Annotation File Specification

JSR 308 Team
MIT Computer Science and Artificial Intelligence Lab
jsr308@csail.mit.edu

December 7, 2008

1 Purpose: External storage of annotations

Java annotations are meta-data about Java program elements, as in "`@Deprecated class Date { ... }`". Ordinarily, Java annotations are written in the source code of a `.java` Java source file. When `javac` compiles the source code, it inserts the annotations in the resulting `.class` file (as "attributes").

Sometimes, it is convenient to specify the annotations outside the source code or the `.class` file.

- When source code is not available, a textual file provides a format for writing and storing annotations that is much easier to read and modify than a `.class` file. Even if the eventual purpose is to insert the annotations in the `.class` file, the annotations must be specified in some textual format first.
- Even when source code is available, sometimes it should not be changed, yet annotations must be stored somewhere for use by tools.
- A textual file for annotations can eliminate code clutter. A developer performing some specialized task (such as code verification, parallelization, etc.) can store annotations in an annotation file without changing the main version of the source code. (The developer's private version of the code could contain the annotations, but the developer could copy them to the separate file before committing changes.)
- Tool writers may find it more convenient to use a textual file, rather than writing a Java or `.class` file parser.
- When debugging annotation-processing tools, a textual file format (extracted from the Java or `.class` files) is easier to read, and is easier for use in testing.

All of these uses require an external, textual file format for Java annotations. The external file format should be easy for people to create, read, and modify. An "annotation file" serves this purpose by specifying a set of Java annotations.

The file format discussed in this document supports both standard Java SE 6 annotations and also the extended annotations proposed in JSR 308 [Ern07]. Section "Class File Format Extensions" of the JSR 308 design document explains how the extended annotations are stored in the `.class` file. The annotation file closely follows the class file format. In that sense, the current design is extremely low-level, and users probably would not want to write the files by hand (but might fill in a template that a tool generated automatically). As future work, we should design a more user-friendly format that permits Java signatures to be directly specified. Furthermore, since the current design is closely aligned to the class file, it is convenient for tools that operate on `.class` files but less convenient for tools that operate on `.java` files. For the short term, the low-level format will serve our purpose, which is primarily to enable testing by the Javari developers.

2 Grammar conventions

Throughout this document, "name" is any valid Java simple name or fully qualified name, "type" is any valid type, and "value" is any valid Java constant, and quoted strings are literal values. The Kleene qualifiers "*" (zero or more), "?" (zero or one), and "+" (one or more) denote plurality of a grammar element. Parentheses ("()") denote grouping, and square brackets ("[]") denote optional syntax, which is equivalent to "(...) ?".

In the annotation file, whitespace (excluding newlines) is optional with one exception: no space is permitted between an "@" character and a subsequent name. Indentation is ignored, but is encouraged to maintain readability of the hierarchy of program elements in the class (see the example in section 3).

Comments can be written throughout the annotation file using the double-slash syntax employed by Java for single-line comments: anything following two adjacent slashes ("//") until the first newline is a comment. This is omitted from the grammar for simplicity. Block comments ("/* ... */") are not allowed.

2.1 annotation file

The annotation file itself contains one or more package definitions; each package definition describes one or more annotations and classes in that package.

annotation-file ::=
  package-definition+

2.2 Package Definitions

Package definitions describe a package containing a list of annotation definitions and classes. A package definition also contains any annotations on the package itself (such as those from a `package-info.java` file).

package-definition ::=
  # To specify the default package, omit the name.
  # Annotations on the default package are not allowed.
  "**package**" [*name*? "**:**" *annotation**] "\n"
  (*annotation-definition* — *class-definition*) *

2.3 Annotation Definitions

An annotation definition describes the annotation's fields and their types, so that they may be referenced in a compact way throughout the annotation file. An annotation must be defined in an annotation file before it may be used, either on a program element or as a field of another annotation definition.

If an annotation file uses an annotation type at least once to directly annotate a program element, the annotation definition must include a retention policy; if the annotation type is used only as a field of other annotations, the retention policy is optional.

annotation-definition ::=
  "**annotation**" [*retention-policy*] "**@**" *name*
  ["**:**" *annotation-field-definition*+]
  "\n"

annotation-field-definition ::=
  type name "\n"

retention-policy ::=
  "**visible**" # Equivalent to `@Retention(RUNTIME)` in source
  — "**invisible**" # Equivalent to `@Retention(CLASS)` in source
  — "**source**" # Equivalent to `@Retention(SOURCE)` in source

2.4 Class Definitions

Class definitions describe the annotations present on the various program elements. It is organized according to the hierarchy of fields and methods in the class. Program elements that are not annotated may be present or may be omitted. Class definitions are defined by the `class-definition` production of the following grammar.

