

Type Annotations Specification (JSR 308)

Michael D. Ernst
mernst@cs.washington.edu

August 12, 2009

The JSR 308 webpage is <http://groups.csail.mit.edu/pag/jsr308/>. It contains the latest version of this document, along with other information such as a FAQ, the reference implementation, and sample annotation processors.

Contents

1	Introduction	2
2	Java language syntax extensions	3
2.1	Source locations for annotations on types	3
2.2	Java language grammar changes	3
2.3	Target meta-annotations for type annotations	4
3	Compiler modifications	5
4	Class file format extensions	5
4.1	The <code>extended_annotation</code> structure	6
4.2	The <code>target_type</code> field	7
4.3	The <code>reference_info</code> field	7
4.3.1	Typecasts, type tests, object creation and class literal	7
4.3.2	Local Variables	7
4.3.3	Type Arguments on Method Parameters	9
4.3.4	Method Receivers	9
4.3.5	Type Arguments on method return type and fields	9
4.3.6	Type Parameters	9
4.3.7	Type Parameter Bounds	9
4.3.8	Constructor and method call type arguments	10
4.3.9	Class <code>extends</code> and <code>implements</code> Clauses	10
4.3.10	<code>throws</code> Clauses	10
4.3.11	Wildcard bounds	10
4.3.12	Generic Type Arguments and Arrays	11
5	Detailed grammar changes	11
A	Example use of type annotations: Type qualifiers	14
A.1	Examples of type qualifiers	14
A.2	Example tools that do pluggable type-checking for type qualifiers	16

B	Discussion of Java language syntax extensions	16
B.1	Examples of annotation syntax	16
B.2	Uses for annotations on types	18
B.3	Syntax of array annotations	21
B.4	Disambiguating type and declaration annotations	22
C	Discussion of tool modifications	23
C.1	Compiler	23
C.2	ASTs and annotation processing	24
C.3	Reflection	25
C.3.1	Non-changes to reflection	25
C.4	Virtual machine and class file analysis tools	25
C.5	Other tools	25
D	Other possible extensions to Java annotations	26
D.1	Duplicate (multiple, repeated) annotations at a location	26
D.1.1	Desugar into wrapper annotations	27
D.1.2	Add new <code>getAnnotations</code> method	28
D.1.3	JSR 175 design justification	28
D.2	Locations for annotations	28
D.2.1	Annotations on statements	28
D.2.2	Expression annotations	29
D.2.3	Implicit Java types in casts	29
D.2.4	Only certain statements	30
D.3	Changes to the annotation type	30
D.3.1	Inheritance among annotations (subclassing/subtyping annotations)	30
D.3.2	Potentially recursive annotations (Annotations as members of annotations)	31
D.3.3	Null as an annotation field value (nullable fields)	32
D.3.4	Positional arguments	32
D.4	Semantics of annotations	32
D.4.1	Annotations for specific purposes	32
D.4.2	Annotation inheritance	33
D.4.3	Default annotations	33
D.5	Type abbreviations and typedefs	34
D.6	Class file syntax	34
D.6.1	Reducing class file size via use of the constant pool	34
D.6.2	Optional/default fields in annotations	34
D.7	Access to method bodies in annotation processing API	34
E	Logistical matters	34
E.1	Edits to existing standards documents	35
E.2	Testing (TCK, Technology Compatibility Kit)	35
E.3	Process	36
F	Related work	36

1 Introduction

JSR 308 extends Java’s annotation system [Blo04a] so that annotations may appear on nearly any use of a type. (By contrast, Java SE 6 permits annotations only on class/method/field/variable declarations; JSR 308 is backward-compatible and continues to permit those annotations.) Such a generalization removes

limitations of Java’s annotation system, and it enables new uses of annotations. This proposal also notes a few other possible extensions to annotations (see Section D).

This document specifies the *syntax* of extended Java annotations, but it makes no commitment as to their *semantics*. As with Java’s existing annotations [Blo04a], the semantics is dependent on annotation processors (compiler plug-ins), and not every annotation is necessarily sensible in every location where it is syntactically permitted to appear. This proposal is compatible with existing annotations, such as those specified in JSR 250, “Common Annotations for the Java Platform” [Mor06], and proposed annotations, such as those to be specified in JSR 305, “Annotations for Software Defect Detection” [Pug06]. (For a comparison of JSR 305 and JSR 308, see Section D.4, page 32.)

This proposal does not change the compile-time, load-time, or run-time semantics of Java. It does not change the abilities of Java annotation processors as defined in JSR 269 [Dar06]. The proposal merely makes annotations more general — and thus more useful for their current purposes, and also usable for new purposes that are compatible with the original vision for annotations [Blo04a].

This document has two parts: a short normative part and a longer non-normative part. The normative part specifies the changes to the Java language syntax (Sections 2 and 5), the Java toolset (Section 3), and the class file format (Section 4).

The non-normative part consists of appendices that discuss and explain the specification or deal with logistical issues. It motivates annotations on types by presenting one possible use, type qualifiers (Appendix A). It gives examples of and further motivation for the Java syntax changes (Appendix B) and lists tools that must be updated to accommodate the Java and class file modifications (Appendix C). Appendix D lists other possible extensions to Java annotations, some of which are within the scope of JSR 308 (and might be included in a future revision) and some of which are not. The document concludes with logistical matters relating to incorporation in the Sun JDK (Section E) and related work (Section F).

A JSR, or Java Specification Request, is a proposed specification for some aspect of the Java platform — the Java language, virtual machine, libraries, etc. For more details, see the Java Community Process FAQ at <http://jcp.org/en/introduction/faq>.

A FAQ (Frequently Asked Questions) document complements this specification; see <http://pag.csail.mit.edu/jsr308/jsr308-faq.html>.

2 Java language syntax extensions

2.1 Source locations for annotations on types

In Java SE 6, annotations can be written only on method parameters and the declarations of packages, classes, methods, fields, and local variables. JSR 308 extends Java to allow annotations on any use of a type. JSR 308 uses a simple prefix syntax for type annotations, with a special case for receiver types and for constructor results.

1. A type annotation appears before the type, as in `@NonNull String`.
2. The annotation on a given array level prefixes the brackets that introduce that level. To declare a non-empty array of English-language strings, write `@English String @NotEmpty []`. The `varargs` syntax `...` is treated analogously to array brackets and may also be prefixed by an annotation.
3. An annotation on the type of a method receiver (`this`) appears just after the parameter list and before any `throws` clause. (Each non-static method has an implicit parameter, `this`, which is called the *receiver*.)

Section B.1 contains examples of the annotation syntax.

2.2 Java language grammar changes

This section summarizes the Java language grammar changes, which correspond to the three rules of Section 2.1. Section 5 shows the grammar changes in detail. Additions are underlined.

1. Any *Type* may be prefixed by *[Annotations]*:

Type:

[Annotations] Identifier *[TypeArguments]* { . Identifier *[TypeArguments]* } { *□* }
[Annotations] *BasicType*

2. To permit annotations on levels of an array (in declarations, not constructors), change “{*□*” to “{*[Annotations]* *□*”}. (This was abstracted out as “*BracketsOpt*” in the 2nd edition of the JLS [GJSB00].) For example:

Type:

[Annotations] Identifier *[TypeArguments]*{ . Identifier *[TypeArguments]* } { *[Annotations]* *□* }
[Annotations] *BasicType*

Also permit annotations on varargs (...):

FormalParameterDeclsRest:

VariableDeclaratorId [, *FormalParameterDecls*]
[Annotations] ... *VariableDeclaratorId*

3. Annotations may appear on the receiver type by changing uses of “*FormalParameters*” (in all 5 places it appears in the grammar) to “*FormalParameters* *[Annotations]*”. For example:

VoidMethodDeclaratorRest:

FormalParameters *[Annotations]* [throws *QualifiedIdentifierList*] (*MethodBody* | ;)

4. Annotations may appear on the type parameter declaration in a class or method declaration.

TypeParameter:

[Annotations] Identifier [extends *Bound*]

5. Annotations may appear on the wildcard in any type argument.

TypeArgument:

[Annotations] ? [(extends | super) *Type*]

2.3 Target meta-annotations for type annotations

Java uses the `@Target` meta-annotation as a machine-checked way of expressing where an annotation is intended to appear. The `ElementType` enum classifies the places an annotation may appear in a Java program. JSR 308 adds two new constants to the `ElementType` enum.

- `ElementType.TYPE_PARAMETER` stands for a type parameter — that is, the declaration of a type variable.
- `ElementType.TYPE_USE` stands for all uses of types.

For example, in this declaration:

```
@Target(ElementType.TYPE_USE)
public @interface NonNull { ... }
```

the `@Target(ElementType.TYPE_USE)` meta-annotation indicates that `@NonNull` may appear on any use of a type.

`ElementType.TYPE_PARAMETER` and `ElementType.TYPE_USE` are distinct from the existing `ElementType.TYPE` enum element of Java SE 6, which indicates that an annotation may appear on a type declaration (a class, interface, or enum declaration).

The compiler applies an annotation to every target that is consistent with its meta-annotation. The order of annotations is not used to disambiguate. As in Java SE 6, the compiler issues an error if a programmer places an annotation in a location not permitted by its `Target` meta-annotation. (The compiler issues the error even if no annotation processor is being run.)

3 Compiler modifications

When generating `.class` files, the compiler must emit the attributes described in Section 4.

The compiler is required to preserve annotations in the class file. More precisely, if a programmer places an annotation (with class file or runtime retention) on the type of an expression, and that expression is represented in the compiled class file, then the annotation must be present, in the compiled class file, on the type of the compiled representation of the expression. If the compiler optimizes away an expression, then it may also optimize away the annotation. (Exception: when a type cast is optimized away without optimizing away its argument, the annotation remains on the argument; see Section 4.2.)

The compiler sometimes creates new methods that did not appear in the source code, and that do nothing but call an existing method. In this case, annotations should be copied from the method being invoked. Two examples are bridge methods (an implementation strategy used when the erased signature of the actual method being invoked differs from that of the compile-time method declaration [GJSB05, §15.12.4.5]) and anonymous constructors [GJSB05, §15.9.5.1]. (As of Java SE 6, `javac` does not copy/transfer any annotations from original methods to the compiler-generated methods; that is probably a bug in `javac`. It is, however, perhaps debatable whether *all* annotations should be copied.)

4 Class file format extensions

This section defines how to store type annotations in a Java class file. It also defines how to store local variable annotations, which are permitted in Java SE 6 source code but are discarded by the compiler.

Why store type annotations in the class file? The class file format represents the type of every variable and expression in a Java class, including all temporaries and values stored on the stack. (Sometimes the representation is explicit, and sometimes it is implicit.) Since JSR 308 permits annotations to be added to a type, the class file format should be updated to continue to represent the full, annotated type of each expression.

More pragmatically, Java annotations must be stored in the class file for two reasons.

First, annotated *signatures* (public members) must be available to tools that read class files. For example, a type-checking compiler plug-in [Dar06, PAC⁺08] needs to read annotations when compiling a client of the class file. The Checker Framework (<http://groups.csail.mit.edu/pag/jsr308/current/checkers-manual.html>) is one way to create such plug-ins.

Second, annotated method *bodies* must be present to permit checking the class file against the annotations. This is necessary to give confidence in an entire program, since its parts (class files) may originate from any source. Otherwise, it would be necessary to simply trust annotated classes of unknown provenance [BHP07].

How Java SE 6 stores annotations in the class file In Java SE 6, an annotation is stored in the class file in an *attribute* [Blo04a, LY07]. An attribute associates data with a program element (a method's bytecodes, for instance, are stored in a `Code` attribute of the method). The `RuntimeVisibleParameterAnnotations` stores formal parameter annotations that are accessible at runtime using reflection, and the `RuntimeInvisibleParameterAnnotations` attribute stores formal parameter annotations that are not accessible at runtime. `RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations` are analogous, but for annotations on fields, methods, and classes.

These attributes contain arrays of `annotation` structure elements, which in turn contain arrays of `element_value` pairs. The `element_value` pairs store the names and values of an annotation's arguments.

Annotations on a field are stored as attributes of the field's `field_info` structure [LY07, §4.6]. Annotations on a method are stored as attributes of the method's `method_info` structure [LY07, §4.7]. Annotations on a class are stored as attributes of the class's `attributes` structure [LY07, §4.2].

Generic type information is stored in a different way in the class file, in a signature attribute. It details are not germane to the current discussion.

