

Annotation File Format Specification

<http://types.cs.washington.edu/annotation-file-utilities/>
November 26, 2014

Contents

1	Purpose: External storage of annotations	1
2	Grammar	2
2.1	Grammar conventions	2
2.2	Java file grammar	3
2.2.1	Package definitions	3
2.2.2	Class definitions	3
2.2.3	Field definitions	5
2.2.4	Method definitions	5
2.3	Bytecode Locations	6
2.3.1	Bytecode offsets	7
2.3.2	Source code indexes	7
2.3.3	Code locations grammar	7
2.3.4	AST paths	8
2.4	Annotations	14
2.4.1	Annotation definitions	14
2.4.2	Annotation uses	14
3	Example	16
4	Types and values	16
5	Alternative formats	18
6	Design and Implementation Details	19
6.1	scene-lib	19
6.2	asmx	20
6.2.1	Bytecode Insertion	20
6.2.2	Bytecode Extraction	20
6.3	annotation-file-utilities	21

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 move 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 5 declaration annotations and also the type annotations that are introduced by Java SE 8. The file format provides a simple syntax to represent the structure of a Java program. For annotations in method bodies of `.class` files the annotation file closely follows section “Class File Format Extensions” of the JSR 308 design document [Ern11], which explains how the annotations are stored in the `.class` file. In that sense, the current design is extremely low-level, and users probably would not want to write the files by hand (but they 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. For `.java` source files, the file format provides a separate, higher-level syntax for annotations in method bodies.

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

2 Grammar

This section describes the annotation file format in detail by presenting it in the form of a grammar. Section 2.1 details the conventions of the grammar. Section 2.2 shows how to represent the basic structure of a Java program (classes, methods, etc.) in an annotation file. Section 2.4 shows how to add annotations to an annotation file.

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 newlines are supported.

2.2 Java file grammar

This section shows how to represent the basic structure of a Java program (classes, methods, etc.) in an annotation file. For Java elements that can contain annotations, this section will reference grammar productions contained in Section 2.4, which describes how annotations are used in an annotation file.

An annotation file has the same basic structure as a Java program. That is, there are packages, classes, fields and methods.

The annotation file may omit certain program elements — for instance, it may mention only some of the packages in a 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.

2.2.1 Package definitions

At the root of an annotation file is one or more package definitions. A package definition describes a package containing a list of annotation definitions and classes. A package definition also contains any annotations on the package (such as those from a `package-info.java` file).

```
annotation-file ::=  
    package-definition+
```

```
package-definition ::=  
    “package” ( “:” ) | ( name “:” decl-annotation* ) “\n”  
    ( annotation-definition | class-definition ) *
```

Use a package line of `package:` for the default package. Note that annotations on the default package are not allowed.

2.2.2 Class definitions

A class definition describes the annotations present on a class declaration, as well fields and methods of the class. It is organized according to the hierarchy of fields and methods in the class. Note that we use *class-definition* also for interfaces, enums, and annotation types (to specify annotations in an existing annotation type, not to be confused with *annotation-definitions* described in Section 2.4.1, which defines annotations to be used throughout an annotation file); 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 or 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.

```
class-definition ::=  
    “class” name “:” decl-annotation* “\n”  
    typeparam-definition*  
    typeparam-bound*  
    extends*  
    implements*  
    field-definition*  
    staticinit*  
    instanceinit*  
    method-definition*
```

Annotations on the “**class**” line are annotations on the class declaration, not the class name.

