Annotation File Format Specification

http://types.cs.washington.edu/annotation-file-utilities/
March 1, 2013

Contents

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 Annotation File Utilities (`http://types.cs.washington.edu/annotation-file-utilities/`) are a set of tools that process annotation files.

The file format discussed in this document supports both standard Java SE 6 annotations and also the extended annotations proposed in JSR 308 [Ern08]. 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.

By convention, an annotation file ends with "`.jaif`" (for "Java annotation index file"), but this is not required.

2 Grammar

2.1 Grammar conventions

Throughout this document, "name" is any valid Java simple name or binary 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. A vertical bar ("|") separates alternatives. Parentheses ("()") denote grouping, and square brackets ("[]") denote optional syntax, which is equivalent to "(...) ?" but more concise. We use the hash/pound/octothorpe symbol ("#") for comments within the grammar.

In the annotation file, besides its use as token separator, 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.

The line end symbol "`\n`" is used for all the different line end conventions, that is, Windows- and Unix-style new lines are supported.

2.2 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+

The annotation file may omit certain program elements — for instance, it may mention only some of the packages in your program, or only some of the classes in a package, or only some of the fields or methods of a class. Program elements that do not appear in the annotation file are treated as unannotated.

2.3 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*) *

2.4 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.

The Annotation File Utilities can read annotation definitions from the classpath, so it is optional to define them in the annotation file.

If an annotation is defined in the annotation file, then it must be defined before it is used. (This requirement makes it impossible to define, in an annotation file, an annotation that is meta-annotated with itself.) In the annotation file, the annotation definition appears within the package that defines the annotation. The annotation may be applied to elements of any package.

annotation-definition ::=
    "**annotation**" "**@**"*name* "**:**" *annotation** "\n"
    *annotation-field-definition**

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

annotation-field-type ::=
    # *primitive-type* is any Java primitive type (`int`, `boolean`, etc.).
    # These are described in detail in Section 4.
    (*primitive-type* | "**String**" | "**Class**" | ("**enum**" *name*) | ("**annotation-field**" *name*)) "[]"?
    | "**unknown[]**" "\n"

2.5 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. Class definitions are defined by the `class-definition` production of the following grammar. Note that we use `class-definition` also for interfaces, enums, and annotation types; for syntactic simplicity, we use "**class**" for all such definitions.

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.) The syntax for inner class names is the same as is used by the `javac` compiler. A good way to get an idea of the inner class names for a class is to compile the class and look at the filenames of the `.class` files that are produced.

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 binary name of the annotation name is required
    # (like the fully-qualified name, but with $ for inner classes).
    "**@**"*name* ["**(**" *annotation-field* ["**,**" *annotation-field*]+ "**)**"]

annotation-field ::=
    # In Java, if a single-field annotation has a field named

"`value`", then that field name may be elided in uses of the
annotation: "`@A(12)`" rather than "`@A(value=12)`".
The same convention holds in an annotation file.
name "=" *value*

class-definition ::=
"**class**" *name* "**:**" *annotation** "`\n`"
*typeparam-definition**
*bound-definition**
*extends-definition**
*implements-definition**
*field-definition**
*method-definition**

extends-definition ::=
Only type annotations are allowed.
"**extends**" "**:**" *annotation** "`\n`"
*type-argument-or-array-definition**

implements-definition ::=
Only type annotations are allowed.
The integer is the zero-based index of the implemented interface.
"**implements**" *integer* "**:**" *annotation** "`\n`"
*type-argument-or-array-definition**

field-definition ::=
The annotation on the "**field**" line is that of the field declaration,
while the annotation on the "**type**" line is that of outermost type.
source-insert-typecast-definition is described in Section 2.7.2.
Casts can only be inserted on a field if the field is initialized at declaration.
Otherwise, if the field is assigned in a method the cast can be inserted within a
method-definition rule.
"**field**" *name* "**:**" *annotation** "`\n`"
*type-annotations**
*source-insert-typecast-definition**

method-definition ::=
The method-key consists of the name followed by the signature
in JVML format, for example: "`foo([ILjava/lang/String;)V`".
Note that the signature is the erased signature of the method and does not
contain generic type information, but does contain the return type.
Using `javap -s` makes it easy to find the signature.
"`<init>`" and "`<clinit>`" are used to name instance and class initialization methods.
The annotation on the "**method**" line is that on the method, not on the return value.
The *source-*-definition* rules are described in Section 2.7.
"**method**" *method-key* "**:**" *annotation** "`\n`"
*typeparam-definition**
*bound-definition**
return-definition?
receiver-definition?
*parameter-definition**

