A Metadata Facility for Java

Simon C. Nash

IBM Distinguished Engineer
Hursley, UK
nash@hursley.ibm.com
Agenda

- Background and motivation
- JSR 175 specification
  - Source syntax
  - Processing model
  - Runtime API
  - Class file format
  - Standard annotations
  - Annotation type evolution
Background

- Java programs are imperative (how), not declarative (what)
- Java has declarations, but only for core language constructs (class, method, etc.)
- The programmer can’t declare other things about the program
- This limits the scope for tools to help the developer be more productive
The Solution

- Allow user-defined metadata annotations to be declared within a Java program
- These annotations can be processed by tools or at runtime
- They aren’t part of the program logic but can (in some sense) affect the meaning of the program
- Semantics can be standard or user-defined
Benefits

- Productivity – quicker to write
- Clarity – easier to understand
- Flexibility – easier to change
- Correctness – fewer runtime errors

*Caveat:* Keep annotations simple and readable
Other Examples of Metadata in Java

- J2EE deployment descriptors
- Type information in class files
- Descriptive information about images and streams (e.g., JPEG)
- Information about JDBC databases
- Information about JMX Mbeans
- Javadoc comments
Examples of Metadata in .NET

- Definition of Web service classes and methods
- Control of data serialization
- Security enforcement
- Design-time support for components
Terminology

- Metadata is a very widely used term
- JSR 175 (the subject of this talk) is about program annotations for Java
- Program annotations are a kind of metadata, and JSR 175 was originally entitled “A Metadata Facility for the Java Programming Language”
Terminology (Continued)

- To avoid confusion and clarify the intended scope of JSR 175, it is now entitled “A Program Annotation Facility for the Java Programming Language”
JSR 175 Annotations

- Simple metadata specified in source code
- Attached to program syntactic elements
- Limited range of types
- Stored in the class file
- Can be read by tools or at runtime
- Introduced in J2SE 1.5
Annotation Syntax

@WebService(contextRoot = "Demo",
    style = WRAPPED,
    sei = StockQuote.class,
    serviceElement = "StockQuoteService")

public class Demo1 {

    ...

    @Operation public float getPrice(String symbol) {
        ...
    }

}

@Namespace("www.ibm.com/Demo")
public interface StockQuote extends java.rmi.Remote {
    ...
}
Annotations

- Annotations have types, members, and values
- General syntax, with some shorthand forms for common cases
- Optional defaults for member values
Annotation Type Declarations

```java
public @interface WebService {
    String contextRoot();
    WSDLStyle style();
    WSDLUse use() default LITERAL;
    Class<? extends java.rmi.Remote> sei();
    String serviceElement();
}

class public enum WSDLStyle {WRAPPED, DOC, RPC}

class public enum WSDLUse {LITERAL, ENCODED}

public @interface Operation { }

public @interface Namespace { String value(); }
```
More on Annotation Type

Declarations

- Packaged and imported like other Java types
- Members can be primitive, String, Class, enum, array, or annotation
- Can themselves be annotated using standard meta-annotation types
  - @Target(ElementTypeArray)
  - @Retention(RetentionPolicy)
  - @Inherited
  - @Documented
Meta-annotations

@Target(TYPE) public @interface WebService {
    String contextRoot();
    WSDLStyle style();
    WSDLUse use() default LITERAL;
    Class<? extends java.rmi.Remote> sei();
    String serviceElement();
}

public enum WSDLStyle {WRAPPED, DOC, RPC}
public enum WSDLUse {LITERAL, ENCODED}

@Target(METHOD) @Retention(RUNTIME)
public @interface Operation { }

@Target({TYPE, METHOD})
public @interface Namespace { String value(); }
Retention Policy

- Runtime retention
  - annotations are present in source code, the class file, and at runtime
- Class retention (the default)
  - annotations are present in source code and the class file, but are not present at runtime
- Source retention
  - annotations are only present in source code and are not present in the class file or at runtime
Complex Annotations

- Annotation types can be used to declare other annotation types

```java
public @interface Name {
    String first();
    String last();
}

public @interface Author {
    Name value();
}

public @interface Reviewer {
    Name value();
}
```
Using Annotations

- Annotations can be used anywhere that declaration modifiers can appear
  - class, interface, field, method, parameter, constructor, enum, local variable
  - also on enum constants and packages
    - package annotation placed in package-info.java
- No repeated annotations of same type
- By convention, annotations precede other modifiers
Writing Annotations

- Three kinds of annotation:
  - Normal annotation
  - Marker annotation
  - Single-value annotation
- The last two are shorthands for the first
Writing Annotations

// Normal annotations
@Copyright(date = 2003, owner = “IBM Corporation”)
@Author(name = “Simon Nash”)
@Authors(names = {“Tom”, “Dick”, “Harry”})  // array value
@Authors(names = {“Tom”}) // single-element array
@Authors(names = “Tom”)  // shorthand for single-element array
@WebService()

// Marker annotation
@WebService  // same as @WebService()

// Single-value annotation
@Name(“Jane Doe”)  // same as @Name(value = “Jane Doe”)
Processing Models

- Source based: process annotations from the source file
- Class based: process annotations from the class file
Source-based Processing