Type parameter definitions The *typeparam-definition* production defines annotations on the declaration of a type parameter, such as on K and T in

```
public class Class<K> {
    public <T> void m() {
        ...
    }
}
```

```
typeparam-definition ::=
    # The integer is the zero-based type parameter index.
    “typeparam” integer “:” type-annotation* “\n”
```

Type Parameter Bounds The *typeparam-bound* production defines annotations on a bound of a type variable declaration, such as on Number and Date in

```
public class Class<K extends Number> {
    public <T extends Date> void m() {
        ...
    }
}
```

```
typeparam-bound ::=
    # The integers are respectively the parameter and bound indexes of
    # the type parameter bound [Ern11].
    “bound” integer “&” integer “:” type-annotation* “\n”
    compound-type*
```

Implements and extends The *extends* and *implements* productions define annotations on the names of classes a class extends or implements.

```
extends ::=
    “extends” “:” type-annotation* “\n”
    compound-type*
```

```
implements ::=
    # The integer is the zero-based index of the implemented interface.
    “implements” integer “:” type-annotation* “\n”
    compound-type*
```

Static and instance initializers The *staticinit* and *instanceinit* productions define annotations on code within static or instance initializer blocks.

```
staticinit ::=
    # The integer is the zero-based index of the implemented interface.
    “staticinit” “*” integer “:” “\n”
    compound-type*
```

```
instanceinit ::=
    # The integer is the zero-based index of the implemented interface.
    “instanceinit” “*” integer “:” “\n”
    compound-type*
```

2.2.3 Field definitions

A field definition can have annotations on the declaration, the type of the field, or — if in source code — the field’s initialization.

```
field-definition ::=  
  “field” name “:” decl-annotation* “\n”  
  type-annotations*  
  expression-annotations*
```

Annotations on the “**field**” line are on the field declaration, not the type of the field.

The *expression-annotations* production specifies annotations on the initialization expression of a field. If a field is initialized at declaration then in bytecode the initialization is moved to the constructor when the class is compiled. Therefore for bytecode, annotations on the initialization expression go in the constructor (see Section 2.2.4), rather than the field definition. Source code annotations for the field initialization expression are valid on the field definition.

2.2.4 Method definitions

A method definition can have annotations on the method declaration, in the method header (return type, parameters, etc.), as well as the method body.

```
method-definition ::=  
  “method” method-key “:” decl-annotation* “\n”  
  typeparam-definition*  
  typeparam-bound*  
  return-type?  
  receiver-definition?  
  parameter-definition*  
  variable-definition*  
  expression-annotations*
```

The annotations on the “**method**” line are on the method declaration, not on the return value. The *method-key* consists of the name followed by the signature in JVMML format. For example, the following method

```
boolean foo(int[] i, String s) {  
  ...  
}
```

has the *method-key*:

```
foo([ILjava/lang/String;)Z
```

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. The method keys “<init>” and “<clinit>” are used to name instance (constructor) and class (static) initialization methods.

Return type A return type defines the annotations on the return type of a method declaration. It is also used for the result of a constructor.

```
return-type ::=  
  “return:” type-annotation* “\n”  
  compound-type*
```

Receiver definition A receiver definition defines the annotations on the type of the receiver parameter in a method declaration. A method receiver is the implicit formal parameter, `this`, used in non-static methods. For source code insertion, the receiver parameter will be inserted if it does not already exist.

Only inner classes have a receiver. A top-level constructor does not have a receiver, though it does have a result. The type of a constructor result is represented as a return type.

```
receiver-definition ::=  
    "receiver:" type-annotation* "\n"  
    compound-type*
```

Parameter definition A parameter definition defines the annotations on a method parameter declaration and the type of a method parameter.

```
parameter-definition ::=  
    # The integer is the zero-based index of the formal parameter in the method.  
    "parameter" integer ":" decl-annotation* "\n"  
    type-annotations*
```

The annotations on the `"parameter"` line are on the formal parameter declaration, not on the type of the parameter. A parameter index of 0 is the first formal parameter. The receiver parameter is not index 0. Use the *receiver-definition* production to annotate the receiver parameter.

2.3 Bytecode Locations

Certain elements in the body of a method or the initialization expression of a field can be annotated. The *expression-annotations* rule describes the annotations that can be added to a method body or a field initialization expression:

```
expression-annotations ::=  
    typecast*  
    instanceof*  
    new*  
    source-insert-typecast*  
    source-insert-annotation*
```

Additionally, a variable declaration in a method body can be annotated with the *variable-definition* rule, which appears below.