*variable-definition**
*typecast-definition**
*instanceof-definition**
*new-definition**
*source-variable-definition**
*source-typecast-definition**
*source-instanceof-definition**
*source-new-definition**
*source-insert-typecast-definition**

type-annotations ::=
holds the type annotations, as opposed to the declaration annotations
"**type:**" *annotation** "\n"
*type-argument-or-array-definition**

type-argument-or-array-definition ::=
The integer list here contains the values of the "type_path" structure [Ern08].
"**inner-type**" *integer* "**,**" *integer* ["**,**" *integer* "**,**" *integer*]* "**:**" *annotation** "\n"

typeparam-definition ::=
The integer is the zero-based type parameter index.
"**typeparam**" *integer* "**:**" *annotation** "\n"

bound-definition ::=
The integers are respectively the parameter and bound indices of
the type parameter bound [Ern08].
"**bound**" *integer* "**&**" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

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

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

parameter-definition ::=
The integer is the index of the formal parameter in the method
(i.e., 0 is the first formal parameter. The receiver parameter is not index 0.
Use *receiver-definition* to annotate the receiver parameter.)
The annotation on the "**parameter**" line is that of the formal parameter declaration,
while the annotation on the "**type**" line is that of the outermost type of the parameter.
"**parameter**" *integer* "**:**" *annotation** "\n"
*type-annotations**

variable-definition ::=
The integers are respectively the index, start, and length
fields of the annotations on this variable [Ern08].
The annotation on the "**local**" line is that of the variable declaration,
while the annotation on the "**type**" line is that of the outermost type of the variable.

A source code index can be used instead. See *source-variable-definition* in Section 2.7.1.
"**local**" *integer* "#" *integer* "+" *integer* "**:**" *annotation** "\n"
*type-annotations**

typecast-definition ::=
The first integer is the offset field and the optional second integer
is the type index of an intersection type [Ern08].
The type index defaults to zero if not specified.
A source code index can be used instead. See *source-typecast-definition* in Section 2.7.1.
"**typecast**" "#" *integer* ["**,**" *integer*] "**:**" *annotation** "\n"
*type-argument-or-array-definition**

instanceof-definition ::=
The integer is the offset field of the annotation [Ern08].
A source code index can be used instead. See *source-instanceof-definition* in Section 2.7.1.
"**instanceof**" "#" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

new-definition ::=
The integer is the offset field of the annotation [Ern08].
A source code index can be used instead. See *source-new-definition* in Section 2.7.1.
"**new**" "#" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

2.6 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.

2.7 Source code locations

The Annotation File Utilities supports two additional methods to specify a location: source code indexes and AST paths. Unfortunately, these methods only allow insertion of annotations in source code, not `.class` files. Therefore, in order to insert annotations in both `.class` files and source code, locations must be redundantly specified as bytecode offsets and source code locations. This can be done in a single `.jaif` file or two seperate `.jaif` files. However, since bytecode offsets and source code locations are not used to add annotations to signatures, it is not necessary to include redundant information to insert annotations on signatures in both `.class` files and source code.

Source code indexes and AST paths are described below.

2.7.1 Source code indexes

For some tools it is easier to generate the index into the source code instead of the bytecode offset. As an experimental feature we support the following additional elements:

source-variable-definition ::=
The *name* is the identifier of the local variable.

The *integer* is the optional zero-based index of the
intended local variable within all local variables with the given *name*.
The default value for the index is zero.
The annotation on the "**local**" line is that of the variable declaration,
while the annotation on the "**type**" line is that of the outermost type of the variable.
"**local**" *name* ["*****" *integer*] "**:**" *annotation** "\n"
*type-annotations**

source-typecast-definition ::=
The first integer is the zero-based index of the typecast within the method and
the optional second integer is the type index of an intersection type [Ern08].
The type index defaults to zero if not specified.
"**typecast**" "*****" *integer* ["**,**" *integer*] "**:**" *annotation** "\n"
*type-argument-or-array-definition**

source-instanceof-definition ::=
The integer is the zero-based index of the instanceof within the method.
"**instanceof**" "*****" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

source-new-definition ::=
The integer is the zero-based index of the object or array creation within the method.
"**new**" "*****" *integer* "**:**" *annotation** "\n"
*type-argument-or-array-definition**

We use the star literal "*****" to distinguish source indexes from the bytecode offsets that are introduced using "#".

Source code indexes only include occurrences in the class that exactly matches the name of the enclosing *class-definition* rule. Specifically, occurrences in nested classes are not included. Use a new *class-definition* rule with the name of the nested class for source code insertions in a nested class.

