

Annotation File Specification

Javari Team
MIT Computer Science and Artificial Intelligence Lab
javari@csail.mit.edu

October 2, 2007

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 [EC06]. 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.

1.1 Alternative formats

We mention two alternatives to the format described in this document. Each of them has its own merits. Probably, all three formats should be implemented, along with tools for converting among them. Then, we can see which of the formats programmers prefer in practice.

An alternative to the format described in this document would be XML. It would be easy to use an XML format to augment the one proposed here, but XML does not seem to provide any compelling advantages. Programmers will 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 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.

Yet 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 better to adopt such a format for all public members, and to use the format described in this document primarily for method bodies.

2 Grammar conventions

The Kleene qualifiers “*” (zero or more) and “+” (one or more) denote plurality of a grammar element. Parentheses (“()”) denote grouping. Square brackets (“[]”) denote optional syntax.

In the annotation file, all whitespace is optional, except where there is no whitespace in the grammar. Specifically, no space is permitted between an “@” character and a subsequent name, and no space is allowed in a compound name on either side of the “.”. Indentation is ignored, but is encouraged to maintain readability of the hierarchy of program elements in the class (see the example in section 2.8).

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 Names and Types

The annotation file format uses JLS conventions for identifying fields, methods, classes and primitive types [GJSB05]. A *name* is either a simple name consisting of just one identifier, or a qualified name consisting of a name, a dot, and an identifier. An *identifier* is defined in JLS 3.8. A *method-signature* consists of the method's name, followed by parentheses around the type parameter list. Each type is the type of the parameter after erasure. The return type of a method is not included in its signature.

By default, every class name must be fully qualified. Even classes in `java.lang` must be fully qualified, as in `java.lang.Object`.

```
name ::=
    # Note that an identifier is allowed to include a $.
    # There cannot be a space on either side of the dot in a compound name.
    identifier |
    name.identifier
```

```
method-signature ::=
    # For constructors, the class name is used instead of “<init>”
    name( [ type [, type]* ] )
```

```
type ::=
    primitive-type |
    name |
```

`type[]` # The brackets are literals, denoting an array type.

```
primitive-type ::=  
    boolean | byte | char | double | float | int | long | short
```

2.2 Annotation file

The annotation file itself contains annotation definitions, which describe the types of annotations to be used, along with package and class definitions that place those annotations on program elements.

```
annotation-file ::=  
    annotation-definition*  
    package-definition*  
    class-definition*
```

2.3 Annotation Definitions

An annotation definition describes the annotation's fields and their types. Annotation fields are restricted to those annotations enumerated in *annotation-element-type*.

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's 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 [, annotation-field-definition]* ;]  
    }
```

```
annotation-field-definition ::=  
    annotation-element-type name
```

```
annotation-element-type ::=  
    # Denotes an option of declaring a single array type.  
    # If used, there should be no whitespace before the square brackets.  
    annotation-element-base-type [[]] |  
    unknown[] # For empty arrays. See section 3.
```

```
annotation-element-base-type ::=  
    primitive-type |  
    String |  
    Class |  
    @name |  
    enum name
```

```
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 Annotations

An annotation describes an *annotation-definition* being applied to some program element. See section 3 for more on allowable types and values for annotation elements.

```
annotation ::=
    @name |
    @name() |
    @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 = annotation-field-value

annotation-field-value ::=
    numeric-primitive | # i.e. 3, 4.5, 20070704L, 3.14e2
    boolean-primitive | # true, false
    character-literal | # i.e. 'A', '\'
    string-literal | # i.e. "", "Hello, world!"
    enumeration-constant | # i.e. RUNNABLE
    class-token |
    array-value |
    annotation # For nested annotations, as in: "@A(b=@B)" or "@A(b=@B(value=1))"

class-token ::=
    # The entire token must end in .class, and have no whitespace.
    class-token-name.class

class-token-name ::=
    # If used, no whitespace is allowed before the square brackets.
    # This is different from annotation-element-type in that nested arrays are allowed.
    name( [ ])*

array-value ::=
    # A comma-separated list of values, enclosed in braces.
    # A trailing comma after the last element is optional.
    { [annotation-field-value [, annotation-field-value]* [, ] ] }
```