Because of the differences between Java source code and `.class` files, the syntax for specifying code locations is different for `.class` files and source code. For `.class` files we use a syntax called "bytecode offsets". For source code we use a different syntax called "source code indexes". These are both described below.

If you wish to be able to insert a given code annotation in both a `.class` file and a source code file, the annotation file must redundantly specify the annotation's bytecode offset and source code index. This can be done in a single `.jaif` file or two separate `.jaif` files. It is not necessary to include redundant information to insert annotations on signatures in both `.class` files and source code.

Additionally, a new typecast with annotations (rather than an annotation added to an existing typecast) can be inserted into source code. This uses a third syntax that is described below under "AST paths". A second way to insert a typecast is by specifying just an annotation, not a full typecast (`insert-annotation` instead of `insert-typecast`). In this case, the source annotation insertion tool generates a full typecast if Java syntax requires one.

2.3.1 Bytecode offsets

For locations in bytecode, 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. The `javap -v` command is an effective technique to discover bytecode offsets. Non-expression annotations such as those on methods, fields, classes, etc., do not use a bytecode offset.

2.3.2 Source code indexes

For locations in source code, the annotation file indicates the kind of expression, plus a zero-based index to indicate which occurrence of that kind of expression. For example,

```
public void method() {
    Object o1 = new @A String();
    String s = (@B String) o1;
    Object o2 = new @C Integer(0);
    Integer i = (@D Integer) o2;
}
```

@A is on new, index 0. @B is on typecast, index 0. @C is on new, index 1. @D is on typecast, index 1.

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.

2.3.3 Code locations grammar

For each kind of expression, the grammar contains a separate location rule. This location rule contains the bytecode offset syntax followed by the source code index syntax.

The grammar uses “#” for bytecode offsets and “*” for source code indexes.

variable-location ::=

```
# Bytecode offset: the integers are respectively the index, start, and length
# fields of the annotations on this variable [Ern11].
(integer “#” integer “+” integer)
# Source code index: 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.
| (name [“*” integer])
```

variable-definition ::=

```
# The annotations on the “local” line are on the variable declaration,
# not the type of the variable.
“local” variable-location “:” decl-annotation* “\n”
type-annotations*
```

typecast-location ::=

```
# Bytecode offset: the first integer is the offset field and the optional
# second integer is the type index of an intersection type [Ern11].
# The type index defaults to zero if not specified.
(“#” integer [ “,” integer ])
```

```

# Source code index: 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 [Ern11]. The type index defaults to zero if not specified.
| (“*” integer [ “,” integer ])

```

```

typecast ::=
  “typecast” typecast-location “:” type-annotation* “\n”
  compound-type*

```

```

instanceof-location ::=
  # Bytecode offset: the integer is the offset field of the annotation [Ern11].
  (“#” integer)
  # Source code index: the integer is the zero-based index of the instanceof within the method.
  | (“*” integer)

```

```

instanceof ::=
  “instanceof” instanceof-location “:” type-annotation* “\n”
  compound-type*

```

```

new-location ::=
  # Bytecode offset: the integer is the offset field of the annotation [Ern11].
  (“#” integer)
  # Source code index: the integer is the zero-based index of the object or array
  # creation within the method.
  | (“*” integer)

```

```

new ::=
  “new” new-location “:” type-annotation* “\n”
  compound-type*

```

2.3.4 AST paths

A path through the AST (abstract syntax tree) specifies an arbitrary expression in source code to modify. AST paths can be used in the `.jaif` file to specify a location to insert either a bare annotation (“**insert-annotation**”) or a cast (“**insert-typecast**”).

For a cast insertion, the `.jaif` file specifies the type to cast to. The annotations on the “**insert-typecast**” line will be inserted on the outermost type of the type to cast to. If the type to cast to is a compound type then annotations on parts of the compound type are specified with the *compound-type* rule. If there are no annotations on the “**insert-typecast**” line then a cast with no annotations will be inserted or, if compound type annotations are specified, a cast with annotations only on the compound types will be inserted.

