handlers, but they can be cumbersome to define:

```
// define sayHello Action - pops up a message Dialog
Action sayHello = new AbstractAction("Hello")
{
    public void actionPerformed(ActionEvent e)
    {
        String s = textField.getText();
        JOptionPane.showMessageDialog(s);
    }
};

// use sayHello - set the action property
textField.setAction(sayHello);
button.setAction(sayHello);
```

Example - defining actions in JSR296

- An annotation type for declaring actions is proposed:

```
@Action
public void sayHello()
{
    String s = textField.getText();
    JOptionPane.showMessageDialog(s);
}
```

- Action properties and bindings can be specified in resource files:

```
# resources/MyForm.properties
sayHello.Action.text = Say &Hello
sayHello.Action.icon = hello.png
sayHello.Action.accelerator = control H
sayHello.Action.shortDescription = Say hello modally

textField.action = sayHello
button.action = sayHello
```

How do annotations work?

- Annotations can be applied to packages, classes, parameters, variables, fields, or methods
- Annotations are defined using an @ syntax
- Each annotation has a corresponding annotation type, which defines the content of the metadata
- Annotation types are a special kind of interface that defines the format of metadata
- Meta-annotation types are used to define properties of annotation types
- Annotations can exist in source code or byte code only, or can be accessible at run-time using reflection

Examples of annotations

```
// Marker annotation
@Preliminary public class TimeTravel {... }

// Single-member annotation
@Copyright("2002 Yoyodyne Propulsion Systems, Inc.")
public class OscillationOverthruster {... }

// Array-valued single-member annotation
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop {... }

// Normal annotation
@RequestForEnhancement(
   id       = 2868724,
   synopsis = "Provide time-travel functionality",
   engineer = "Mr. Peabody",
   date     = "4/1/2004")
public static void travelThroughTime(Date destination){... }
```

Examples of annotation types

```
public @interface Preliminary { }

public @interface Copyright
{
   String value();
}

public @interface Endorsers
{
   String[] value();
}

public @interface RequestForEnhancement
{
   int        id();
   String     synopsis();
   String     engineer();
   String     date();
}
```

Built-in annotation types

- The following annotation types are defined in `java.lang` and have special meaning to the Java compiler:

```
@Deprecated
@Override
@SuppressWarnings
```

- Annotation types can be annotated using one of the following meta-annotation types, which are defined in `java.lang.annotation`:

```
@Documented
@Inherited
@Retention
@Target
```

Annotation processing

- Every annotation has a corresponding annotation type
- The Java compiler knows enough about annotations to check that each annotation is syntactically correct
- However, the semantics of the annotation are implemented by a separate annotation processor
- Only three basic annotations are built into the Java language, and therefore understood by the compiler:

```
@SuppressWarnings
@Deprecated
@Override
```

- However, the compiler will preserve annotations in the compiled byte code, and make them accessible at run time
- This makes it possible to use reflection to implement annotation processing at run-time

Example - unit testing (from JDK doc)

- JUnit is a simple but effective testing tool
- It makes it easy to run test methods automatically
- There is no need for a test program, because JUnit uses reflection to discover the names of the test methods
- The original version of JUnit depended on naming conventions, but the latest version uses annotations
- The following example shows how a simplified test framework could be implemented using annotations

Step 1 - define an @Test annotation

```
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 { }
```

Step 2 - annotate some source code

```
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() { }
}
```

Step 3 - run tests

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

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

Example - event handling (from Core Java)

- Suppose you want to arrange for a particular method to be called when an event occurs:

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

- Using annotations, you could simply write:

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

- The event handlers are installed at run-time by processing the annotations:

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

Step 1 - define annotation type

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
   String source();
}
```

Step 2 - create a JFrame

```
class ButtonFrame extends JFrame
{
    public ButtonFrame()
    {
        panel = new JPanel();
        add(panel);
        yellowButton = new JButton("Yellow");
        panel.add(yellowButton);

        ActionListenerInstaller.processAnnotations(this);
    }

    @ActionListenerFor(source="yellowButton")
    public void yellowBackground()
    {
        panel.setBackground(Color.YELLOW);
    }

    private JPanel panel;
    private JButton yellowButton;
}
```

Step 3 - process annotations

```
public static void processAnnotations(Object obj)
{
    try
    {
        Class cl = obj.getClass();
        for (Method m : cl.getDeclaredMethods())
        {
           ActionListenerFor a =
              m.getAnnotation(ActionListenerFor.class);
           if (a != null)
           {
              Field f = cl.getDeclaredField(a.source());
              f.setAccessible(true);
              addListener(f.get(obj), obj, m);
           }
        }
    }
    catch ( Exception e ) {… }
}
```

Summary

- Annotations are an important new language feature in Java
- The use of annotations in Java libraries and tools is rapidly increasing, both in standard Java APIs and 3rd party add-ons
- Enterprise Java has been greatly simplified by the introduction of annotations to support a more declarative programming model
- A number of JSR groups are working on standard annotations for particular aspects of Java
- New annotation types are easy to define, and annotation processing using reflection isn't much harder
- The J2SE 6 compiler can be extended by adding annotation processors to the class path of the compiler
- Annotation processing can also be done at load time
- Thus, the power of annotations is limited only by the imagination of the annotation creator

References

- "Annotations" section of Java Language guide:
  `…/docs/technotes/guides/language/annotations.html`
- Java EE 5 Tutorial and Java EE 1.4 Tutorial:
  `http://java.sun.com/javaee/reference/tutorials/`
- Chapter 13 of Core Java, Vol. 2 (7th Edition):
  `http://www.horstmann.com/corejava/cj7v2ch13ex.pdf`