Annotation File Format Specification

https://checkerframework.org/annotation-file-utilities/

November 2, 2026

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 . . . 9
- 2.4 Annotations . . . 15
  - 2.4.1 Annotation definitions . . . 15
  - 2.4.2 Annotation uses . . . 15

3 Example **17**

4 Types and values **17**

5 Alternative formats **19**

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 (`https://checkerframework.org/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 were 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 [Ern13], 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, "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 a 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 are 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 as 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-definition*s 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 an annotation 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() {
        ...
    }
}
```

or on the type parameters on the left-hand side of a member reference, as on `String` in `List<String>::size`.

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

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 [Ern13].
    "**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`. (Note: For interface declarations, *implements* rather than *extends* defines annotations on the names of extended interfaces.)

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 static initializer block.
    "**staticinit**" "*****" *integer* "**:**" "\n"
    *compound-type**
instanceinit ::=
    # The integer is the zero-based index of the instance initializer block.
    "**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 out of the field definition when the class is compiled: the initialization of an instance field is moved to each constructor, and the initialization of a static field is moved to the class initialization method `<clinit>()V`. Therefore, for bytecode, annotations on the initialization expression of an instance field go in each constructor, and annotations on the initialization expression of a static field go in `<clinit>()V` (see Section 2.2.4), rather than in the field definition. (The *staticinit* and *instanceinit* productions do not apply: they represent only explicit initializer blocks, not field initializers.) 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 signature (return type, parameters, etc.), and in 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 simple name followed by a method descriptor, which is the signature in JVML format (see JVMS §4.3.3, `https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.3.3`). 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. (The name of the constructor—that is, the final element of the class name—can be used in place of "`<init>`".) For both instance and class initializers, the "return type" part of the signature should be `V` (for `void`).

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 constructors of 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 formal parameter definition defines the annotations on a method formal parameter declaration and the type of a method formal parameter, but *not* the receiver formal 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**
    *call**
    *reference**
    *lambda**
    *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 [Ern13].
    (*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 [Ern13].
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 [Ern13]. 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 [Ern13].
("#" *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 [Ern13].
("#" *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**
call-location ::=
Bytecode offset: the integer is the offset field of the annotation [Ern13].
("#" *integer*)
Source code index: the integer is the zero-based index of the method call
within the field or method definition.
| ("*****" *integer*)

call ::=
"**call**" *call-location* "**:**" "`\n`"
*typearg-definition**

reference-location ::=
Bytecode offset: the integer is the offset field of the annotation [Ern13].

("#" *integer*)
Source code index: the integer is the zero-based index of the member
reference [Ern13].
| ("***** " *integer*)

reference ::=
"**reference**" *reference-location* "**:**" *type-annotation** "`\n`"
*compound-type**
*typearg-definition**

lambda-location ::=
Bytecode offset: the integer is the offset field of the annotation [Ern13].
("#" *integer*)
Source code index: the integer is the zero-based index of the lambda
expression [Ern13].
| ("***** " *integer*)

lambda ::=
"**lambda**" *lambda-location* "**:**" "`\n`"
*parameter-definition**
*variable-definition**
*expression-annotations**
typearg-definition ::=
The integer is the zero-based type argument index.
"**typearg**" *integer* "**:**" *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 unannotated 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-definition*s and *method-definition*s. 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*
| *while-loop*
| *wildcard*

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 they are 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-annotation*s 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 [Ern13].

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 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 of 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`"

The type argument index used in the *type-path* rule must be "**0**" unless the *type-kind* is "**3**". In the latter 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]

[1] `https://checkerframework.org/jsr308/specification/java-annotation-design.html#class-file:ext:type_path`

```
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.

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, `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.[2]

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`.

[2]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.

```
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.

- 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] defines a format called "stub files". A stub file is similar to the `.spec`/`.jml` files described in the previous paragraph. It uses Java syntax, allows annotations on class declarations, fields, and method signatures, and omits method bodies. A stub file is used to add annotations to existing Java classes. For example, the Checker Framework uses stub files to add annotations to 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. Unlike a `.jaif` file, a stub file cannot express annotations within method bodies. Further, stub files are only read by the Checker Framework during type checking; they cannot be used to insert annotations into a source file or classfile.

Eclipse defines its own file format for external nullness annotations: `https://wiki.eclipse.org/JDT_Core/Null_Analysis/External_Annotations#File_format`. It works only for nullness annotations. It is more compact but less readable than the Annotation File Format. It is intended for tool use, not for editing by ordinary users, who are expected to interact with it via the Eclipse GUI.

References

[Che] Checker Framework website. `https://checkerframework.org/`.

[DDE+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 2011.

[Ern13] Michael D. Ernst. Type Annotations specification (JSR 308). `https://checkerframework.org/jsr308/`, October 2013.

[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.