Note that the type specified on the “**insert-typecast**” line cannot contain any qualified type names. For example, use `Entry<String, Object>` instead of `Map.Entry<java.lang.String, java.lang.Object>`.

```

source-insert-typecast ::=
  # ast-path is described below.
  # type is the un-annotated type to cast to.
  “insert-typecast” ast-path “:” type-annotation* type “\n”
  compound-type*

```

An AST path represents a traversal through the AST. AST paths can only be used in *field-definitions* and *method-definitions*. An AST path starts with the first element under the definition. 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 output, 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);
        }
    }
}

```

Using `insert-annotation` instead of `insert-typecast` yields almost the same result — it also inserts a cast. The sole difference is the inability to specify the type in the cast expression. If you use `insert-annotation`, then the annotation inserter infers the type, which is `int` in this case.

Note that a cast can be inserted on any expression, not just the deepest 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 org.checkerframework.common.util.debug.TreeDebug ASTPathExample.java
```

The following is the grammar for AST paths.

```

ast-path ::=
    ast-entry [ “,” ast-entry ]+

```

```

ast-entry ::=
    annotated-type
    | annotation
    | 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*
- | *intersection-type*
- | *labeled-statement*
- | *lambda-expression*
- | *member-reference*
- | *member-select*
- | *method-invocation*
- | *new-array*
- | *new-class*
- | *parameterized-type*
- | *parenthesized*
- | *return*
- | *switch*
- | *synchronized*
- | *throw*
- | *try*
- | *type-cast*
- | *type-parameter*
- | *unary*
- | *union-type*
- | *variable-type*
- | *while-loop*
- | *wildcard-tree*

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

annotation ::= =
 “Annotation” “.” (“type” | “argument” *integer*)

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 ::=
 “CompoundAssignment” “.” (“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”)

intersection-type ::=
 “IntersectionType” “.” “bound” *integer*

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”)

type-parameter ::=
 “TypeParameter” “.” “bound” *integer*

unary ::=
 “Unary” “.” “expression”

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

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

```
while-loop ::=
    "WhileLoop" "." ( "condition" | "statement" )
```

```
wildcard ::=
    "Wildcard" "." "bound"
```

2.4 Annotations

This section describes the details of how annotations are defined, how annotations are used, and the different kinds of annotations in an annotation file.

2.4.1 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. Any annotation that is used in an annotation file must be defined before use. (This requirement makes it impossible to define, in an annotation file, an annotation that is meta-annotated with itself.) The two exceptions to this rule are the `@java.lang.annotation.Target` and `@java.lang.annotation.Retention` meta-annotations. These meta-annotations are often used in annotation definitions so for ease of use are they not required to be defined themselves. 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.

Note that these annotation definitions should not be confused with the `@interface` syntax used in a Java source file to declare an annotation. An annotation definition in an annotation file is only used internally. An annotation definition in an annotation file will often mirror an `@interface` annotation declaration in a Java source file in order to use that annotation in an annotation file.

```
annotation-definition ::=
    # The decl-annotations are the meta-annotations on this annotation.
    "annotation" "@" name ":" decl-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"
```

2.4.2 Annotation uses

Java SE 8 has two kinds of annotations: “declaration annotations” and “type annotations”. Declaration annotations can be written only on method formal parameters and the declarations of packages, classes, methods, fields, and local variables. Type annotations can be written on any use of a type, and on type parameter declarations. Type annotations must be meta-annotated with `ElementType.TYPE_USE` and/or `ElementType.TYPE_PARAMETER`. These meta-annotations are described in more detail in the JSR 308 specification [Ern11].

The previous rules have used two productions for annotation uses in an annotation file: *decl-annotation* and *type-annotation*. The *decl-annotation* and *type-annotation* productions use the same syntax to specify an

annotation. These two different rules exist only to show which type of annotation is valid in a given location. A declaration annotation must be used where the *decl-annotation* production is used and a type annotation must be used where the *type-annotation* production is used.

The syntax for an annotation is the same as in a Java source file.

decl-annotation ::=