Changes in JSR 308 JSR 308 introduces two new attributes: `RuntimeVisibleTypeAnnotations` and `RuntimeInvisibleTypeAnnotations`. These attributes are structurally identical to the `RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations` attributes described above with one exception: rather than an array of `annotation` elements, `RuntimeVisibleTypeAnnotations` and `RuntimeInvisibleTypeAnnotations` contain an array of `extended_annotation` elements, which are described in Section 4.1.

An annotation is stored in a `Runtime[In]visibleTypeAnnotations` attribute on the smallest enclosing class, field, or method.

Backward compatibility For backward compatibility, JSR 308 uses new attributes for storing the type annotations. In other words, JSR 308 merely reserves the names of a few attributes and specifies their layout. JVMs ignore unknown attributes. JSR 308 does not alter the way that existing annotations on classes, methods, method parameters, and fields are stored in the class file. JSR 308 mandates no changes to the processing of existing annotation locations; in the absence of other changes to the class file format, class files generated from programs that use no new annotations will be identical to those generated by a standard Java SE 6 compiler. Furthermore, the bytecode array will be identical between two programs that differ only in their annotations. Attributes have no effect on the bytecode array, because they exist outside it; however, they can represent properties of it by referring to the bytecode (including specific instructions, or bytecode offsets).

To do In Java SE 6, the class file stores the types of elements on the stack; this eliminates the need for bytecode verification to perform type inference for local variables and stack elements. The class file format should similarly store the annotations on those types, to aid in annotation verification in class files.

4.1 The `extended_annotation` structure

The `extended_annotation` structure has the following format:

```
extended_annotation {
    // Original fields from "annotation" structure:
    u2 type_index;
    u2 num_element_value_pairs;
    {
        u2 element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs];
    // New fields in JSR 308:
    u1 target_type;    // the type of the targeted program element, see Section 4.2
    {
        ...
    } reference_info; // uniquely identifies the targeted program element, see Section 4.3
}
```

We first briefly recap the three fields of `annotation` [LY07, §4.8.15].

- `type_index` is an index into the constant pool indicating the annotation type for this annotation.
- `num_element_value_pairs` is a count of the `element_value_pairs` that follow.
- Each `element_value_pairs` table entry represents a single element-value pair in the annotation (in the source code, these are the arguments to the annotation): `element_name_index` is a constant pool entry for the name of the annotation type element, and `value` is the corresponding value [LY07, §4.8.15.1].

Compared to `annotation`, the `extended_annotation` structure contains two additional fields. These fields implement a discriminated (tagged) union type: field `target_type` is the tag (see Section 4.2), and its value determines the size and contents of `reference_info` (see Section 4.3).

4.2 The `target_type` field

The `target_type` field denotes the type of program element that the annotation targets, such as whether the annotation is on a field, a method receiver, a cast, or some other location. Figure 1 gives the value of `target_type` for every possible annotation location.

Enumeration elements marked `*` never appear in a `target_type` field. Sometimes this is because annotations cannot be written in that location, such as because Java prohibits writing the location itself. In other cases this is because the annotations were permitted in Java 5, so there is already a place to store them in the class file rather than placing them in the new `Runtime[In]visibleTypeAnnotations` attribute. The unused enumeration elements are included for completeness.

4.3 The `reference_info` field

`reference_info` is a structure that contains enough information to uniquely identify the target of a given annotation. A different `target_type` may require a different set of fields, so the structure of the `reference_info` is determined by the value of `target_type`.

4.3.1 Typecasts, type tests, object creation and class literal

When the annotation's target is a typecast, an `instanceof` expression, a `new` expression, or a class literal expression `reference_info` has the following structure:

```
{
    u2 offset;
} reference_info;
```

The `offset` field denotes the offset (i.e., within the bytecodes of the containing method) of the `checkcast` bytecode emitted for the typecast, the `instanceof` bytecode emitted for the type tests, or of the `new` bytecode emitted for the object creation expression. Typecast annotations are attached to a single bytecode, not a bytecode range (or ranges): the annotation provides information about the type of a single value, not about the behavior of a code block. A similar explanation applies to type tests and object creation.

For annotated typecasts, the attribute may be attached to a `checkcast` bytecode, or to any other bytecode. The rationale for this is that the Java compiler is permitted to omit `checkcast` bytecodes for typecasts that are guaranteed to be no-ops. For example, a cast from `String` to `@NonNull String` may be a no-op for the underlying Java type system (which sees a cast from `String` to `String`). If the compiler omits the `checkcast` bytecode, the `@NonNull` attribute would be attached to the (last) bytecode that creates the target expression instead. This approach permits code generation for existing compilers to be unaffected.

See Section 4.3.12 for handling of generic type arguments and arrays.

4.3.2 Local Variables

When the annotation's target is a local variable, `reference_info` has the following structure:

```
{
    u2 table_length;
    {
        u2 start_pc;
        u2 length;
        u2 index;
    } table[table_length];
} reference_info;
```

The `table_length` field specifies the number of entries in the `table` array; multiple entries are necessary because a compiler is permitted to break a single variable into multiple live ranges with different local variable indices. The `start_pc` and `length` fields specify the variable's live range in the bytecodes of the local variable's containing method (from offset `start_pc` to offset `start_pc + length`). The `index` field stores the local variable's

Annotation target	target_type value	reference.info definition
method receiver	0x06	§4.3.4
method receiver generic/array	0x07*	Future extension to JSR 308
method return type	0x0A*	USE Runtime[In]visibleAnnotations
method return type generic/array	0x0B	§4.3.5
method parameter	0x0C*	USE Runtime[In]visibleParameterAnnotations
method parameter generic/array	0x0D	§4.3.3
field	0x0E*	USE Runtime[In]visibleAnnotations
field generic/array	0x0F	§4.3.5
class type parameter bound	0x10	§4.3.7
class type parameter bound generic/array	0x11	+§4.3.12
method type parameter bound	0x12	§4.3.7
method type parameter bound generic/array	0x13	+§4.3.12
class extends/implements	0x14	§4.3.9
class extends/implements generic/array	0x15	+§4.3.12
exception type in throws	0x16	§4.3.10
exception type in throws generic/array	0x17*	forbidden by Java
wildcard bound	0x1C	§4.3.11
wildcard bound generic/array	0x1D	+§4.3.12
method type parameter	0x20	§4.3.6
method type parameter generic/array	0x21*	Java does not support
class type parameter	0x22	§4.3.6
class type parameter generic/array	0x23*	Java does not support
typecast	0x00	§4.3.1
typecast generic/array	0x01	+§4.3.12
type test (instanceof)	0x02	§4.3.1
type test (instanceof) generic/array	0x03	+§4.3.12
object creation (new)	0x04	§4.3.1
object creation (new) generic/array	0x05	+§4.3.12
local variable	0x08	§4.3.2
local variable generic/array	0x09	+§4.3.12
type argument in constructor call	0x18	§4.3.8
type argument in constructor call generic/array	0x19	+§4.3.12
type argument in method call	0x1A	§4.3.8
type argument in method call generic/array	0x1B	+§4.3.12
class literal	0x1E	§4.3.1
class literal generic/array	0x1F*	forbidden by Java

Figure 1: Values of `target_type` for each possible target of a type annotation. Ordinary Java annotations on declarations are not included, because they appear in `annotation`, not `extended_annotation`, attributes in the class file. Table elements such as local variable, method parameter, and field refer to the declaration, not the use, of such elements.

Enumeration elements marked * will never appear in a `target_type` field; see Section 4.2.

The top half of the table contains targets that may appear on classes and members — places that annotations already appear in the class file in Java SE 6. The bottom half of the table contains targets that only appear inside method bodies, where annotations do not appear in Java SE 6 (not even local variable annotations written by the programmer).

index in that method. These fields are similar to those of the optional `LocalVariableTable` attribute [LY07, §4.8.12].

Storing local variable annotations in the class file raises certain challenges. For example, live ranges are not isomorphic to local variables. Further, a local variable with no live range may not appear in the class file (but it is also irrelevant to the program).

4.3.3 Type Arguments on Method Parameters

When the annotation's target is a method parameter, `reference_info` indicates which of the method's formal parameters is being annotated:

```
{
    u1 parameter_index;
    u2 location_length;
    u1 location[location_length];
} reference_info;
```

The `location_length` and `location` fields are as in Section 4.3.12.

4.3.4 Method Receivers

When the annotation's target is a method receiver (or an inner class constructor receiver), `reference_info` is empty.

4.3.5 Type Arguments on method return type and fields

When the annotation's target is a type argument of a method return type, a constructor result, or a field, `reference_info` has the following structure:

```
{
    u2 location_length;
    u1 location[location_length];
} reference_info;
```

The `location_length` and `location` fields are as in Section 4.3.12.

4.3.6 Type Parameters

When the annotation's target is a type parameter of a class or method, `reference_info` has the following structure:

```
{
    u1 param_index;
} reference_info;
```

`param_index` specifies the 0-based index of the type parameter.

4.3.7 Type Parameter Bounds

When the annotation's target is a bound of a type parameter of a class or method, `reference_info` has the following structure:

```
{
    u1 param_index;
    u1 bound_index;
} reference_info;
```

`param_index` specifies the index of the type parameter, while `bound_index` specifies the index of the bound. Consider the following example:

```
<T extends @A Object & @B Comparable, U extends @C Cloneable>
```

Here `@A` has `param_index` 0 and `bound_index` 0, `@B` has `param_index` 0 and `bound_index` 1, and `@C` has `param_index` 1 and `bound_index` 0.

4.3.8 Constructor and method call type arguments

When the annotation's type is a type argument in a constructor call or a method call, `reference_info` has the following structure:

```
{
    u2 offset;
    u1 type_index;
} reference_info;
```

The `offset` field denotes the offset (i.e., within the bytecodes of the containing method) of the `new` bytecode emitted for constructor call, or the `invoke{interface|special|static|virtual}` bytecode emitted for method invocation. Like type cast type annotations, type argument annotations are attached to a single bytecode, not a bytecode range.

`type_index` specifies the index of the type argument in the expression.

4.3.9 Class extends and implements Clauses

When the annotation's target is a type in an `extends` or `implements` clause, `reference_info` has the following structure:

```
{
    u1 type_index;
} reference_info;
```

`type_index` specifies the index of the type in the clause: `-1` (255) is used if the annotation is on the superclass type, and the value `i` is used if the annotation is on the `i`th superinterface type (counting from zero).

4.3.10 throws Clauses

When the annotation's target is a type in a `throws` clause, `reference_info` has the following structure:

```
{
    u1 type_index;
} reference_info
```

`type_index` specifies the index of the exception type in the clause: the value `i` denotes an annotation on the `i`th exception type (counting from zero).

4.3.11 Wildcard bounds

When the annotation's target is the bound of a wildcard, `reference_info` has the following structure:

```
{
    u1 wildcard_location_type;    // The location of the wildcard, values as for target_type (Section 4.2)
    {
        ...
    } reference_info; // uniquely identifies the targeted program element, see Section 4.3
} reference_info
```

Here are examples:

`@A in void m(List<? extends @A String> lst) { ... }` is described as:

```
target_type = WILDCARD_BOUND
reference_info =
{
    wildcard_location_type = METHOD_PARAMETER (0x0C)
    wildcard_location =
    {
        u1 parameter_index = 0;
    }
}
```

Declaration: @A Map<@B Comparable<@C Object[@D][@E][@F]>, @G List<@H Document>>

Annotation	location_length	location
@A	not applicable	
@B	1	0
@C	2	0, 0
@D	3	0, 0, 0
@E	3	0, 0, 1
@F	3	0, 0, 2
@G	1	1
@H	2	1, 0

Figure 2: Values of the `location_length` and `location` fields for a sample declaration.

@B in `Map<Object, ? extends List<@B String>> newMap() { ... }` is described as:

```
target_type = WILDCARD_BOUND_GENERIC_OR_ARRAY
reference_info =
{
  // location of wildcard: type argument of the return type
  wildcard_location_type = RETURN_TYPE_GENERIC_OR_ARRAY;
  wildcard_location =
  {
    // the second type variable
    location_length = 1;
    location = { 1 };
  }
  // the generic information of the annotation itself
  location_length = 1;
  location = { 0 };
}
```

4.3.12 Generic Type Arguments and Arrays

When the annotation's target is a generic type argument or array type, `reference_info` contains what it normally would for the raw type (e.g., `offset` for an annotation on a type argument in a typecast), *plus* the following fields at the end:

```
u2 location_length;
u1 location[location_length];
```

The `location_length` field specifies the number of elements in the variable-length `location` field. `location` encodes which type argument or array element the annotation targets. Specifically, the *i*th item in `location` denotes the index of the type argument or array dimension at the *i*th level of the hierarchy. Figure 2 shows the values of the `location_length` and `location` fields for the annotations in a sample field declaration.