Inner classes are treated as ordinary classes whose names happen to contain `$` signs and must be defined at the top level of a class definition file. (To change this, the grammar would have to be extended with a closing delimiter for classes; otherwise, it would be ambiguous whether a field/method appearing after an inner class definition belonged to the inner class or the outer class.)

annotation ::=
  # The name may be the annotation's simple name, unless the file
  # contains definitions for two annotations with the same simple name.
  # In this case, the fully-qualified annotation name is required.
  "**@**"*name* ["**(**" *annotation-field* ["**,**" *annotation-field*]+ "**)**"]

annotation-field ::=
  # In Java, single-field annotations often have the field
  # name "`value`", and that field name is elided in uses of the
  # annotation: "`@A(12)`" rather than "`@A(value=12)`". In an
  # annotation file, however, the "`value=`" is always required.
  name "**=**" *value*

class-definition ::=
  "**class**" *name* "**:**" *annotation** "\n"
  *bound-definition**
  *field-definition**
  *method-definition**

field-definition ::=
  "**field**" *name* "**:**" *annotation** "\n"
  *type-argument-or-array-definition**

method-definition ::=
  # The method key consists of the name followed by the signature
  # in JVML format, for example: `foo([ILjava/lang/String;)V`
  "**method**" *method-key* "**:**" *annotation** "\n"
  *bound-definition**
  *type-argument-or-array-definition**
  *parameter-definition**
  receiver-definition?
  *variable-definition**
  *typecast-definition**
  *instanceof-definition**
  *new-definition**

type-argument-or-array-definition ::=
  # The integer list here contains the values of the "location"
  # array [Ern07].
  "**inner-type**" *integer* ["**,**" *integer*]+ "**:**" *annotation** "\n"

bound-definition ::=

# The integers are respectively the parameter and bound indices of
# the type parameter bound [Ern07].
"**bound**" ,*integer* &*integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

receiver-definition ::=
"**receiver**" *annotation** "\n"

parameter-definition ::=
# the integer is the index of the parameter in the method
# (i.e., 0 is the first method parameter)
"**parameter**" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

variable-definition ::=
# The integers are respectively the index, start, and length
# fields of the annotations on this variable [Ern07].
"**local**" *integer* "#" *integer* "+" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

typecast-definition ::=
# The integer is the offset field of the annotation [Ern07].
"**typecast**" "#" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

instanceof-definition ::=
# The integer is the offset field of the annotation [Ern07].
"**instanceof**" "#" *integer* "**:**" *annotation** "\n"

new-definition ::=
# the integer is the offset field of the annotation [Ern07].
"**new**" "#" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

2.5 Dependence on bytecode offsets

For annotations on expressions (typecasts, instanceof, new, etc.), the annotation file uses offsets into the bytecode array of the class file to indicate the specific expression to which the annotation refers. Because different compilation strategies yield different `.class` files, a tool that maps such annotations from an annotation file into source code must have access to the specific `.class` file that was used to generate the annotation file. For non-expression annotations such as those on methods, fields, classes, etc., the `.class` file is not necessary.

3 Example

Consider the code of Figure 1. Figure 2 shows two legal annotation files each of which represents its annotations.

```
package p1;

import p2.*; // for the annotations @A through @D

public @A(12) class Foo {

    public int bar;             // no annotation
    private @B List<@C String> baz;

    public Foo(@B List<@C String> a) @D("spam") {
        @B List<@C String> l = new LinkedList<@C String>();
        l = (@B List<@C String>)l;
    }
}
```

Figure 1: Example Java code with annotations.

```
package p2:
annotation @A:
    int value
annotation @B:
annotation @C:
annotation @D:
    String value

package p1:
class Foo: @A(value=12)

    field bar:

    field baz: @B
        inner-type 0: @C

    method <init>:
        parameter #0: @B
            inner-type 0: @C
        receiver: @D(value="spam")
        local 1 #3+5: @B
            inner-type 0: @C
        typecast #7: @B
            inner-type 0: @C
        new #0:
            inner-type 0: @C
```

```
package p2:
annotation @A
    int value

package p2:
annotation @B

package p2:
annotation @C

package p2:
annotation @D
    String value

package p1:
class Foo: @A(value=12)

package p1:
class Foo:
    field baz: @B

package p1:
class Foo:
    field baz:
        inner-type 0: @C

// ... definitions for p1.Foo.<init>()
// omitted for brevity
```

Figure 2: Two distinct annotation files each corresponding to the code of Figure 1.