- Source file
  - Java compiler
  - Annotation processor
  - Annotated class file
Class-based Processing

source file -> Java compiler -> annotated class file

annotation processor
Integrated Tool Set

source file → Java compiler and annotation processor → annotated class file
Source-based Processing

- Use source file to perform annotation-specific error checking, any other required processing, and write annotations to the class file
  - ✓ provides best diagnostics, with line numbers for error locations
  - ✓ can handle all annotation types
  - ❌ requires source file to be available
Class-based Processing

- Use standard Java compiler to create class file containing same annotations as source
- Do annotation-specific error checking and any other required processing by reading annotations from the class file
  - Doesn’t need the source to be available
  - Can’t handle annotations with SOURCE retention or on local variables
  - Difficult to provide line numbers in diagnostics
Reading Annotations

- **Introspectors**: programs that query their own annotations at runtime
  - can load both annotation types and programs
- **Specific tools**: programs that query known annotation types of arbitrary other programs
  - can load annotation types but not programs
- **General tools**: programs that query arbitrary annotation types of arbitrary other programs
  - can’t load annotation types or programs
APIs for Reading Annotations

- **Introspectors**
  - JSR 175 reflection extensions

- **Specific tools**
  - JSR 175 reflection extensions may be usable if "class mirror" APIs implement these interfaces

- **General tools**
  - No standard "arm’s length" API available
  - JSR 175 has sketch of Doclet API extensions in non-normative appendix
JSR 175 Reflection Extensions

// Implemented by java.lang.Class,  
// java.lang.reflect.AccessibleObject, and java.lang.Package  
public interface AnnotatedElement {  
    boolean isAnnotationPresent(  
        Class<? extends Annotation> annotationType);  

    <T extends Annotation> T getAnnotation(  
        Class<T> annotationType);  

    Annotation[] getAnnotations();  

    Annotation[] getDeclaredAnnotations();  
}
JSR 175 Reflection Extensions

// Added to java.lang.reflect.Method and
// java.lang.reflect.Constructor
public Annotation[][] getParameterAnnotations();

// Added to java.lang, thrown by methods on annotation
// interfaces that return Class types
public class ClassNotPresentException
    extends RuntimeException {
    public ClassNotPresentException(String className,
        Throwable cause) {}
    public String className() {}
}

// Added to java.lang.Class
public boolean isAnnotationType();
Class File Format

- **RuntimeVisibleAnnotations attribute**
  - contains array of annotation structs
  - each of these contains an array of member-value pairs
  - member_value is a union of all possible types for values

- **RuntimeInvisibleAnnotations attribute**

- **RuntimeVisibleParameterAnnotations attribute**

- **RuntimeInvisibleParameterAnnotations attribute**

- **AnnotationDefault attribute**

- For more details, read the JSR 175 spec
Standard Annotations

- Meta-annotations (already described)
- `java.lang.Overrides`

  ```
  // specifies that the target method must be an override
  @Target(METHOD) public @interface Overrides { }
  ```

- As specified by other JSRs:
  - 181, 207, 224, etc.
Annotation Type Evolution

- Can add members, should specify default
  - java.lang.IncompleteAnnotationError otherwise
- Removing members is OK if they’re not accessed
  - java.lang.NoSuchMethodError otherwise
- Changing member types is problematical
  - AssociationTypeMismatchError
Type Evolution (Continued)

- Adding defaults is OK; don’t remove them
  - compile failure, IncompleteAnnotationError

- Adding targets is OK; don’t remove them
  - compile failure, silently discarded at runtime

- Changing retention policy is OK
  - downgrade takes effect immediately
  - upgrade takes effect on recompile
JSR 175 limitations

- Not a general solution for Java metadata
  - no programmatic way to create or update annotations
  - annotations in source and class files only
  - limited set of annotation types
- Not intended for modifying source code (a.k.a. preprocessing)
- No support for J2SE 1.4 or earlier
- Annotation type inheritance not permitted
JSR 175 limitations (Continued)

- Can’t attach annotations to arbitrary Java code blocks or statements
- Some annotations can’t be written to the class file
  - local variable annotations are source-only
- No standard “arm’s length” API for reading annotations without loading classes
Use Cases for Annotations

- Simplified Web services development/deployment
- Simplified EJB or Servlet development/deployment
- Java to XML mapping
- WSDL to Java mapping
- BeanInfo replacement
- Relational database schema mapping
- Custom serialization
- Identifying optimization opportunities
- etc., etc.
JSR 181

- Web services annotations for simplified development and deployment
- Based on JSR 175 (may include transitional support for J2SE 1.4)
- Example attributes: operation, protocol, target-namespace, wsdl, schema, xmlns
- Can process annotations from source file or from class file
- Other issues still being discussed by Expert Group
JSR 181 Example

@Protocol(httpSoap=true, soapStyle=documentLiteral)
@TargetNamespace(namespace=http://schemas.myDomain.com/ws/)
public class MyWebService {
    @Operation
    public double zipDistance(
        String fromZip, String toZip) {
        ...
        return distance.getDistance(fromZip, toZip);
    }
    ...
}
References

- **JSR 175:**

- **JSR 181:**

- **J2SE 1.5 (Tiger):**
Summary

- Annotations expand the scope of Java programming
  - more declarative, less imperative
- JSR 175 provides the base framework, but the value comes from the annotations that people create
  - what will be the killer annotations?
- Tools are the key
Questions?