TODO: The specification does not indicate how to differentiate @A and @B uniquely in `OuterClass<@A String>.InnerClass<@B String>`.

5 Detailed grammar changes

This section gives detailed changes to the grammar of the Java language [GJSB05, ch. 18], based on the conceptually simple summary from Section 2.2. Additions are underlined.

This section is of interest primarily to language tool implementers, such as compiler writers. Most users can read just Sections 2.1 and B.1.

Infelicities in the Java grammar make this section longer than the simple summary of Section 2.2. Some improvements are possible (for instance, by slightly refactoring the Java grammar), but this version attempts to minimize changes to existing grammar productions.

Type:

[Annotations] UnannType

UnannType:

Identifier [TypeArguments]{ . Identifier [TypeArguments]} {[Annotations] []}
BasicType

FormalParameterDecls:

[final] [Annotations] UnannType *FormalParameterDeclsRest*

ForVarControl:

[final] [Annotations] UnannType *Identifier* *ForVarControlRest*

MethodOrFieldDecl:

UnannType *Identifier* *MethodOrFieldRest*

InterfaceMethodOrFieldDecl:

UnannType *Identifier* *InterfaceMethodOrFieldRest*

MethodDeclaratorRest:

FormalParameters {[Annotations] []} [Annotations] [throws *QualifiedIdentifierList*] (*MethodBody* | ;)

VoidMethodDeclaratorRest:

FormalParameters [Annotations] [throws *QualifiedIdentifierList*] (*MethodBody* | ;)

InterfaceMethodDeclaratorRest:

FormalParameters {[Annotations] []} [Annotations] [throws *QualifiedIdentifierList*] ;

VoidInterfaceMethodDeclaratorRest:

FormalParameters [Annotations] [throws *QualifiedIdentifierList*] ;

ConstructorDeclaratorRest:

FormalParameters [Annotations] [throws *QualifiedIdentifierList*] *MethodBody*

Primary:

...
BasicType {[Annotations] []} .class

IdentifierSuffix:

[Annotations] [([{[Annotations] []} .class | *Expression*])
 ...

VariableDeclaratorRest:

{[Annotations] []} [= *VariableInitializer*]

ConstantDeclaratorRest:

{[Annotations] []} [= *VariableInitializer*]

VariableDeclaratorId:

Identifier {[Annotations] □}

FormalParameterDeclsRest:

VariableDeclaratorId [, *FormalParameterDecls*]
[Annotations] ... *VariableDeclaratorId*

TypeParameter:

[Annotations] *Identifier* [extends *Bound*]

A Example use of type annotations: Type qualifiers

One example use of annotation on types is to create custom type qualifiers for Java, such as `@NonNull`, `@ReadOnly`, `@Interned`, or `@Tainted`. Type qualifiers are modifiers on a type; a declaration that uses a qualified type provides extra information about the declared variable. A designer can define new type qualifiers using Java annotations, and can provide compiler plug-ins to check their semantics (for instance, by issuing lint-like warnings during compilation). A programmer can then use these type qualifiers throughout a program to obtain additional guarantees at compile time about the program.

The type system defined by the type qualifiers does not change Java semantics, nor is it used by the Java compiler or run-time system. Rather, it is used by the checking tool, which can be viewed as performing type-checking on this richer type system. (The qualified type is usually treated as a subtype or a supertype of the unqualified type.) As an example, a variable of type `Boolean` has one of the values `null`, `TRUE`, or `FALSE` (more precisely, it is null or it refers to a value that is equal to `TRUE` or to `FALSE`). A programmer can depend on this, because the Java compiler guarantees it. Likewise, a compiler plug-in can guarantee that a variable of type `@NonNull Boolean` has one of the values `TRUE` or `FALSE` (but not `null`), and a programmer can depend on this. Note that a type qualifier such as `@NonNull` refers to a type, not a variable, though JSR 308 could be used to write annotations on variables as well.

Type qualifiers can help prevent errors and make possible a variety of program analyses. Since they are user-defined, developers can create and use the type qualifiers that are most appropriate for their software.

A system for custom type qualifiers requires extensions to Java’s annotation system, described in this document; the existing Java SE 6 annotations are inadequate. Similarly to type qualifiers, other pluggable type systems [Bra04] and similar lint-like checkers also require these extensions to Java’s annotation system.

Our key goal is to create a type qualifier system that is compatible with the Java language, VM, and toolchain. Previous proposals for Java type qualifiers are incompatible with the existing Java language and tools, are too inexpressive, or both. The use of annotations for custom type qualifiers has a number of benefits over new Java keywords or special comments. First, Java already implements annotations, and Java SE 6 features a framework for compile-time annotation processing. This allows JSR 308 to build upon existing stable mechanisms and integrate with the Java toolchain, and it promotes the maintainability and simplicity of the modifications. Second, since annotations do not affect the runtime semantics of a program, applications written with custom type qualifiers are backward-compatible with the vanilla JDK. No modifications to the virtual machine are necessary.

Four compiler plug-ins that perform type qualifier type-checking, all built using JSR 308, are distributed at the JSR 308 webpage, <http://groups.csail.mit.edu/pag/jsr308/>. The four checkers, respectively, help to prevent and detect null pointer errors (via a `@NonNull` annotation), equality-checking errors (via a `@Interned` annotation), mutation errors (via the Javari [BE04, TE05] type system), and mutation errors (via the IGJ [ZPA⁺07] type system). A paper [PAC⁺08] discusses experience in which these plug-ins exposed bugs in real programs.

A.1 Examples of type qualifiers

The ability to place annotations on arbitrary occurrences of a type improves the expressiveness of annotations, which has many benefits for Java programmers. Here we mention just one use that is enabled by extended annotations, namely the creation of type qualifiers. (Figure 3 gives an example of the use of type qualifiers.)

As an example of how JSR 308 might be used, consider a `@NonNull` type qualifier that signifies that a variable should never be assigned `null` [Det96, Eva96, DLNS98, FL03, CMM05]. A programmer can annotate any use of a type with the `@NonNull` annotation. A compiler plug-in would check that a `@NonNull` variable is never assigned a possibly-null value, thus enforcing the `@NonNull` type system.

`@ReadOnly` and `@Immutable` are other examples of useful type qualifiers [ZPA⁺07, BE04, TE05, GF05, KT01, SW01, PBKM00]. Similar to C’s `const`, an object’s internal state may not be modified through references that are declared `@ReadOnly`. A type qualifier designer would create a compiler plug-in (an annotation processor) to check the semantics of `@ReadOnly`. For instance, a method may only be called on a `@ReadOnly` object if the method was declared with a `@ReadOnly` receiver. (Each non-static method has an implicit parameter, `this`, which is

```

1 @DefaultQualifier("NonNull")
2 class DAG {
3
4     Set<Edge> edges;
5
6     // ...
7
8     List<Vertex> getNeighbors(@Interned @ReadOnly Vertex v) @ReadOnly {
9         List<Vertex> neighbors = new LinkedList<Vertex>();
10        for (Edge e : edges)
11            if (e.from() == v)
12                neighbors.add(e.to());
13        return neighbors;
14    }
15 }

```

Figure 3: The DAG class, which represents a directed acyclic graph, illustrates how type qualifiers might be written by a programmer and checked by a type-checking plug-in in order to detect or prevent errors.

- (1) The `@DefaultQualifier("NonNull")` annotation (line 1) indicates that no reference in the DAG class may be null (unless otherwise annotated). It is equivalent to writing line 4 as `"@NonNull Set<@NonNull Edge> edges;"`, for example. This guarantees that the uses of `edges` on line 10, and `e` on lines 11 and 12, cannot cause a null pointer exception. Similarly, the (implicit) `@NonNull` return type of `getNeighbors()` (line 8) enables its clients to depend on the fact that it will always return a `List`, even if `v` has no neighbors.
- (2) The two `@ReadOnly` annotations on method `getNeighbors` (line 8) guarantee to clients that the method does not modify, respectively, its `Vertex` argument or its DAG receiver. The lack of a `@ReadOnly` annotation on the return value indicates that clients are free to modify the returned `List`.
- (3) The `@Interned` annotation on line 8 (along with an `@Interned` annotation on the return type in the declaration of `Edge.from()`, not shown) indicates that the use of object equality (`==`) on line 11 is a valid optimization. In the absence of such annotations, use of the `equals` method is preferred to `==`.

called the *receiver*.) `@ReadOnly`'s immutability guarantee can help developers avoid accidental modifications, which are often manifested as run-time errors. An immutability annotation can also improve performance. The Access Intents mechanism of WebSphere Application Server already incorporates such functionality: a programmer can indicate that a particular method (or all methods) on an Enterprise JavaBean is `readonly`.

Additional examples of useful type qualifiers abound. We mention just a few others. C uses the `const`, `volatile`, and `restrict` type qualifiers. Type qualifiers `YY` for two-digit year strings and `YYYY` for four-digit year strings helped to detect, then verify the absence of, Y2K errors [EFA99]. Expressing units of measurement (e.g., SI units such as meter, kilogram, second) can prevent errors in which a program mixes incompatible quantities; units such as dollars can prevent other errors. Range constraints, also known as ranged types, can indicate that a particular `int` has a value between 0 and 10; these are often desirable in realtime code and in other applications, and are supported in languages such as Ada and Pascal. Type qualifiers can indicate data that originated from an untrustworthy source [PØ95, VS97]; examples for C include `user` vs. `kernel` indicating user-space and kernel-space pointers in order to prevent attacks on operating systems [JW04], and `tainted` for strings that originated in user input and that should not be used as a format string [STFW01]. A `localizable` qualifier can indicate where translation of user-visible messages should be performed. Annotations can indicate other properties of its contents, such as the format or encoding of a string (e.g., XML, SQL, human language, etc.). `Local` and `remote` qualifiers can indicate whether particular resources are available on the same machine or must be retrieved over the network. An `interned` qualifier can indicate which objects have been converted to canonical form and thus may be compared via reference equality. Type qualifiers such as `unique` and `unaliased` can express properties about pointers and aliases [Eva96, CMM05]; other qualifiers can detect and prevent deadlock in concurrent programs [FTA02, AFKT03]. A `ThreadSafe` qualifier [GPB⁺06] could indicate that a given field should contain a thread-safe implementation of a given interface; this is more flexible than annotating the interface itself to require that *all* implementations must be thread-safe.

Annotations can identify performance characteristics or goals; for example, some collections should not be iterated over, and others should not be used for random access. Annotations (both type qualifiers and others) can specify cut points in aspect-oriented programming (AOP) [EM04]. Flow-sensitive type qualifiers [FTA02] can express typestate properties such as whether a file is in the open, read, write, readwrite, or closed state, and can guarantee that a file is opened for reading before it is read, etc. The Vault language’s type guards and capability states are similar [DF01].

A.2 Example tools that do pluggable type-checking for type qualifiers

The Checker Framework (<http://groups.csail.mit.edu/pag/jsr308/current/checkers-manual.html>) gives a way to create pluggable type-checkers. A pluggable type-checker verifies absence of certain bugs (and also verifies correct usage of type qualifiers). The Checker Framework is distributed with a set of example type-checkers. The Checker Framework is built on the Type Annotations (JSR 308) syntax, though it also permits annotations to be written in comments for compatibility with previous versions of Java.

B Discussion of Java language syntax extensions

In Java SE 6, annotations can be written only on method parameters and the declarations of packages, classes, methods, fields, and local variables. Additional annotations are necessary in order to fully specify Java classes and methods.

B.1 Examples of annotation syntax

This section gives examples of the annotation syntax specified in Sections 2.1 and 5. This list is not necessarily exhaustive (but if you notice something missing, let us know so that we can add it). Section B.2 motivates annotating these locations by giving the meaning of annotations that need to be applied to these locations.