4 Types and Values

The Java language permits several types for annotation fields: primitives, `String`s, `java.lang.Class` tokens (possibly parameterized), enumeration constants, subannotations, and one-dimensional arrays of these. These types are represented in an annotation file as follows:

- Primitive: the name of the primitive type, such as `boolean`.
- String: `String`.
- Class token: `Class`; the parameterization, if any, is not represented in annotation files.
- Enumeration constant: `enum` followed by the fully qualified name of the enumeration class, such as `enum java.lang.Thread$State`.
- Array: The representation of the element type followed by `[]`, such as `String[]`, with one exception: an annotation definition may specify a field type as `unknown[]` if, in all occurrences of that annotation in the annotation file, the field value is a zero-length array.[1]

Annotation field values are represented in an annotation file as follows:

- Numeric primitive value: literals as they would appear in Java source code.
- Boolean: `true` or `false`.
- Character: A single character or escape sequence in single quotes, such as `'A'` or `'\''`.
- String: A string literal as it would appear in source code, such as `"\"Yields falsehood when quined\" yields falsehood when quined."`.
- Class token: The fully qualified name of the class (using `$` for inner classes) or the name of the primitive type or `void`, possibly followed by `[]`s representing array layers, followed by `.class`. Examples: `java.lang.Integer[].class`, `java.util.Map$Entry.class`, and `int.class`.
- Enumeration constant: the name of the enumeration constant, such as `RUNNABLE`.
- Array: a sequence of elements inside `{}` with a comma between each pair of adjacent elements; a comma following the last element is optional as in Java. Examples: `{1}`, `{true, false,}` and `{}`.

The following example annotation file shows how types and values are represented.

```
package p1:

annotation @ClassInfo:
    String remark
    Class favoriteClass
    Class favoriteCollection // it's probably Class<? extends Collection>
                             // in source, but no parameterization here
    char favoriteLetter
    boolean isBuggy
    enum p1.DebugCategory[] defaultDebugCategories
    @p1.CommitInfo lastCommit
```

[1]There is a design flaw in the format of array field values in a class file. An array does not itself specify an element type; instead, each element specifies its type. If the annotation type `X` has an array field `arr` but `arr` is zero-length in every `@X` annotation in the class file, there is no way to determine the element type of `arr` from the class file. This exception makes it possible to define `X` when the class file is converted to an annotation file.

```
annotation @CommitInfo:
    byte[] hashCode
    int unixTime
    String author
    String message

class Foo: @p1.ClassInfo(
    remark="Anything named \"Foo\" is bound to be good!",
    favoriteClass=java.lang.reflect.Proxy.class,
    favoriteCollection=java.util.LinkedHashSet.class,
    favoriteLetter='F',
    isBuggy=true,
    defaultDebugCategories={DEBUG_TRAVERSAL, DEBUG_STORES, DEBUG_IO},
    lastCommit=@p1.CommitInfo(
        hashCode={31, 41, 59, 26, 53, 58, 97, 92, 32, 38, 46, 26, 43, 38, 32, 79},
        unixTime=1152109350,
        author="Joe Programmer",
        message="First implementation of Foo"
    )
)
```

5 Alternative formats

We mention two alternatives to the format described in this document. Each of them has its own merits. In the future, the other formats could be implemented, along with tools for converting among them.

An alternative to the format described in this document would be XML. XML does not seem to provide any compelling advantages. Programmers interact with annotation files in two ways: textually (when reading, writing, and editing annotation files) and programmatically (when writing annotation-processing tools). Textually, XML can be very hard to read; style sheets mitigate this problem, but editing XML files remains tedious and error-prone. Programmatically, a layer of abstraction (an API) is needed in any event, so it makes little difference what the underlying textual representation is. XML files are easier to parse, but the parsing code only needs to be written once and is abstracted away by an API to the data structure.

Another alternative is a format like the `.spec`/`.jml` files of JML [LBR06]. The format is similar to Java code, but all method bodies are empty, and users can annotate the public members of a class. This is easy for Java programmers to read and understand. (It is a bit more complex to implement, but that is not particularly germane.) Because it does not permit complete specification of a class's annotations (it does not permit annotation of method bodies), it is not appropriate for certain tools, such as type inference tools. However, it might be desirable to adopt such a format for public members, and to use the format described in this document primarily for method bodies.

References

[Ern07] Michael D. Ernst. Annotations on Java types: JSR 308 working document. `http://pag.csail.mit.edu/jsr308/`, November 12, 2007.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3), March 2006.