2.5 Package Definitions

Package definitions describe the annotations on a package (specified via the `package-info.java` file).

```
package-definition ::=
    # Annotations on the default package are not allowed.
    package name annotation* ;
```

2.6 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.

class-definition ::=

```
class name annotation* {
    bound-definition*
    field-definition*
    method-definition*
}
```

field-definition ::=

```
field name annotation* {
    type-argument-or-array-definition*
}
```

method-definition ::=

```
method method-signature annotation* {
    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; see [EC06].
inner-type integer [, integer]* annotation* ;
```

bound-definition ::=

```
# The integers are respectively the parameter and bound indices of
# the type parameter bound; see [EC06].
bound ,integer &integer annotation* {
    type-argument-or-array-definition*
}
```

receiver-definition ::=

```
receiver annotation* ;
```

parameter-definition ::=

```
# the integer is the index of the parameter in the method
# (i.e., 0 is the first method parameter)
parameter integer annotation* {
    type-argument-or-array-definition*
}
```

variable-definition ::=

```
# The integers are respectively the index, start, and length
```

```

# fields of the annotations on this variable; see [EC06].
local integer # integer + integer annotation* {
    type-argument-or-array-definition*
}

typecast-definition ::=
# The integer is the offset field of the annotation;
# see [EC06].
typecast # integer annotation* {
    type-argument-or-array-definition*
}

instanceof-definition ::=
# The integer is the offset field of the annotation; see [EC06].
instanceof # integer annotation* ;

new-definition ::=
# the integer is the offset field of the annotation; see [EC06].
new # integer annotation* {
    type-argument-or-array-definition*
}

```

2.7 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.

2.8 Example

Figure 1 lists Java code containing various annotations. Figure 2 shows two legal annotation files, each of which represents these annotations.

3 Types and Values

The Java language permits several types for annotation elements: primitives, `Strings`, `java.lang.Class` tokens (possibly parameterized), enumeration constants, subannotations, and one-dimensional arrays of these types. The `annotation-element-type` production enumerates these types. Here is a more thorough description of how each type is represented in an annotation file:

- 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`.

```

package p1;

import p2.*;
// Package p2 contains annotations @A through @D.
// @A has a single element, of type int.
// @D has a single element, of type String.

public @A(12) @B @C 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> foolist = new LinkedList<@C String>();
        foolist = (@B List<@C String>)foolist;
    }
}

```

Figure 1: Example Java code with annotations.

- 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 the field value is a zero-length array in all occurrences of that annotation in the annotation file.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 `$` for inner classes), the name of the primitive type or `void`, possibly followed by []s representing array layers, followed by `.class`. Examples: `java.lang.Integer[].class`, `java.util.Map$Entry.class`, and `int.class`.
- Enumeration constant: the name of the enumeration constant, such as `RUNNABLE`.
- Array: a sequence of elements inside braces {}, with a comma between each pair of adjacent elements; as in Java, a comma following the last element is optional. Examples: `{1}`, `{true, false,}` and `{}`.

Figure 3 lists an annotation file that demonstrates how various types and values are represented.

References

[EC06] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. <http://pag.csail.mit.edu/jsr308/>, October 17, 2006.

¹There 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 @p2.A {
    int value;
}
annotation @p2.B {}
annotation @p2.C {}
annotation @p2.D {
    String value;
}

class p1.Foo @A(value=12) @B @C {
    field bar {}
    field baz @B {
        inner-type 0 @C;
    }

    method Foo(java.util.List) {
        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;
        }
    }
}

```

Figure 2: The annotation file corresponding to the code of figure 1.

- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.
- [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.

```

annotation @p1.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;
}

annotation @p1.CommitInfo {
    byte[] hashCode,
    int unixTime,
    String author,
    String message;
}

// A very extensive annotation on class Foo.
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"
    )
) {
    // Annotations on Foo's elements.
}

```

Figure 3: An annotation file demonstrating how various types and values are represented.