```
# annotation must be a declaration annotation.
annotation
```

type-annotation ::=

```
# annotation must be a type annotation.
annotation
```

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 name of the annotation name is required.
"@ 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
```

Certain Java elements allow both declaration and type annotations (for example, formal method parameters). For these elements, the *type-annotations* rule is used to differentiate between the declaration annotations and the type annotations.

type-annotations ::=

```
# holds the type annotations, as opposed to the declaration annotations.
"type:" type-annotation* "\n"
compound-type*
```

Compound type annotations A compound type is a parameterized, wildcard, array, or nested type. Annotations may be on any type in a compound type. In order to specify the location of an annotation within a compound type we use a "type path". A type path is composed one or more pairs of type kind and type argument index.

type-kind ::=

```
"0" # annotation is deeper in this array type
| "1" # annotation is deeper in this nested type
| "2" # annotation is on the bound of this wildcard type argument
| "3" # annotation is on the i'th type argument of this parameterized type
```

type-path ::=

```
# The integer is the type argument index.
type-kind "," integer [ "," type-kind "," integer ]*
```

compound-type ::=

```
"inner-type" type-path ":" annotation* "\n"
```

```

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.

The type argument index used in the *type-path* rule must be “0” unless the *type-kind* is “3”. In this case, the type argument index selects which type argument of a parameterized type to use.

Type paths are explained in more detail, with many examples to ease understanding, in Section 3.4 of the JSR 308 Specification.¹

3 Example

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

4 Types and values

The Java language permits several types for annotation fields: primitives, `Strings`, `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.²

¹http://types.cs.washington.edu/jsr308/specification/java-annotation-design.html#class-file:ext:type_path

²There 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.

<pre> 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 </pre>	<pre> 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 </pre>
--	--

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

The Checker Framework [DDE⁺11, Che] uses two additional formats for annotations. The first format is called “stub files.” A stub file is similar to the `.spec/.jml` files described in the previous paragraph. It uses Java syntax, only allows annotations on method headers and does not require method bodies. A stub file is used to add annotations to method headers of existing Java classes. For example, the Checker Framework uses stub files to add annotations to method headers of libraries (such as the JDK) without modifying the source code or bytecode of the library. A single stub file can contain multiple packages and classes. This format only allows annotations on method headers, not class headers, fields, and method bodies like in a `.jaif` file. Further, stub files are only used by the Checker Framework at run time, they cannot be used to insert annotations into a source or classfile.

The Checker Framework also uses a format called an “annotated JDK.” The annotated JDK is a `.jar` file containing the JDK with annotations. It is created with the Annotation File Utilities, but the annotations are stored in a format similar to a stub file, instead of in a `.jaif` file. The annotated JDK starts with a source file for each file in the JDK to be annotated. Like a stub file, each source file only contains method headers with annotations. The annotated JDK also supports annotations in the class header. To build the annotated JDK `.jar` file, the source files are compiled, then the `extract-annotations` script is run on them to generate a `.jaif` file for each source file. The `insert-annotations` script then inserts the annotations contained in each `.jaif` file into the corresponding JDK class file. These are then packaged up into a single `.jar` file. Like a stub files, the annotated JDK is easier to read and write since it uses Java syntax. However, the annotated JDK requires a different file for each original Java source file. It does not allow annotations on fields and in method bodies. The annotated JDK also only contains annotations in the JDK and not other Java files.

6 Design and Implementation Details

This section describes some high level-design and implementation details of the Annotation File Utilities, including the different components of the Annotation File Utilities and how they fit together. It is intended for someone who is beginning work on the Annotation File Utilities or is curious about how the Annotation File Utilities work.

The Annotation File Utilities is composed of three sub-projects: `scene-lib`, `asmx`, and `annotation-file-utilities`. The `scene-lib` sub-project represents a `.jaif` file. The `asmx` sub-project inserts and extracts annotations to/from bytecode. The `annotation-file-utilities` sub-project inserts annotations into source code.

6.1 `scene-lib`

`scene-lib` is an interface to a `.jaif` file. It reads in and writes out `.jaif` files and provides an internal representation of a `.jaif` file to access and manipulate.