- for generic type arguments to parameterized classes:

```
Map<@NonNull String, @NotEmpty List<@ReadOnly Document>> files;
```
- for generic type arguments in a generic method or constructor invocation:

```
o.<@NonNull String>m("...");
```
- for type parameter bounds and wildcard bounds:

```
class Folder<F extends @Existing File> { ... }
Collection<? super @Existing File>
```
- for type parameters:

```
interface WonderfulList<@Reified E> { ... }
```
- for class inheritance:

```
class UnmodifiableList<T> implements @ReadOnly List<@ReadOnly T> { ... }
```
- for throws clauses:

```
void monitorTemperature() throws @Critical TemperatureException { ... }
```
- for method receivers:

```
public String toString() @ReadOnly { ... }
public void write() @Writable throws IOException { ... }
```

It is desirable for a method to express constraints on the generic parameters of the receiver, just as is already possible for other formal parameters. The grammar of Section 5 does not currently permit this, but it may be revised in the future. For example, here are examples of a possible syntax:


```

class C<T> {
    public int size() @ReadOnly<@ReadOnly> { ... }
}
class Map<T,U> {
    public void requiresNonNullKeys() <@NonNull,> { ... }
}

```

- for constructor results:

```

class Invocation {
    @Immutable Invocation() { ... }
    ...
}

```

Note that the result of a constructor is different from the receiver. The receiver only exists for inner class constructors. It is the containing object, and in the body of the constructor it is referred to as *Supertype.this*. In the constructor body, the result is referred to as *this*. In any non-constructor, the receiver (if any) is referred to as *this*.

- for arrays:

```

@ReadOnly Document [][] docs1 = new @ReadOnly Document [2][12]; // array of arrays of read-only documents
Document @ReadOnly [][] docs2 = new Document @ReadOnly [2][12]; // read-only array of arrays of documents
Document[] @ReadOnly [] docs3 = new Document[2] @ReadOnly [12]; // array of read-only arrays of documents

```

This syntax permits independent annotations for each distinct level of array, and for the elements.

- for typecasts:

```

myString = (@NonNull String) myObject;

```

It is not permitted to omit the Java type, as in `myString = (@NonNull) myObject;`; see Sections B.2 and D.2.3.

- for constructor results:

```

new @Interned MyObject()

```

For generic constructors (JLS §8.8.4), the annotation follows the explicit type arguments (JLS §15.9):

```

new <String> @Interned MyObject()

```

- for type tests:

```

boolean isNonNull = myString instanceof @NonNull String;

```

It is not permitted to omit the Java type, as in `myString instanceof @NonNull;` see Sections B.2 and D.2.3.

- for object creation:

```

new @NonEmpty @ReadOnly List<String>(myNonEmptyStringSet)

```

- for class literals:

```

Class<@NonNull String> c = @NonNull String.class;

```

- for static member access:

```

@NonNull Type.field

```

B.2 Uses for annotations on types

This section gives examples of annotations that a programmer may wish to place on a type. Each of these uses is either impossible or extremely inconvenient in the absence of the new locations for annotations proposed in this document. For brevity, we do not give examples of uses for every type annotation. The specific annotation names used in this section, such as `@NonNull`, are examples only; this document does not define any annotations, merely specifying where they can appear in Java code.

It is worthwhile to permit annotations on all uses of types (even those for which no immediate use is apparent) for consistency, expressiveness, and support of unforeseen future uses. An annotation need not utilize every possible annotation location. For example, a system that fully specifies type qualifiers in signatures but infers them for implementations [GF05] may not need annotations on typecasts, object creation, local variables, or certain other locations. Other systems may forbid top-level (non-type-argument, non-array) annotations on object creation (`new`) expressions, such as `new @Interned Object()`.

Generics and arrays Generic collection classes are declared one level at a time, so it is easy to annotate each level individually.

It is desirable that the syntax for arrays be equally expressive. Here are examples of uses for annotations on array levels:

- The Titanium [YSP⁺98] dialect of Java requires the ability to place the `local` annotation (indicating that a memory reference in a parallel system refers to data on the same processor) on various levels of an array, not just at the top level.
- In a dependent type system [Pfe92, Xi98, XP99], one wishes to specify the dimensions of an array type, such as `Object @Length(3) [] @Length(10) []` for a 3×10 array.
- An immutability type system, as discussed in Section A.2, needs to be able to specify which levels of an array may be modified. Consider specifying a procedure that inverts a matrix in place. The procedure parameter type should guarantee that the procedure does not change the shape of the array (does not replace any of the rows with another row of a different length), but must permit changing elements of the inner arrays. In other words, the top-level array is immutable, the inner arrays are mutable, and their elements are immutable.
- An ownership domain system [AAA06] uses array annotations to indicate properties of array parameters, similarly to type parameters.
- The ability to specify the nullness of the array and its elements separately is so important that JML [LBR06] includes special syntax `\nonnullelements(a)` for a possibly-null array `a` with non-null elements.

A simple example is a method that accepts a list of files to search. `null` may be used to indicate that there were no files to search, but when a list is provided, then the `Files` themselves must be non-null. Using JSR 308, a programmer would declare the parameter as `@NonNull File @Nullable [] filesToSearch` — more concisely, depending on the default nullness, as either `File @Nullable [] filesToSearch` or `@NonNull File [] filesToSearch`.

The opposite example, of a non-null array with nullable elements, is typical of fields in which, when an array element is no longer relevant, it is set to null to permit garbage collection.

- In a type system for preventing null pointer errors, using a default of non-null, and explicitly annotating references that may be null, results in the fewest annotations and least user burden [FL03, CJ07, PAC⁺08]. Array elements can often be null (both due to initialization, and for other reasons), necessitating annotations on them.

Receivers A type qualifier on a formal parameter is a contract regarding what the method may (or may not) do with that parameter. Since the method receiver (`this`) is an implicit formal parameter, programmers should be able to express type qualifiers on it, for consistency and expressiveness. An annotation on the receiver is a contract regarding what the method may (or may not) do with its receiver.

For example, consider the following method:

```
package javax.xml.bind;
class Marshaller {
    void marshal(@ReadOnly Object jaxbElement,
                @Mutable Writer writer) @ReadOnly {
        ...
    }
}
```

The annotations indicate that `marshal` modifies its second parameter but does not modify its first parameter nor its receiver.

A receiver annotation is different than a class annotation, a method annotation, or a return value annotation:

- There may be different receiver annotations on different methods that cannot be factored out into the containing class.
- Stating that a method does not modify its receiver is different than saying the method has no side effects at all, so it is not appropriate as a method annotation (such as JML’s `pure` annotation [LBR06]).
- A receiver annotation is also distinct from a return value annotation: a method might modify its receiver but return an immutable object, or might not modify its receiver but return a mutable object.

Since a receiver annotation is distinct from other annotations, new syntax is required for the receiver annotation. The syntax is adopted from C++, just as the overall syntax of Java was. This syntax is cleaner than creating a parallel annotation, such as `@ReadOnlyReceiver`, for each type annotation, and is also cleaner than changing the definition of annotations to permit writing the similar `@Receiver(@ReadOnly)`.

As with Java’s annotations on formal parameters, annotations on the receiver do not affect the Java signature, compile-time resolution of overloading, or run-time resolution of overriding. The Java type of every receiver in a class is the same — but their annotations, and thus their qualified type in a type qualifier framework, may differ.

Some people question the need for receiver annotations. In case studies [PAC⁺08], every type system required some receiver annotations. Even the Nullness type system required them to express whether the receiver was fully initialized (only in a fully-initialized object can fields be guaranteed to be non-null). So, the real question is how to express receiver annotations, not whether they should exist.

Casts There are two distinct reasons to annotate the type in a type cast: to fully specify the casted type (including annotations that are retained without change), or to indicate an application-specific invariant that is beyond the reasoning capability of the Java type system. Because a user can apply a type cast to any expression, a user can annotate the type of any expression. (This is different than annotating the expression itself; see Section D.2.2.)

1. Annotations on type casts permit the type in a type cast to be fully specified, including any appropriate annotations. In this case, the annotation on the cast is the same as the annotation on the type of the operand expression. The annotations are preserved, not changed, by the cast, and the annotation serves as a reminder of the type of the cast expression. For example, in

```
@ReadOnly Object x;
... (@ReadOnly Date) x ...
```

the cast preserves the annotation part of the type and changes only the Java type. If a cast could not be annotated, then a cast would remove the annotation:

```
@ReadOnly Object x;
... (Date) x ...      // annotation processor issues warning due to casting away @ReadOnly
```

This cast changes the annotation; it uses `x` as a non-`@ReadOnly` object, which changes its type and would require a run-time mechanism to enforce type safety.

An annotation processor could permit the unannotated cast syntax but implicitly add the annotation, treating the cast type as `@ReadOnly Date`. This has the advantage of brevity, but the disadvantage of being less explicit and of interfering somewhat with the second use of cast annotations. Experience will indicate which design is better in practice.

2. A second use for annotations on type casts is — like ordinary Java casts — to provide the compiler with information that is beyond the ability of its typing rules. Such properties are often called “application invariants”, since they are facts guaranteed by the logic of the application program.

As a trivial example, the following cast changes the annotation but is guaranteed to be safe at run time:

```
final Object x = new Object();
... (@NonNull Object) x ...
```

An annotation processing tool could trust such type casts, perhaps issuing a warning to remind users to verify their safety by hand or in some other manner. An alternative approach would be to check the type cast dynamically, as Java casts are, but we do not endorse such an approach, because annotations are not intended to change the run-time behavior of a Java program and because there is not generally a run-time representation of the annotations.

Type tests Annotations on type tests (`instanceof`) allow the programmer to specify the full type, as in the first justification for annotations on type casts, above. However, the annotation is not tested at run time — the JVM only checks the base Java type. In the implementation, there is no run-time representation of the annotations on an object’s type, so dynamic type test cannot determine whether an annotation is present. This abides by the intention of the Java annotation designers, that annotations should not change the run-time behavior of a Java program.

Annotation of the type test permits the idiom

```
if (x instanceof MyType) {
    ... (MyType) x ...
}

if (x instanceof MyType) {
    ... (MyType) x ...
}
```

to be used with the same annotated type τ in both occurrences. By contrast, using different types in the type test and the type cast might be confusing.

To prevent confusion caused by incompatible annotations, an annotation processor could require the annotation parts of the operand and the type to be the same:

```
@ReadOnly Object x;
if (x instanceof Date) { ... }      // error: incompatible annotations
if (x instanceof @ReadOnly Date) { ... } // OK
Object y;
if (y instanceof Date) { ... }      // OK
if (y instanceof @NonNull Date) { ... } // error: incompatible annotations
```

(As with type casts, an annotation processor could implicitly add a missing annotation; this would be more concise but less explicit, and experience will dictate which is better for users.)

As a consequence of the fact that the annotation is not checked at run time, in the following

```

if (x instanceof @A1 T) { ... }
else if (x instanceof @A2 T) { ... }

```

the second conditional is always dead code. An annotation processor may warn that one or both of the `instanceof` tests is a compile-time type error.

A non-null qualifier is a special case because it is possible to check at run time whether a given value can have a non-null type. A type-checker for a non-null type system could take advantage of this fact, for instance to perform flow-sensitive type analysis in the presence of a `x != null` test, but JSR 308 makes no special allowance for it.

Object creation Java’s `new` operator indicates the type of the object being created. As with other Java syntax, programmers should be able to indicate the full type, even if in some cases (part of) the type can be inferred. In some cases, the annotation cannot be inferred; for instance, it is impossible to tell whether a particular object is intended to be mutated later in the program or not, and thus whether it should have a `@Mutable` or `@Immutable` annotation. Annotations on object creation expressions could also be statically verified (at compile time) to be compatible with the annotations on the constructor.

Type bounds Annotations on type parameter bounds (`extends`) and wildcard bounds (`extends` and `super`) allow the programmer to fully constrain generic types. Creation of objects with constrained generic types could be statically verified to comply with the annotated bounds.

Inheritance Annotations on class inheritance (`extends` and `implements`) are necessary to allow a programmer to fully specify a supertype. It would otherwise be impossible to extend the annotated version of a particular type *t* (which is often a valid subtype or supertype of *t*) without using an anonymous class.

These annotations also provide a convenient way to alias otherwise cumbersome types. For instance, a programmer might declare

```

final class MyStringMap extends
    @ReadOnly Map<@NonNull String, @NonEmpty List<@NonNull @ReadOnly String>> {}

```

so that `MyStringMap` may be used in place of the full, unpalatable supertype. (However, also see Section D.5 for problems with this approach.)

Throws clauses Annotations in the `throws` clauses of method declarations allow programmers to enhance exception types. For instance, programs that use the `@Critical` annotation from the above examples could be statically checked to ensure that `catch` blocks for `@Critical` exceptions are not empty.