2.7.2 AST paths

The Annotation File Utilities supports using a path through the AST (abstract syntax tree) to specify an arbitrary expression in source code to modify. Currently, AST paths can only be used to specify a location to insert a cast.

source-insert-typecast-definition ::=
ast-path is described below.
type is the un-annotated type to cast to.
"**insert-typecast**" *ast-path*"**:**" *annotation** *type* "\n"

An AST path represents a traversal through the AST nodes. AST paths are restricted for use in *field-definition*s and *method-definition*s. An AST path starts with the first element under the definition type. For methods this is `Block` and for fields this is `Variable`.

An AST path is composed of one or more AST entries, separated by commas. Each AST entry is composed of a tree kind, a child selector, and an optional argument. An example AST entry is:

```
Block.statement 1
```

The tree kind is `Block`, the child selector is `statement` and the argument is `1`.

The available tree kinds correspond to the Java AST tree nodes (from the package `com.sun.source.tree`), but with "Tree" removed from the name. For example, the class `com.sun.source.tree.BlockTree` is represented as `Block`. The child selectors correspond to the method names of the given Java AST tree node, with "get" removed from the beginning of the method name and the first letter lowercased. In cases where the child selector method returns a list, the method name is made singular and the AST entry also contains an argument to select the index of the list to take. For example, the method `com.sun.source.tree.BlockTree.getStatements()` is represented as `Block.statement` and requires an argument to select the statement to take.

The following is an example of an entire AST path:

```
Block.statement 1, Switch.case 1, Case.statement 0, ExpressionStatement.expression,
    MethodInvocation.argument 0
```

Since the above example starts with a `Block` it belongs in a *method-definition*. This AST path would select an expression that is in statement 1 of the method, case 1 of the switch statement, statement 0 of the case, and argument 0 of a method call (`ExpressionStatement` is just a wrapper around an expression that can also be a statement).

The following is an example of an annotation file with AST paths used to specify where to insert casts.

```
package p:
annotation @A:

class ASTPathExample:

field a:
    insert-typecast Variable.initializer, Binary.rightOperand: @A Integer

method m()V:
    insert-typecast Block.statement 0, Variable.initializer: @A Integer
    insert-typecast Block.statement 1, Switch.case 1, Case.statement 0,
        ExpressionStatement.expression, MethodInvocation.argument 0: @A Integer
```

And the matching source code:

```
package p;

public class ASTPathExample {

    private int a = 12 + 13;

    public void m() {
        int x = 1;
        switch (x + 2) {
            case 1:
                System.out.println(1);
                break;
            case 2:
                System.out.println(2 + x);
                break;
            default:
                System.out.println(-1);
        }
    }
}
```

The following is the ouput, with the casts inserted.

```
package p;
import p.A;

public class ASTPathExample {

    private int a = 12 + ((@A Integer) (13));

    public void m() {
        int x = ((@A Integer) (1));
        switch (x + 2) {
            case 1:
                System.out.println(1);
                break;
            case 2:
                System.out.println(((@A Integer) (2 + x)));
                break;
            default:
                System.out.println(-1);
        }
    }
}
```

Note that two additional sets of parentheses are always added for each cast insertion: one set around the expression to be cast, and the other set around the cast and the expression. For example, a cast insertion looks like this:

((*cast type*) (*original expression*))

This is to limit the cast to the entire expression, but nothing more. These parentheses may be unnecessary and can be removed by the programmer afterward. Also note that a cast can be inserted on any expression, not just the lowest expression in the AST. For example, a cast could be inserted on the expression `i + j`, the identifier `i`, and/or the identifier `j`.

To help create correct AST paths it may be useful to view the AST of a class. The Checker Framework has a processor to do this. The following command will output indented AST nodes for the entire input program.

```
javac -processor checkers.util.debug.TreeDebug ASTPathExample.java
```

The following is the grammar for AST paths.