Internally, a `.jaif` file is represented by the `scene-lib/src/annotations/el/AScene.java` class. The `AScene` class (or “annotated scene”) roughly parallels the root of an abstract syntax tree. An `AScene` has a number of classes (`AClass`) as children. Each class has a number of methods (`AMethod`), fields (`AElement`), etc. as children. All of these classes are related in a type hierarchy (Figure 3). Note that this type hierarchy does not have a traditional “is-a” relationship between its classes. For example, it is not valid to say that a method is-a block, as the type hierarchy implies. In fact, it makes more sense to view the type hierarchy upside down, like so: a method contains a block, a block contains multiple expressions, and an expression contains an element. (Perhaps object composition would have been a better pattern to follow here.)

Each class in the type hierarchy has one or more fields to hold annotations for the different components of the class. For example, the `AMethod` class has the following fields: `bounds`, `return type`, `receiver parameters`, and `throws clause`. Each of these fields holds the annotations stored on that part of the method. For details on the remainder of the classes in the type hierarchy, and their respective fields, see the documentation for each file in `scene-lib/src/annotations/el/`.

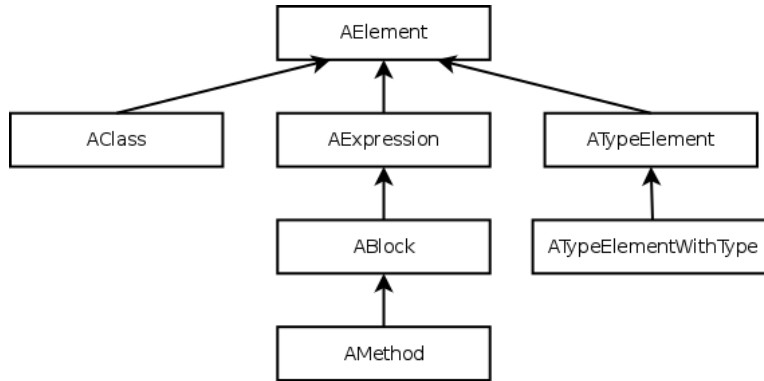


Figure 3: The type hierarchy of the `scene-lib` classes used to represent a `.jaif` file.

An `AScene` instance can be created in two ways. An empty `AScene` can be created by calling the `AScene` constructor, or an `AScene` can be created by parsing an existing `.jaif` file. Once an `AScene` is created, annotations can be added to it by adding them to the correct fields of the children. An `AScene` can also be output to create a new `.jaif` file.

6.2 `asmx`

The `asmx` sub-project is used to insert and extract annotations from bytecode. `asmx` is based on an old version (2.2.2) of the ASM Framework [ASM]. It has been modified to allow it to read and write JSR 308 [Ern11] annotations in bytecode. However, it is far behind the current ASM version (4.1). Experimental support for type annotations has been added to ASM for a future ASM 5 release. When ASM 5 is released, we should discard the custom `asmx` in the Annotation File Utilities and adapt the Annotation File Utilities to use the official, supported version of ASM.

6.2.1 Bytecode Insertion

Annotations can be inserted into bytecode by executing the `annotation-file-utilities/scripts/insert-annotations` script. This script takes one or more `(class name, .jaif file)` pairs as arguments. The annotations specified in the `.jaif` file are inserted into the classfile directly before the `.jaif` file in the argument list.

First, each `.jaif` file is parsed into an `AScene` (as described in Section 6.1). Then, `asmx/src/org/objectweb/asm/ClassReader.java` parses the classfile. As it is parsing the classfile, it passes the parsed bytecode off to the `scene-lib/src/annotations/io/classfile/ClassAnnotationSceneWriter.java` class. This class has a reference to the `AScene` parsed from the `.jaif` file. As this class receives the parsed bytecode it inserts the relevant annotations from the `AScene` in the bytecode and then writes the bytecode back out.

6.2.2 Bytecode Extraction

Annotations can be extracted from bytecode by executing the `annotation-file-utilities/scripts/extract-annotations` script. This script takes one or more class names as arguments and outputs the annotations found in those classes to `.jaif` files.

First, an empty `AScene` is constructed to store the annotations. `asmx/src/org/objectweb/asm/ClassReader.java` parses the classfile and passes the parsed bytecode off to the `scene-lib/src/annotations/io/classfile/ClassAnnotationSceneReader.java` class. This class filters out the annotations in the bytecode and adds them to the correct part of the `AScene`. After this, the `AScene` is output to a `.jaif` file.

6.3 annotation-file-utilities

The `annotation-file-utilities` sub-project inserts annotations into source code. It can be run by executing the `annotation-file-utilities/scripts/insert-annotations-to-source` script. The script takes one or more `.jaif` files, followed by one or more `.java` source files as arguments. The annotations in the `.jaif` files are inserted into the `.java` source files.

First, an instance of `annotation-file-utilities/src/annotator/specification/IndexFileSpecification.java` is created. Its `parse` method parses the `.jaif` file into an `AScene` (as described in Section 6.1). The `parse` method calls the `parseScene` method, which traverses through the `AScene` and creates an `annotation-file-utilities/src/annotator/specification/CriterionList.java`. A `CriterionList` identifies a unique AST node that is the location of an insertion. It contains objects that implement the `annotation-file-utilities/src/annotator/find/Criterion.java` interface. Each `Criterion` has an `isSatisfiedBy` method — a predicate that takes an AST node and returns `true` if the AST node satisfies the `Criterion` and `false` otherwise. To determine if a given node matches a `CriterionList`, the node is passed to all of the `Criteria` in the `CriteriaList`. If every `Criterion` returns `true` then it is match. If one or more `Criteria` return `false` then it is not a match. The various `Criterion` classes are in the `annotation-file-utilities/src/annotator/find/` directory. For example, take the following source code:

```
package afu.example;

public class Test {
    public void m(boolean b, int i) {
        // ...
    }
}
```

The `CriterionList` to specify the location of the `i` parameter contains the following `Criteria`s:

- `InPackageCriterion("afu.example")`
- `InClassCriterion("Test")`
- `InMethodCriterion("m(ZI)V")`
- `ParamCriterion(1)`

After this `CriterionList` is built up an `annotation-file-utilities/src/annotator/find/Insertion.java` is created. An `Insertion` stores an `annotation-file-utilities/src/annotator/find/Criteria.java` (which is created from a `CriterionList`) and the text to be inserted. All of these `Insertions` are then added to a list. The Java compiler then is called to parse the Java source into an abstract syntax tree. This is followed by a call to the `getPositions` method of `annotation-file-utilities/src/annotator/find/TreeFinder.java`, which scans through each node of the abstract syntax trees. For each node, it runs through the `Criteria` for each un-matched `Insertion`. If at least one of the `Criteria` does not match, then this is not the correct place for the `Insertion` and the `Insertion` will be checked at the remaining nodes of the tree. If all of the `Criteria` match, then this node is the correct place for the `Insertion`. It is removed from the list of un-matched `Insertions` and the position where to insert the `Insertion` is determined. This position is the integer index in the file where the `Insertion` should be inserted. After the positions are found for all of the `Insertions`, the `Insertion` text is inserted into the file. This happens backwards, with `Insertions` at the end of the file (i.e. with higher positions) being inserted first. If `Insertions` were instead inserted from the beginning of the file then a single `Insertion` would invalidate all of the positions for the following `Insertions`.

If there are remaining `Insertions` that were not matched to a node in the abstract syntax tree then an error message is displayed.

References

- [ASM] ASM website. <http://asm.ow2.org/>.
- [Che] Checker Framework website. <http://types.cs.washington.edu/checker-framework/>.
- [DDE⁺11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.
- [Ern11] Michael D. Ernst. Type Annotations specification (JSR 308). <http://types.cs.washington.edu/jsr308/>, October 2011.
- [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.