B.3 Syntax of array annotations

As discussed in Section B.2, it is desirable to be able to independently annotate both the base type and each distinct level of a nested array. Forbidding annotations on arbitrary levels of an array would simplify the syntax, but it would reduce expressiveness to an unacceptable degree. The syntax of array annotations follows the same general prefix rule as other annotations, though it looks slightly different because the syntax of array types is different than the syntax of other Java types. (Arrays are less commonly used than generics, so even if you don’t like the array syntax, it need not bother you in most cases.)

Most programmers read the Java type `String[][]` as “array of arrays of Strings”. Analogously, the construct `new String[2][5]` is “new length-2 array of length-5 array of Strings”. After `a = new String[2][5]`, `a` is an array with 2 elements, and `a[1]` is a 5-element array.

In other words, the order of reading an array type is left-to-right for the brackets, *then* left-to-right for the base type.

```

type:                String                []                []
order of reading:  2-----> 1 ----->

```

To more fully describe the 2x5 array, a programmer could use the type “length-2 array of length-5 array of Strings”:

```
type:                String  @Length(2) [] @Length(5) []  
order of reading:  2-----> 1 ----->
```

The prefix notation is natural, because the type is read in exactly the same order as any Java array type. As another example, to express “non-null array of length-10 arrays of English Strings” a programmer would write

```
type:                @English String  @NonNull [] @Length(10) []  
order of reading:  2-----> 1 ----->
```

An important property of this syntax is that adding array levels does not change the meaning of existing annotations. For example, `var1` has the same annotations as the elements of `arr2`:

```
@NonNull String var1;  
@NonNull String[] arr2;
```

because in each case `@NonNull` refers to the `String`, not the array. This consistency is especially important since the two variables may appear in a single declaration:

```
@NonNull String var1, arr2[];
```

A potential criticism is that a type annotation at the very beginning of a declaration does not refer to the full type, even though variable annotations (which also occur at the beginning of the declaration) do refer to the entire variable. As an example, in `@NonNull String[] arr2`; the variable `arr2` is not non-null. This is actually a criticism of Java itself, not of the JSR 308 annotation extension, which is merely consistent with Java. In a declaration `String[] arr2`;, the top-level type constructor does not appear on the far left. An annotation on the whole type (the array) should appear on the syntax that indicates the array — that is, on the brackets.

Other array syntaxes can be imagined, but they are less consistent with Java syntax and therefore harder to read and write. Examples include making annotations at the beginning of the type refer to the whole type, using a postfix syntax rather than a prefix syntax, and postfix syntax within angle brackets as for generics.

B.4 Disambiguating type and declaration annotations

An annotation before a method declaration annotates either the return type, or the method declaration. There is never any ambiguity regarding the programmer intention: in that location, a type annotation annotates the return type, and a non-type annotation annotates the method itself. The `@Target` meta-annotation indicates whether an annotation is a type annotation. Field declarations are treated similarly.

Suppose that we have these annotation declarations:

```
@Target(ElementType.TYPE_USE)  
@interface NonNegative { }  
  
@Target(ElementType.METHOD)  
@interface Override { }  
  
@Target(ElementType.FIELD)  
@interface GuardedBy { ... }
```

Then, in

```
@Override  
@NonNegative int getHeight() { ... }
```

`@Override` applies to the method and `@NonNegative` applies to the return type. Furthermore, in these two field declarations

```
@NonNegative int balance;  
@GuardedBy("myLock") long lastAccessedTime;
```

the annotation `@NonNegative` applies to the field type `int`, not to the whole variable declaration nor to the variable `balance` itself. The annotation `@GuardedBy("accessLock")` applies to the field `lastAccessedTime`.

Here are a few facts that follow from the specification. For brevity, we use “type annotation” as shorthand for “an annotation that is meta-annotated with `@Target(ElementType.TYPE_USE)`”.

- A type annotation need not also be meta-annotated with the targets `ElementType.TYPE`, `ElementType.METHOD`, or `ElementType.FIELD` in order to be applied to a class, a method return type, or a field type — and it generally should *not* contain such a `@Target` meta-annotation.
- A type annotation may not appear before a void method.
- A type annotation may not appear on a package (packages do not contain a use of a type).
- A type annotation may appear before a constructor, in which case it represents the object that the constructor is creating.
- As with any other non-static method, a type annotation may appear on the receiver of an inner class constructor.

Strictly speaking, a `@Target` meta-annotation could indicate that an annotation is *both* a type annotation and a non-type annotation. In such a case, the annotation would apply to both the return type and the method declaration, and it would exist twice in the class file. We have not found an example where that is desirable; although it is legal, it is considered bad style.

A related case is that when annotation has no `@Target` meta-annotation (which is bad style!), it is treated as if it applies to *all* locations, to ensure that it can be used wherever the designer intended. Applying the annotation to multiple locations does not cause any problems, because any annotation processor will look only where the designer had in mind and will ignore any other positions. For example, the `@Deprecated` annotation lacks a `@Target` annotation, so in this code:

```
@Deprecated int foo() { return 0; }
```

the `@Deprecated` annotation applies to both the `int` data type and to the `foo` method. Any annotation processor that processes the `@Deprecated` annotation will not even look in the type annotation position, and the existence of an extra annotation in that position will not interfere with any other annotation processor either.

In summary: for certain syntactic locations, which target (Java construct) is being annotated depends on the annotation. There is no ambiguity for the compiler: the compiler applies the annotation to every target that is consistent with its meta-annotation (see Section 2.3). In practice, programmers have no difficulty in understanding what a given annotation means.

C Discussion of tool modifications

This section primarily discusses tool modifications that are consequences of JSR 308’s changes to the Java syntax and class file format, as presented in Sections 2 and 4.

C.1 Compiler

The syntax extensions described in Section 2 require the `javac` Java compiler to accept annotations in the proposed locations and to add them to the program’s AST. The relevant AST node classes must also be modified to store these annotations.

Javac's `-xprint` functionality reads a `.class` file and prints the interface (class declarations with signatures of all fields and methods). (The `-xprint` functionality is similar to `javap`, but cannot provide any information about bytecodes or method bodies, because it is implemented internally as an annotation processor.) This must be updated to print the extended annotations as well. Also see Section C.4.

Section 3 requires compilers to place certain annotations in the class file. This is consistent with the principle that annotations should not affect behavior: in the absence of an annotation processor, the compiler produces the same bytecodes for annotated code as it would have for the same code without annotations. (The class file may differ, since the annotations are stored in it, but the bytecode part does not differ.)

This may change the compiler implementation of certain optimizations, such as common subexpression elimination, but this restriction on the compiler implementation is unobjectionable for three reasons.

1. Java-to-bytecode compilers rarely perform sophisticated optimizations, since the bytecode-to-native (JIT) compiler is the major determinant in Java program performance. Thus, the restriction will not affect most compilers.
2. The compiler workarounds are simple. Suppose that two expressions that are candidates for common subexpression elimination have different type annotations. A compiler could: not perform the optimization when the annotations differ; create a single expression whose type has both of the annotations (e.g., merging `@Positive Integer` 42 and `@Even Integer` 42 into `@Positive @Even Integer` 42); or create an unannotated expression and copy its value into two variables with differently-annotated types.
3. It seems unlikely that two identical, non-trivial expressions would have differently-annotated types. Thus, any compiler restrictions will have little or no effect on most compiled programs.

Java compilers can often produce bytecode for an earlier version of the virtual machine, via the `-target` command-line option. For example, a programmer could execute a compilation command such as `javac -source 7 -target 5 MyFile.java`. A Java 7 compiler produces a class file with the same attributes for type annotations as when the target is a version 7 JVM. However, the compiler is permitted to also place type annotations in declaration attributes. For instance, the annotation on the top level of a return type would also be placed on the method (in the method attribute in the class file). This enables class file analysis tools that are written for Java SE 5 to view a subset of the type qualifiers (lacking generics, array levels, method receivers, etc.), albeit attached to declarations.

A user can use a Java SE 5/6 compiler to compile a Java class that contains type annotations, so long as the type annotations only appear in places that are legal in Java SE 5. Furthermore, the compiler must be provided with a definition of the annotation that is meta-annotated not with `@Target(ElementType.TYPE_USE)` (since `ElementType.TYPE_USE` does not exist in Java SE 5/6), but with no meta-annotation or with a meta-annotation that permits annotations on any declaration.

C.2 ASTs and annotation processing

The Java Model AST of JSR 198 (Extension API for Integrated Development Environments) [Cro06] gives access to the entire source code of a method. This AST (abstract syntax tree) must be updated to represent all new locations for annotations.

Sun's Tree API, which exposes the AST (including annotations) to authors of `javac` annotation processors (compile-time plug-ins), must be updated to reflect the modifications made to the internal AST node classes described in Section 2. The same goes for other Java compilers, such as that of Eclipse).

Like reflection, the JSR 269 (annotation processing) model does not represent constructs below the method level, such as individual statements and expressions. Therefore, it needs to be updated only with respect to declaration-related annotations (the top of Figure 1; also see Section D.7). The JSR 269 model, `javax.lang.model.*`, already has some classes representing annotations; see <http://java.sun.com/javase/6/docs/api/javax/lang/model/element/package-summary.html>. The annotation processing API in `javax.annotation.processing` must also be revised.

C.3 Reflection

The `java.lang.reflect.*` and `java.lang.Class` APIs give access to annotations on public API elements such as classes, method signatures, etc. They must be updated to give the same access to the new extended annotations in the top of Figure 1.

Here are a few examples (the design is not yet complete).

1. `java.lang.reflect.Type` needs to implement `java.lang.reflect.AnnotatedElement`. (An alternative would be an `java.lang.reflect.AnnotatedType` interface, with two methods that return an annotation and a type. This design seems suboptimal, because most clients would immediately cast the result of the latter method.)

C.3.1 Non-changes to reflection

Reflection gives no access to method implementations, so no changes are needed to provide access to annotations on casts (or other annotations inside a method body), type parameter names, or similar implementation details.

The Mirror API `com.sun.mirror.*` need not be updated, as it has been superseded by JSR 269 [Dar06].

Method `Method.getParameterAnnotations()` returns the annotations on the parameter *declarations*, just as in Java SE 6. It does not return type annotations. There is no point in new methods that parallel it, such as `Method.getReceiverAnnotation` (for the receiver `this`) and `Method.getReturnAnnotation` (for the return value). Rather, the interface will provide a uniform mechanism for querying annotations on types.

The semantics of reflective invocation is not changed. (The changes described in this section are to APIs that query classes, method signatures, etc.) For instance, suppose that a program reflectively calls a method with a parameter whose type is annotated as `@ReadOnly`, but the corresponding argument has a declared type that is non-`@ReadOnly`. The call succeeds. This is a requirement for backward compatibility: the existence of annotations in the class file should not cause a standard JVM to behave differently than if the annotations are not present (unless the program uses reflection to explicitly examine the annotations). Likewise, other reflective functionality such as `AtomicReferenceFieldUpdater` can bypass annotation constraints on a field.

C.4 Virtual machine and class file analysis tools

No modifications to the virtual machine are necessary. (The changes to reflection (Section C.3) do change virtual machine APIs in a minor way, but the representation of execution of bytecodes is unaffected.)

The `javap` disassembler must recognize the new class file format and must output annotations.

The `pack200/unpack200` tool must preserve the new attributes through a compress-decompress cycle.

The compiler and other tools that read class files are trivially compatible with class files produced by a Java SE 5/6 compiler. However, the tools would not be able to read the impoverished version of type qualifiers that is expressible in Java SE 5 (see Section C.1). It is desirable for class file tools to be able to read at least that subset of type qualifiers. Therefore, APIs for reading annotations from a class file should be dependent on the class file version (as a number of APIs already are). If the class file version indicates Java 5 or 6, and none of the extended annotations defined by JSR 308 appear in the class file, then the API may return (all) annotations from declarations when queried for the annotations on the top-level type associated with the declaration (for example, the top-level return type, for a method declaration).

C.5 Other tools

Javadoc must output annotations at the new locations when those are part of the public API, such as in a method signature.

Similar modifications need to be made to tools outside the Sun JDK, such as IDEs (Eclipse, IDEA, JBuilder, jEdit, NetBeans), other tools that manipulate Java code (grammars for CUP, `javacc`), and tools that manipulate class files (ASM, BCEL). These changes need to be made by the authors of the respective tools.

A separate document, “Custom type qualifiers via annotations on Java types” (<http://groups.csail.mit.edu/pag/jsr308/java-type-qualifiers.pdf>), explores implementation strategies for annotation processors that act as type-checking compiler plug-ins. It is not germane to this proposal, both because this proposal does not concern itself with annotation semantics and because writing such plug-ins does not require any changes beyond those described in this document.