ast-path ::=
    ast-entry ["," *ast-entry*]+

ast-entry ::=
    annotated-type
    | *array-access*
    | *array-type*
    | *assert*
    | *assignment*
    | *binary*
    | *block*

| *case*
| *catch*
| *compound-assignment*
| *conditional-expression*
| *do-while-loop*
| *enhanced-for-loop*
| *expression-statement*
| *for-loop*
| *if*
| *instance-of*
| *labeled-statement*
| *lambda-expression*
| *member-reference*
| *member-select*
| *method-invocation*
| *new-array*
| *new-class*
| *parameterized-type*
| *parenthesized*
| *return*
| *switch*
| *synchronized*
| *throw*
| *try*
| *type-cast*
| *unary*
| *union-type*
| *variable-type*
| *while-loop*
| *wildcard-tree*

annotated-type :: =
“**AnnotatedType**” “**.**” ((“**annotation**” *integer*) | “**underlyingType**”)

array-access ::=
“**ArrayAccess**” “**.**” (“**expression**” | “**index**”)

array-type ::=
“**ArrayType**” “**.**” “**type**”

assert ::=
“**Assert**” “**.**” (“**condition**” | “**detail**”)

assignment ::=
“**Assignment**” “**.**” (“**variable**” | “**expression**”)

binary ::=
“**Binary**” “**.**” (“**leftOperand**” | “**rightOperand**”)

block ::=
“**Block**” “**.**” “**statement**” *integer*

case ::=
“**Case**” “**.**” (“**expression**” | (“**statement**” *integer*))

catch ::=
“**Catch**” “**.**” (“**parameter**” | “**block**”)

compound-assignment ::=
“**CompountAssignment**” “**.**” (“**variable**” | “**expression**”)

conditional-expression ::=
“**ConditionalExpression**” “**.**” (“**condition**” | “**trueExpression**” | “**falseExpression**”)

do-while-loop ::=
“**DoWhileLoop**” “**.**” (“**condition**” | “**statement**”)

enhanced-for-loop ::=
“**EnhancedForLoop**” “**.**” (“**variable**” | “**expression**” | “**statement**”)

expression-statement ::=
“**ExpressionStatement**” “**.**” “**expression**”

for-loop ::=
“**ForLoop**” “**.**” ((“**initializer**” *integer*) | “**condition**” | (“**update**” *integer*) | “**statement**”)

if ::=
“**If**” “**.**” (“**condition**” | “**thenStatement**” | “**elseStatement**”)

instance-of ::=
“**InstanceOf**” “**.**” (“**expression**” | “**type**”)

labeled-statement ::=
“**LabeledStatement**” “**.**” “**statement**”

lambda-expression ::=
“**LambdaExpression**” “**.**” ((“**parameter**” *integer*) | “**body**”)

member-reference ::=
“**MemberReference**” “**.**” (“**qualifierExpression**” | (“**typeArgument**” *integer*))

member-select ::=
“**MemberSelect**” “**.**” “**expression**”

method-invocation ::=
“**MethodInvocation**” “**.**” ((“**typeArgument**” *integer*) | “**methodSelect**”
| (“**argument**” *integer*))

new-array ::=
“**NewArray**” “**.**” (“**type**” | (“**dimension**” | “**initializer**”) *integer*)

new-class ::=

“**NewClass**” “**.**” (“**enclosingExpression**” | (“**typeArgument**” *integer*) | “**identifier**” | (“**argument**” *integer*) | “**classBody**”)

parameterized-type ::=
“**ParameterizedType**” “**.**” (“**type**” | (“**typeArgument**” *integer*))

parenthesized ::=
“**Parenthesized**” “**.**” “**expression**”

return ::=
“**Return**” “**.**” “**expression**”

switch ::=
“**Switch**” “**.**” (“**expression**” | (“**case**” *integer*))

synchronized ::=
“**Synchronized**” “**.**” (“**expression**” | “**block**”)

throw ::=
“**Throw**” “**.**” “**expression**”

try ::=
“**Try**” “**.**” (“**block**” | (“**catch**” *integer*) | “**finallyBlock**” | (“**resource**” *integer*))

type-cast ::=
“**TypeCast**” “**.**” (“**type**” | “**expression**”)

unary ::=
“**Unary**” “**.**” “**expression**”

union-type ::=
“**UnionType**” “**.**” “**typeAlternative**” *integer*

variable ::=
“**Variable**” “**.**” (“**type**” | “**initializer**”)

while-loop ::=
“**WhileLoop**” “**.**” (“**condition**” | “**statement**”)

wildcard ::=
“**Wildcard**” “**.**” “**bound**”

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
import java.util.*;

public @A(12) class Foo {

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

    public Foo(@D("spam") Foo this, @B List<@C String> a) {
        @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>(
      Ljava/util/List;)V:
        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, annotations, 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 binary name of the enumeration class, such as `enum java.lang.Thread$State`.
- Annotation: `@` followed by the binary name of the annotation type.
- 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 binary 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. Also as in Java, the braces may be omitted if the array has only one element. Examples: `{1}`, `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
```

[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.

```
    enum p1.DebugCategory[] defaultDebugCategories
    @p1.CommitInfo lastCommit

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

[Ern08] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[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.