A separate document, “Annotation File Specification” (<http://types.cs.washington.edu/annotation-file-utilities/annotation-file-format.pdf>), describes a textual format for annotations that is independent of `.java` or `.class` files. This textual format can represent annotations for libraries that cannot or should not be modified. We have built tools for manipulating annotations, including extracting annotations from and inserting annotations in `.java` and `.class` files. That file format is not part of this proposal for extending Java’s annotations; it is better viewed as an implementation detail of our tools.

D Other possible extensions to Java annotations

The Type Annotations (JSR 308) specification may be extended in the future, both to improve the specification and also to add new material. Any improvement to Java’s annotation system is within bounds. The JSR is titled “Type Annotations” to avoid a vague and inelegant title such as “Extensions to Java’s annotation system”, and because the initial extensions enable annotations on types. However, non-type-related improvements to annotations are within scope and will be considered. This is especially true if the additional changes are small, there is no better venue to add such an annotation, and the new syntax would permit unanticipated future uses.

This section gives examples of further extensions to Java’s annotation system that may be considered, either for JSR 308 or in the future. Inclusion here is not an endorsement; this list is intended to be a resource for current and future language designers, explaining pros and cons of possible changes. If you care enough about one of these extensions, please volunteer to be the point person for the work. That could make them a reality.

Java has a massive worldwide user base, so the bar to modifying the Java language is quite high. Any proposed modification requires both compelling use cases (for many users, not just a niche community) that are currently impossible, and it requires a convincing demonstration (not just a claim) that there are no negative consequences, such as undesirable interactions with current language features or tools. Where possible, this section gives a brief discussion of benefits and potential problems as a starting point. (Problems that potentially apply to all extensions, such as existing code that reflects over annotations encountering unexpected structures, are not repeated for each extension.)

The last annotations JSR. It is not a goal that JSR 308 is the last annotation-related JSR. It is acceptable to leave some issues to future language designers, just as JSR 175 (the previous annotations JSR [Blo04a]) did. Leaving issues unresolved is preferable to making hasty decisions, or decisions with unknown technical implications. By contrast, it *is* a goal of JSR 308 not to unnecessarily close off realistic future avenues of extension.

D.1 Duplicate (multiple, repeated) annotations at a location

It may be desirable for some annotations to be specified more than once at a single location. Currently, this is impossible: “It is a compile-time error if a declaration is annotated with more than one annotation for a given annotation type” [GJSB05, §9.7]. (By contrast, `C#` supports duplicate annotations on a given program element.)

As a related motivation, array-valued annotations can be clumsy to write:

```
@Resources({
    @Resource(name = "db1", type = DataSource.class)
    @Resource(name = "db2", type = DataSource.class)
```

```

})
public class MyClass { ... }

```

A cleaner syntax is desirable for both purposes:

```

@Resource(name = "db1", type = DataSource.class)
@Resource(name = "db2", type = DataSource.class)
public class MyClass { ... }

```

We note two possible approaches to this problem: desugaring into arrays, or adding a `getAnnotations` method to two interfaces. Among the two proposals, the second is cleaner, more elegant, and easier for clients to use. However, it requires either changing existing interfaces or creating new ones.

For backward compatibility, in both approaches an annotation may be repeated only if its definition is meta-annotated with `@Repeatable`:

```

public @interface Repeatable {
    public boolean value() = true;
}

```

In the below, assume that the definitions of `A` and `B` are meta-annotated with `@Repeatable`.

D.1.1 Desugar into wrapper annotations

Desugar duplicate annotations into the current array syntax. For instance, desugar

```
@A(1) @B @A(2) Object x;
```

into

```
@AContainer({@A(1), @A(2)}) @B Object x;
```

This approach treats duplicate annotations as purely a syntactic convenience; it does not change annotations in any deep way. This approach is compatible with certain existing J2EE annotation processors that are already written to process both `@A` and `@AContainer`.

This proposal would need to specify what happens if only one `@A` annotation is present. Most likely, the desugaring would still place `@A` into a singleton `@AContainer` annotation. Otherwise, client code would become more complicated, since it would have to deal with the possibility of either an `@A` annotation or an `@AContainer` annotation.

This proposal would need to specify what happens if both `@A` and `@AContainer` annotations are present. Most likely, the desugaring would insert `@A` into the existing `@AContainer` annotation, either at the front or the back depending on whether the `@A` or `@AContainer` annotation came first. Otherwise, client code would become more complicated, since it would have to deal with the possibility of both an `@A` annotation or an `@AContainer` annotation, and with handling the relative order.

One problem with this proposal is that it loses the ordering of differently-named annotations (even if the ordering of same-named annotations is preserved within the container). For example, it cannot distinguish these declarations:

```

@A(1) @B @A(2) Object x;
@A(1) @A(2) @B Object x;

```

Another problem is that it requires defining an `@AContainer` annotation for each annotation `@A`, or else annotation `@A` cannot be duplicated.

D.1.2 Add new `getAnnotations` method

Add a new `getAnnotations` method, which returns multiple annotations, to two interfaces: `java.lang.reflect.AnnotatedElement` and `javax.lang.model.element.Element`.

1. Slightly modify the specification of the existing method

```
<T extends Annotation> T getAnnotation(Class<T> annotationClass)
```

When duplicate annotations exist, method `getAnnotation` could either give the first one (convenient for backward compatibility) or throw an exception (convenient to prevent erroneous processing). The latter seems like a better choice.

2. Create a new method

```
<T extends Annotation> T[] getAnnotations(Class<T> annotationClass)
```

that returns all annotations for the specified annotation type that are present on the receiver element. As with `getDeclaredAnnotations`, the return value is never null but may have zero length.

The arrays returned by the methods `getAnnotations(Class)`, `getAnnotations()`, and `getDeclaredAnnotations()` are required to preserve the ordering of the annotations as they were declared in the original `.java` source file.

No other changes would be necessary. Existing code that uses some other workaround (like special `@Container` annotations) would continue to work. Or, it could be converted to take advantage of this new mechanism.

The problem with this proposal is that it requires adding a single method, `getAnnotations`, to two interfaces. While the changes would not break upward binary-compatibility, they would break source-compatibility: existing implementations of `Element` and `AnnotatedElement` would not compile without adding a `getAnnotations` method. Sun's policy generally prohibits such changes, though exceptions are made. For example, the SQL interfaces have far more implementations than the compiler or reflection APIs, but Java SE 6 adds three methods to the `java.sql.Statement` interface and makes it extend `Wrapper`.

An alternative to changing the `Element` (and `AnnotatedElement`) interface is to create a new `Element2` interface that augments `Element` with the `getAnnotations` method, and to mark `Element` as deprecated. Then, programmers should rewrite their existing code to use the `Element2` interface instead of the `Element` interface. However, their code continues to compile until they make the change.

D.1.3 JSR 175 design justification

The JSR 175 Design FAQ [Blo04b] briefly addresses why this feature was not originally included:

Why is it illegal [to] annotate a single element repeatedly with the same annotation type?

A similar effect is available by annotating once with an annotation type whose sole element is of an array type. The resulting system is simpler and cleaner: you don't have to worry about whether you're going to get back a single instance of the annotation type or multiple instances when you query an annotation. Also it's consistent with the usual restriction on modifiers.

D.2 Locations for annotations

D.2.1 Annotations on statements

Annotations on statements (or on some subset of statements, such as blocks or loops) would be useful for a variety of purposes, including atomicity/concurrency. Supporting annotations on statements would require defining both Java syntax and a convention for storing the information in the class file. See http://doc.ece.uci.edu/mediawiki/index.php/JSR-308_Statements for a proposal that summarizes why statement

annotations are desirable, and that proposes a Java syntax, a classfile storage format, and how other tools will accommodate them. If you would like to help this feature become a reality, then please pitch in! You can join the `jsr308-statements@lists.csail.mit.edu` mailing list (via <https://lists.csail.mit.edu/mailman/listinfo/jsr308-statements/>), expand the partial design on the wiki, or work on the implementation.

The JSR 175 Design FAQ [Blo04b] briefly addresses why this feature was not originally included:

Why can't you annotate arbitrary program elements such as blocks and individual statements?

This would greatly complicate the annotation syntax: We would have to sacrifice the simplicity of saying that annotations are simply modifiers, which can be used on declarations.

D.2.2 Expression annotations

Annotating an expression indicates some property of the computation, such as that it should be performed atomically, that it acquires no locks, or that it should be formatted specially by an IDE. JSR 308 does not support expression annotations, because we have not yet discovered compelling use cases for them that cannot be equally well supported by statement annotations. (A minor inconvenience is that the use of statement annotations may require the programmer to create a separate statement for the expression to be annotated.) Expression annotations are not type annotations and are different than annotating a type cast, which indicates a property of a value (the result of an expression).

D.2.3 Implicit Java types in casts

Arbitrary values can be annotated using an annotation on a cast:

```
@Language("SQL") String "select * from foo"
```

A possible shorthand would be to permit the Java type to be implicit:

```
@Language("SQL")) "select * from foo"
```

This is not permitted, nor may a cast be omitted in a type test, as in “`x instanceof @NonNull`”. There are several reasons for this decision:

1. Erasing the annotations should leave a valid Java program.
2. Stating the type reinforces that the annotation is a type annotation rather than an expression annotation.
3. Especially in a type test, stating the type reinforces that the run-time effect is to check and change the Java type. In general, no run-time check of the annotation is possible.
4. The benefit of omitting the type in the cast seems relatively minor.

An even shorter shorthand would drop the parentheses:

```
@Language("SQL") "select * from foo"
```

In addition to the benefits and problems noted above, such an annotation is syntactically ambiguous with an expression annotation. Whether an annotation applies to expressions or to types is clear from the annotation's documentation and its `@Target` meta-annotation, similarly to how it is determined whether an annotation applies to a type or to a declaration (Section B.4).

D.2.4 Only certain statements

It would be possible to permit annotations only on blocks and/or loops, as a restricted special case of statements. This is less general than permitting annotations on statements, and uses are more syntactically cluttered (for instance, this requires a statement to be converted into a block before it can be annotated). Most declarations could not be annotated as statements because enclosing the declaration in a block to annotate it would change (and limit) the variable's scope. This limitation in flexibility does yield the advantage that there would be no syntactic ambiguity between (say) statement annotations and declaration or type annotations.

Similarly, permitting annotations on partial constructs (such as only the body of a loop) appears both more complex, and no more useful, than annotating complete constructs (such as a full statement).

D.3 Changes to the annotation type

D.3.1 Inheritance among annotations (subclassing/subtyping annotations)

Annotations cannot subclass one another, and an annotation is not allowed to extend/implement any interfaces. These restrictions make it difficult to share behavior or to express similarities or relationships among annotation types. For example, it is inconvenient to define annotations with choices in their structure, such as a discriminated union that uses field names that act as explicit tags. It is impossible to create annotation processors or APIs that work with a specific set of annotations (say, all those with a given set of element names).

Permitting inheritance among annotations would solve these problems. (To work around the problem, one could meta-annotate an annotation as a “subannotation” of another, and then the annotation processor could do all the work to interpret the meta-annotation. This is clumsy and indirect.)

Without other changes, inheritance among annotations would not enable recursive or self-referential annotations (see Section D.3.2).

There are two general designs that permit inheritance among annotations: permitting annotations to subclass other annotations, or permitting annotations to extend/implement interfaces, or both. We briefly elaborate on the two designs.

Subclassing annotations Permitting inheritance among annotations requires that the `final` modifier on an annotation type declaration works just like for ordinary classes: it prevents creation of subtypes of the given annotation. A framework that defines annotation `@A` will not want to load an untrusted subclass `@B` of `@A` into the framework.

Subclassing annotations raises the possibility of multiple effective annotations applying at a location. Suppose that annotation types `Sub1` and `Sub2` are subtypes of `Super` which has an `int` value field and a location is annotated with both `@Sub1(1)` and `@Sub2(2)`. The old `getAnnotation(Class)` method should throw an error in this case. If duplicate annotations are permitted (see Section D.1), then the new `getAnnotations(Class)` method returns all of the annotations. (Then, the client can decide what to do with them, such as raise an error, merge the annotations, or take some other action.)

It is already possible in Java SE 6 for a location to have both an inherited and an explicit annotation of the same type. The `@Inherited` meta-annotation specification states that a `getAnnotation[s]` query returns only the explicit annotation. This requirement should perhaps be relaxed, even if inheritance among annotations is not permitted. For some annotation processors, it is best to check that the explicit annotation is consistent with the overridden, inherited one and possibly merge information in the two annotations.

The JSR 175 Design FAQ [Blo04b] briefly addresses why this feature was not originally included:

Why don't you support annotation subtyping (where one annotation type extends another)?

It complicates the annotation type system, and makes it much more difficult to write “Specific Tools”.

...

“Specific Tools” — Programs that query known annotation types of arbitrary external programs. Stub generators, for example, fall into this category. These programs will read annotated classes without loading them into the virtual machine, but will load annotation interfaces.

Extending/implementing interfaces Permitting annotation types to extend/implement interfaces is less powerful than subclassing among annotations. For example, interfaces can’t define default values, nor is it possible to subclass an annotation to enhance functionality. An advantage of this approach is that there is no possibility of multiple annotations with a given supertype appearing at a location.

One use case is marker interfaces. In another use case, the interface could define several elements/methods. These could define a standard set of element names that are used by a set of specific annotations. APIs that use the interface could work with any annotation from that set.

The currently permitted annotation member types (“primitive types, String, Class and any invocation of Class, an enum type, an annotation type, or an array of one of the preceding types” [GJSB05, §9.6]) should be extended to include interfaces (or perhaps only interfaces that extend `java.lang.annotation.Annotation`).

D.3.2 Potentially recursive annotations (Annotations as members of annotations)

In Java, an annotation can have a parameter (equivalently, a member) that is an annotation (JLS §9.6 and §9.7). JLS §9.7 gives this example:

```
@Author(@Name(first = "Joe", last = "Hacker"))
```

However, an annotation type cannot have a member of its own type. “It is a compile-time error if an annotation type T contains an element of type T , either directly or indirectly” [GJSB05, §9.6]. (If inheritance among annotations is permitted (see Section D.3.1), then the natural interpretation of this is that no element’s declared type may be either the annotation itself or any supertype thereof.)

As a particular example, it is not possible to define an annotation that takes an *arbitrary* annotation as a parameter, as in

```
@DefaultAnnotation(@AnyAnnotation)
```

These limitations reduce the expressiveness of annotations. Here are some examples:

- It is impossible to define annotations that take an arbitrary annotation as an argument.

Two examples of such annotations are the `@DefaultAnnotation` example above, and an annotation that expresses that a method is polymorphic over annotations (as opposed to polymorphic over types, as generics do).

Another example is annotation parameters that do not correspond to a type parameter. Consider a container class that is designed to hold things of a specific type (say, it interacts with the elements in domain-specific ways), not of an arbitrary type. The container has no type parameter, but it may still make sense to have annotations on the contained thing. Most of the examples of generics and arrays from Section B.2 also serve as examples here. The argument applies not just to container classes but also to algorithms that process data of a specific type.

These examples require the annotation declaration to be potentially recursive. However, they do not require that any recursive annotation ever be created in practice. They could be achieved even if creation of recursive annotation instances was prohibited.

- It is impossible to define annotations with recursive structure.

More generally, the kind of data that can be encoded in annotations is limited. It is generally possible to store data which has a fixed structure, but it is much more difficult to encode more complex data, including hierarchical data. As the success of XML, YAML, and other such technologies has shown, the ability to encode structured data is very powerful for a wide variety of uses, many of which we cannot anticipate today. Today’s annotations only fulfill a small piece of that potential.

To express the base case, recursive annotations also require either inheritance among annotations (Section D.3.1), or nullable members (Section D.3.3), or a requirement that recursion takes place through an array (an empty array would terminate the recursion).

A more modest approach that makes annotations somewhat more expressive would be to permit inheritance among annotations (see Section D.3.1) without permitting possibly self-referential annotations.

The JSR 175 Design FAQ [Blo04b] briefly addresses why this feature was not originally included:

Why is it illegal for an annotation type to contain an element of the same type?

At first glance it would appear that you could never annotate with such a type, so the prohibition might seem unnecessary, but you can get yourself into trouble if you also provide a default:

```
@interface Foo {
    Foo foo() default @Foo;
}

@Foo int i;
```

If this were legal, the resulting annotation would self-referential (or infinite), as well as useless.

D.3.3 Null as an annotation field value (nullable fields)

Currently, it is possible to choose/create a special value and to store that value in an annotation field. The proposal would permit use of `null` instead of an explicit user-specified value.

The proposal makes some programming idioms slightly shorter. For example, specifying the base case of a recursive data structure will require simply writing `null` instead of calling a constructor. The proposal does not eliminate any substantial complexity when processing a data structure, but only converts a check against a given value into a check against null.

We note some possible objections to the proposal.

The proposal doesn't make anything possible that was not possible before.

The programmer-defined special value provides better documentation than `null`, which might mean "none", "uninitialized", `null` itself, etc.

The proposal is more error-prone. It's much easier to forget checking against null than to forget checking for an explicit value.

The proposal may make the standard idiom more verbose. Currently only the users of an annotation need to check for its special values. With the proposal, many tools that process annotations will have to check whether a field's value is null lest they throw a null pointer exception.

D.3.4 Positional arguments

Annotation types cannot have positional arguments (except for the `value` argument, when it is the only argument). This limitation makes writing annotations with multiple arguments more verbose than necessary.

On a somewhat related topic, the "SingleElementAnnotation" form relies on the field name `value`. This name is not always appropriate; designers who wish to permit the "SingleElementAnnotation" form are forced to give a member a confusing name. It would be nice to permit the "SingleElementAnnotation" form to initialize a different member than `value` (say, via a `@ValueMember` annotation).

D.4 Semantics of annotations

D.4.1 Annotations for specific purposes

JSR 308 does not define any annotations. JSR 308 extends the Java and class file syntax to permit annotations to be written in more places, and thus makes existing and future annotations more useful to programmers.

By contrast, JSR 305 “Annotations for Software Defect Detection” aims to define a small set of annotations. Examples include type annotations such as non-nullness (`@NonNull`), signedness (`@Nonnegative`), tainting, and string format; and also non-type annotations such as whether a method’s return value should always be checked by the caller. A programmer who cares about code quality will use both annotations defined in the JSR 305 “standard library”, and also others that are defined by third parties or by the programmer himself. For more details about JSR 305, see <http://jcp.org/en/jsr/detail?id=305> and <http://groups.google.com/group/jsr-305/>.

Any type annotation, including those defined by JSR 305, is of limited use without the JSR 308 syntax. Without the JSR 308 annotation syntax, a static checker may lose track of type qualifiers whenever a program uses generic types (e.g., collection classes), whenever a method is invoked on an object, whenever a cast is performed, whenever a class is subclassed, etc. From the point of code checking, using the old Java annotation syntax is even worse than the type unsoundness of pre-generics Java, when there was no compiler-enforced type correctness guarantee for collections classes. Therefore, use of JSR 305 without JSR 308 is much less effective.

As of fall 2008, no reference implementation that uses JSR 305 annotations is planned. This hinders both programmers who want to use the annotations, and also people trying to interpret the meaning of the specification. By contrast, the JSR 308 reference implementation and the Checker framework for compile-time type-checking have been available since January 2007.

D.4.2 Annotation inheritance

(This section is *not* about whether the definition of an annotation may inherit from other classes/interfaces. Rather, it is about whether a class/interface inherits annotations from its superclasses/interfaces.)

The annotation type `java.lang.annotation.Inherited` (JLS §9.6.1.3) indicates that annotations on a class `c` corresponding to a given annotation type are inherited by subclasses of `c`. This implies that annotations on interfaces are not inherited, nor are annotations on members (methods, constructors, fields, etc.).

It might be useful to permit methods, fields, etc. to inherit annotations.

It might be useful to permit an annotation to be inherited from an interface as well as from a superclass.

It might be useful to permit annotation inheritance to merge information from the current and inherited annotations, instead of always choosing the inherited one.

D.4.3 Default annotations

Specifying a default for annotations can reduce code size and (when used carefully and sparingly) increase code readability. For instance, Figure 3 uses `@DefaultQualifier("NonNull")` to avoid the clutter of 5 `@NonNull` annotations. It would be nicer to have a general mechanism, such as

```
@DefaultAnnotation(NonNull.class, locations={ElementType.LOCAL_VARIABLE})
```

Defaults for annotations are a semantic issue that is out of the scope of JSR 308. It will be taken up by JSR 305 (“Annotations for Software Defect Detection” [Pug06]).

The defaulting syntax must also be able to specify the arguments to the default annotation (in the above example, the arguments to `@NonNull`).

A better syntax would use an annotation, not a string or class literal, as the argument to `@DefaultAnnotation`, as in

```
@DefaultAnnotation(@MyAnnotation(arg="foo"))
```

In Java, it is not possible to define an annotation that takes an arbitrary annotation as a parameter; see Section D.3.2.

An issue for JSR 260 (Javadoc) and JSR 305 (Annotation semantics) is how inherited and defaulted annotations are handled in Javadoc: whether they are written out in full, or in some abbreviated form. Just as too many annotations may clutter source code, similar clutter-reduction ideas may need to be applied to Javadoc.

D.5 Type abbreviations and typedefs

An annotated type may be long and hard to read; compare `Map<String, Object>` to `@NonNull Map<@NonNull String, @NonNull Object>`. Class inheritance annotations and subclassing provides a partial solution, as noted on page 21 in Section B.2 with the following example:

```
final class MyStringMap extends
    @ReadOnly Map<@NonNull String, @NonEmpty List<@NonNull @ReadOnly String>> {}
```

This approach limits reusability: if a method is declared to take a `MyStringMap` parameter, then a `Map` (even of the right type, including annotations) cannot be passed to it. (By contrast, a `MyStringMap` can always be used where a `Map` of the appropriate type is expected.) Goetz [GPB⁺06] recommends exploiting Java's type inference to avoid some (but not all) instances of the long type name.

In summary, a built-in typedef mechanism might achieve both code readability and reusability.

D.6 Class file syntax

Changes to the class file syntax are out of the scope of JSR 308, which, for backward compatibility, currently does not change the way that existing annotations are stored in the class file. Class file syntax changes require modification of compilers, JVMs, javap, and other class file tools (see Sections C.4 and C.5).

D.6.1 Reducing class file size via use of the constant pool

Annotations could be stored in the constant pool, and use constant pool references from the annotation points. That would reduce class file size, especially if an annotation is used in many places in the same class, as is more likely once JSR 308 support is in place.

D.6.2 Optional/default fields in annotations

In order to reduce the size of the class file, some fields may be omitted from the .class file, in which case any access of them returns their default value.

D.7 Access to method bodies in annotation processing API

A type-checking compiler plug-in (annotation processor) must process annotations (including those in method bodies), and it also must check each use of a variable/method whose declared type is annotated. For example, if a variable `x` is declared as `@NonNull Object x;`, then every assignment to `x` must be checked, because any assignment `x = null;` would be illegal.

The JSR 269 annotation processing API does not process annotations on local variables, as it is not designed to access method bodies. This limitation makes JSR 269 insufficient for creating a type-checking compiler plug-in.

An annotation and source code processing API for JSR 308 annotations could take advantage of JSR 198's Java Model. The Java Model defines a parsed view into the contents of a source file that is intended for construction of IDE extensions that are portable across multiple IDEs — precisely the situation with compiler plug-ins. JSR 308 may be shipped without defining this API, but defining this API may be desirable in the future (say, in a later version of Java), particularly after more experience is gained with JSR 308 annotation processors.

E Logistical matters

JSR 308 ("Type annotations") should be included under the Java SE 7 umbrella JSR (which lists the JSRs that are part of the Java SE 7 release). However, it should be a separate JSR because it needs a separate

expert group. The expert group will have overlap with any others dealing with other added language features that might be annotatable (such as method-reference types or closures), to check impact.

The specification and the TCK will be freely available, most likely licensed under terms that permit arbitrary use. The reference implementation is built on the OpenJDK Java implementation and is publicly available; our goal is for Sun to incorporate JSR 308 into the official OpenJDK release.

To ease the transition from standard Java SE 6 code to code with the extended annotations, the reference implementation recognizes the extended annotations when surrounded by comment markers:

```
List</*@ReadOnly*/ Object> myList;
```

This permits use of both standard Java SE 6 tools and the new annotations even before Java SE 7 is released. However, it is not part of the proposal; that is, it is not required that every Java compiler parses comments. The final Java SE 7 implementation may or may not support a switch that makes it recognize the new annotations when embedded in comments. The Spec# [BLS04] extension to C# can be made compilable by a standard C# compiler in a similar way, by enclosing its annotations in special `/*^...^*/` comment markers. The `/*@` comment syntax is a standard part of the Splint [Eva96], ESC/Java [FLL⁺02], and JML [LBR06] tools (that is, not with the goal of backward compatibility).

E.1 Edits to existing standards documents

Edits to the Java Language Specification (JLS): We need a document, complementary to the design document, that lists every edit that is required in the JLS. A preliminary step would be a list of all the locations that must be edited (for instance, by searching the entire JLS for uses of “annotation”, but the list will be a superset of the list of locations that were edited for JSR 175). The most important locations are the following.

- Changes to sections 9.6 and 9.7
- Merge the BNF description of the Java syntax changes (Sections 2.2 and 5) into JLS chapter 18: Syntax.

Edits to the Java Virtual Machine Specification (JVMS) [LY99, LY07]: We need a document, complementary to the design document, that lists every edit that is required in the JVMS. The most important of these is the following:

- Sections 4.8.15-18 define the `RuntimeV, Invisible, ParameterAnnotations` attributes. (See http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf for the JDK 1.5 revisions to chapter 4, “The class file Format”.) Similar definitions are required for `Runtime[In]visibleTypeAnnotations`.

E.2 Testing (TCK, Technology Compatibility Kit)

JSR 308 will ship with a test suite (known as a TCK, or Technology Compatibility Kit).

Each tool that needs to be tested appears in Section 3; the TCK will include tests for each of them.

For each modified tool, we will test backward compatibility by passing all of its existing tests. (We may need to modify a few of them, for instance those that depend on specific bytecodes that are created by the compiler.)

We will test most other functionality by creating a set of Java programs that include annotations in every possible location. For instance, this can be used to test all aspects of the compiler (parsing, code generation, `-Xprint`).

We will provide multiple annotation processors (including at least one for checking `@NonNull` and one for checking `@Interned`) that utilize the new annotations, along with a test suite for each one. Each annotation processor’s test suite consists of annotated code, along with expected output from the given annotation processor. Since the annotation processors utilize all aspects of JSR 308, this serves as an additional end-to-end test of the JSR 308 implementation. As a side benefit, the annotation processors will be useful in their

own right, will thereby illustrate the utility of JSR 308, and will serve as examples for people who wish to create their own type-checking plug-ins.

E.3 Process

JSR 308 follows an unusually open and transparent process. Any interested party may participate, decisions are made by consensus to the greatest extent possible, and discussions are publicly archived. The Java Community Process requires that an expert group formally approve the JSR at each stage. The expert group members will only decide issues on which the group cannot obtain consensus.

When posting, please act professionally and courteously. For example, your arguments should be technical, specific, and based on logic; do not rely on your status or past accomplishments to convince others, and give specific examples rather than vague descriptions. As another example, it is perfectly acceptable to criticize a technical proposal, but do not make personal attacks. Obviously, you should read the specification and, preferably, try the implementation before posting to the mailing list.

F Related work

Section A.2 gave many examples of how type qualifiers have been used in the past. Also see the related work section of [PAC⁺08].

C#’s attributes [ECM06, chap. 24] play the same role as Java’s annotations: they attach metadata to specific parts of a program, and are carried through to the compiled bytecode representation, where they can be accessed via reflection. The syntax is different: C# uses [AnnotationName] or [AnnotationName: data] where Java uses @AnnotationName or @AnnotationName(data); C# uses AttributeUsageAttribute where Java uses Target; and so forth. However, C# permits metadata on generic arguments, and C# permits multiple metadata instances of the same type to appear at a given location.

Like Java, C# does not permit metadata on elements within a method body. (The “[a]C#” language [CCC05], whose name is pronounced “annotated C sharp”, is an extension to C# that permits annotation of statements and code blocks.)

Harmon and Klefstad [HK07] propose a standard for worst-case execution time annotations.

Pechtchanski’s dissertation [Pec03] uses annotations in the aid of dynamic program optimization. Pechtchanski implemented an extension to the Jikes compiler that supports stylized comments, and uses these annotations on classes, fields, methods, formals, local variable declarations, object creation (**new**) expressions, method invocations (calls), and program points (empty statements). The annotations are propagated by the compiler to the class file.

Mathias Ricken’s LAPT-javac (<http://www.cs.rice.edu/~mgricken/research/laptjavac/>) is a version of javac (version 1.5.0.06) that encodes annotations on local variables in the class file, in new `Runtime{Inv,V}isable-LocalVariableAnnotations` attributes. The class file format of LAPT-javac differs from that proposed in this document. Ricken’s xajavac (Extended Annotation Enabled javac) permits subtyping of annotations (<http://www.cs.rice.edu/~mgricken/research/xajavac/>).

The Java Modeling Language, JML [LBR06], is a behavioral modeling language for writing specifications for Java code. It uses stylized comments as annotations, some of which apply to types.

Ownership types [CPN98, Boy04, Cla01, CD02, PNCB06, NVP98, DM05, LM04, LP06] permit programmers to control aliasing and access among objects. Ownership types can be expressed with type annotations and have been applied to program verification [LM04, Mül02, MPHL06], thread synchronization [BLR02, JPLS05], memory management [ACG⁺06, BSBR03], and representation independence [BN02].

JavaCOP [ANMM06] is a framework for implementing pluggable type systems in Java. Whereas JSR 308 uses standard interfaces such as the Tree API and the JSR 269 annotation processing framework, JavaCOP defines its own incompatible variants. A JavaCOP type checker must be programmed in a combination of Java and JavaCOP’s own declarative pattern-matching and rule-based language. JavaCOP’s authors have defined over a dozen type-checkers in their language. The paper does not report that they have run any of

these type-checkers on a real program; this is due to limitations that make JavaCOP impractical (so far) for real use.

JACK makes annotations on array brackets refer to the array, not the elements of the array [MPPD08].

Acknowledgments

Matt Papi designed and implemented the JSR 308 compiler as modifications to Sun’s OpenJDK javac compiler, and contributed to the JSR 308 design.

The members of the JSR 308 mailing list (<https://lists.csail.mit.edu/mailman/listinfo/jsr308/>) provided valuable comments and suggestions. Additional feedback is welcome.

JSR 308 received the Most Innovative Java SE/EE JSR of the Year award in 2007, at the 5th annual JCP Program Awards. JSR 308’s spec leads (Michael Ernst and Alex Buckley) were nominated as Most Outstanding Spec Lead for Java SE/EE in 2008, at the 6th annual JCP Program Awards.

References

- [AAA06] Marwan Abi-Antoun and Jonathan Aldrich. Bringing ownership domains to mainstream Java. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 702–703, Portland, OR, USA, October 24–26, 2006.
- [ACG⁺06] Chris Andrea, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time systems. In *ECOOP 2006 — Object-Oriented Programming, 20th European Conference*, pages 124–147, Nantes, France, July 5–7, 2006.
- [AFKT03] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI 2003, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, CA, USA, June 9–11, 2003.
- [ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 57–74, Portland, OR, USA, October 24–26, 2006.
- [BE04] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.
- [BHP07] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering*, pages 215–229, Braga, Portugal, March 27–30, 2007.
- [Blo04a] Joshua Bloch. JSR 175: A metadata facility for the Java programming language. <http://jcp.org/en/jsr/detail?id=175>, September 30, 2004.
- [Blo04b] Joshua Bloch. JSR175: A program annotation facility for the Java programming language: Proposed final draft 2. <http://jcp.org/en/jsr/detail?id=175>, August 12, 2004.
- [BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 211–230, Seattle, WA, USA, October 28–30, 2002.

- [BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Marseille, France, March 10–13, 2004.
- [BN02] Anindya Banerjee and David A. Naumann. Representation independence, confinement, and access control. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–177, Portland, Oregon, January 16–18, 2002.
- [Boy04] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 2004.
- [Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, Vancouver, BC, Canada, October 2004.
- [BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebe, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI 2003, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 324–337, San Diego, CA, USA, June 9–11, 2003.
- [CCC05] Walter Cazzola, Antonio Cisternino, and Diego Colombo. Freely annotating C#. *Journal of Object Technology*, 4(10):31–48, December 2005. Special Issue: OOPS Track at SAC 2005.
- [CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 292–310, Seattle, WA, USA, October 28–30, 2002.
- [CJ07] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 227–247, Berlin, Germany, August 1–3, 2007.
- [Cla01] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.
- [CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 85–95, Chicago, IL, USA, June 13–15, 2005.
- [CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 48–64, Vancouver, BC, Canada, October 20–22, 1998.
- [Cro06] Jose Cronembold. JSR 198: A standard extension API for Integrated Development Environments. <http://jcp.org/en/jsr/detail?id=198>, May 8, 2006.
- [Dar06] Joe Darcy. JSR 269: Pluggable annotation processing API. <http://jcp.org/en/jsr/detail?id=269>, May 17, 2006. Public review version.
- [Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.
- [DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 2001, Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, USA, June 20–22, 2001.
- [DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

- [DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.
- [ECM06] Ecma 334: C# language specification, 4th edition. ECMA International, June 2006.
- [EFA99] Martin Elsmann, Jeffrey S. Foster, and Alexander Aiken. *Carillon — A System to Find Y2K Problems in C Programs*, July 30, 1999.
- [EM04] Michael Eichberg and Mira Mezini. Alice: Modularization of middleware using aspect-oriented programming. In *4th International Workshop on Software Engineering and Middleware (SEM04)*, pages 47–63, Linz, Austria, December 2004.
- [Eva96] David Evans. Static detection of dynamic memory errors. In *PLDI 1996, Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, USA, May 21–24, 1996.
- [FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA, November 6–8, 2003.
- [FLL⁺02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.
- [FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 17–19, 2002.
- [GF05] David Greenfieldboyce and Jeffrey S. Foster. Type qualifiers for Java. <http://www.cs.umd.edu/Grad/scholarlypapers/papers/greenfieldboyce.pdf>, August 8, 2005.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.
- [GPB⁺06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- [HK07] Trevor Harmon and Raymond Klefstad. Toward a unified standard for worst-case execution time annotations in real-time Java. In *WPDRTS 2007, Fifteenth International Workshop on Parallel and Distributed Real-Time Systems*, Long Beach, CA, USA, March 2007.
- [JPLS05] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Koblenz, Germany, September 7–9, 2005.
- [JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *13th USENIX Security Symposium*, pages 119–134, San Diego, CA, USA, August 11–13, 2004.
- [KT01] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

- [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.
- [LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, pages 491–, Oslo, Norway, June 16–18, 2004.
- [LP06] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–371, Charleston, SC, USA, January 11–13, 2006.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1999.
- [LY07] Tim Lindholm and Frank Yellin. The class file format. http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf, December 2007. Revision to chapter 4 of [LY99] for JDK 1.5.
- [Mor06] Rajiv Mordani. JSR 250: Common annotations for the Java platform. <http://jcp.org/en/jsr/detail?id=250>, May 11, 2006.
- [MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, October 2006.
- [MPPD08] Chris Male, David Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Compiler Construction: 14th International Conference, CC 2008*, pages 229–244, Budapest, Hungary, April 3–4, 2008.
- [Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
- [NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP ’98, the 12th European Conference on Object-Oriented Programming*, pages 158–185, Brussels, Belgium, July 20–24, 1998.
- [PAC⁺08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.
- [PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Mississauga, Ontario, Canada, November 13–16, 2000.
- [Pec03] Igor Pechtchanski. *A Framework for Optimistic Program Optimization*. PhD thesis, New York University, September 2003.
- [Pfe92] Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.
- [PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 311–324, Portland, OR, USA, October 24–26, 2006.
- [PØ95] Jens Palsberg and Peter Ørbæk. Trust in the λ -calculus. In *Proceedings of the Second International Symposium on Static Analysis, SAS ’95*, pages 314–329, Glasgow, UK, September 25–27, 1995.

- [Pug06] William Pugh. JSR 305: Annotations for software defect detection. <http://jcp.org/en/jsr/detail?id=305>, August 29, 2006. JSR Review Ballot version.
- [STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington, DC, USA, August 15–17, 2001.
- [SW01] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP’2001: 3rd Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, June 18, 2001.
- [TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.
- [VS97] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT ’97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 607–621, Lille, France, April 14–18, 1997.
- [Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, December 1998.
- [XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 20–22, 1999.
- [YSP⁺98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September–November 1998.
- [ZPA⁺07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.