Annotations on Java types

JSR 308 working document

Michael D. Ernst
`mernst@csail.mit.edu`

November 5, 2007

The JSR 308 webpage is `http://pag.csail.mit.edu/jsr308/`. It contains the latest version of this document, along with other information such as links to the prototype implementation and sample annotation processors.

1 Introduction

JSR 308 proposes an extension to Java's annotation system [Bra04a] that permits annotations to appear on any use of a type. (By contrast, Java SE 6 permits annotations to appear only on class/method/field/variable declarations; JSR 308 is backward-compatible and continues to permit those annotations.) Such a generalization removes arbitrary 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 [Bra04a], 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 JSR 305, "Annotations for Software Defect Detection" [Pug06]. (For a comparison of JSR 305 and JSR 308, see Section D.4.3, page 25.)

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 [Bra04a].

This document has two parts: a normative part and a 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); in all, it is less than 8 pages.

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).

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 two exceptions that are necessitated by non-orthogonality in the Java grammar.

1. A type annotation appears before the type, as in `@NonNull String`.
2. An annotation on the type of a method receiver (`this`) appears just before the `throws` clause — i.e., after the parameter list.
3. Annotations on the top level of an array follow the first rule and appear before the array type. Annotations on types of array elements (i.e., on other than the top level of the array) appear within the brackets `[]` that indicate the levels of the array.

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. 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* | `;` *)*

3. 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

2.3 Target meta-annotation for type annotations

Java uses the `@Target` meta-annotation as a machine-checked way of expressing where an annotation is intended to appear. JSR 308 uses `ElementType.TYPEREF` to indicate a type annotation:

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

An annotation that is meta-annotated with `@Target(ElementType.TYPEREF)` may appear on any use of a type. `ElementType.TYPEREF` is new in JSR 308, and is distinct from the existing `ElementType.TYPE` enum element of Java SE 6, which indicates that an annotation may appear on a type 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.

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.

When creating 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]), annotations should be copied from the method being invoked. (As of Java SE 6, javac does not copy/transfer any annotations from original methods to the bridge methods; that is probably a bug in javac.)

4 Class file format extensions

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] needs to read annotations when compiling a client of the class file. 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. (A third *non*-goal is providing reflective access within method bodies.)

This document proposes conventions for storing the annotations described in Section 2, as well as for storing local variable annotations, which are permitted in Java syntax but currently discarded by the compiler. Class files already store annotations in the form of "attributes" [Bra04a, LY]. JVMs ignore unknown attributes. 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. JSR 308 does not alter the way that existing annotations on classes, methods, method parameters, and fields are stored in the class file. Class files generated from programs that use no new annotations will be identical to those generated by a standard Java SE 6 (that is, pre-extended-annotations) 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).

In Java SE 6, annotations are stored in the class file in attributes of the classes, fields, or methods they target. Attributes are sections of the class file that associate data with a program element (a method's bytecodes, for instance, are stored in a `Code` attribute). The `RuntimeVisibleAnnotations` attribute is used for annotations that are accessible at runtime using reflection, and the `RuntimeInvisibleAnnotations` attribute is used for annotations that are not accessible at runtime. 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.

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, `RuntimeVisible-TypeAnnotations` and `RuntimeInvisibleTypeAnnotations` contain an array of `extended_annotation` elements, which are described in Section 4.1 below.

The `Runtime[In]visibleTypeAnnotations` attributes store annotations written in the new locations described in Section 2, and on local variables. For annotations on the type of a field, the `field_info` structure (see JVMS3 §4.6) corresponding to that field stores the `Runtime[In]visibleTypeAnnotations` attributes. For annotations on types in method signatures or bodies, the `method_info` structure (see JVMS3 §4.7) that corresponds to the annotations' containing method stores the `Runtime[In]visibleTypeAnnotations` attributes. For annotations on class type parameter bounds and class extends/implements types, the `attributes` structure (see JVMS3 §4.2) stores the `Runtime[In]visibleTypeAnnotations` attributes.

4.1 The `extended_annotation` structure

The `extended_annotation` structure has the following format, which adds `target_type` and `reference_info` to the `annotation` structure defined in JVMS3 §4.8.15:

```
extended_annotation {
    u2 type_index;
    u2 num_element_value_pairs;
    {
        u2 element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs];
    u1 target_type;    // new in JSR 308: where the annotation appears
    {
        ...
    } reference_info;  // new in JSR 308: where the annotation appears
}
```

We briefly recap the fields of `annotation`, which are described in in JVMS3 §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; for details, see JVMS 3 §4.8.15.1.

The following sections describe the fields of the `extended_annotation` structure that differ from `annotation`.

4.1.1 The `target_type` field

The `target_type` field denotes the type of program element that the annotation targets. As described above, annotations in any of the following locations are written to `Runtime[In]visibleTypeAnnotations` attributes in the class file:

- on typecasts, type tests, object creations, local variables, bounds of type parameters of classes and methods, `extends` and `implements` clauses of class declarations, and `throws` clauses of method declarations;
- on a type argument or array type of any of the above;
- on method receivers;
- on a type argument or array type of a field, method, or method parameter.

The corresponding values for each of these cases are shown in Figure 1. Some locations are assigned numbers even though annotations in those locations are prohibited or are actually written to `Runtime[In]visibleAnnotations` or `Runtime[In]visibleParameterAnnotations`. While those locations will never appear in a `target_type` field, including them in the enumeration may be convenient for software that processes extended annotations. They are marked $*$ in Figure 1.

4.1.2 The `reference_info` field

The `reference_info` field is used to reference the annotation's target in bytecode. The contents of the `reference_info` field is determined by the value of `target_type`.

TODO: The `reference_info` attribute field (for local variables) should be a list of PC ranges, rather than a single one, to accommodate compiler optimizations or other code reordering.

Annotation Target	`target_type` **Value**
typecast	`0x00`
typecast generic/array	`0x01`
type test (`instanceof`)	`0x02`
type test (`instanceof`) generic/array	`0x03`
object creation (`new`)	`0x04`
object creation (`new`) generic/array	`0x05`
method receiver	`0x06`
method receiver generic/array	`0x07`∗
local variable	`0x08`
local variable generic/array	`0x09`
method return type	`0x0A`∗
method return type generic/array	`0x0B`
method parameter	`0x0C`∗
method parameter generic/array	`0x0D`
field	`0x0E`∗
field generic/array	`0x0F`
class type parameter bound	`0x10`
class type parameter bound generic/array	`0x11`
method type parameter bound	`0x12`
method type parameter bound generic/array	`0x13`
class `extends`/`implements`	`0x14`
class `extends`/`implements` generic/array	`0x15`
exception type in `throws`	`0x16`
exception type in `throws` generic/array	`0x17`∗
type argument in constructor call	`0x18`
type argument in constructor call generic/array	`0x19`
type argument in method call	`0x1A`
type argument in method call generic/array	`0x1B`
wildcard bound	`0x1C`
wildcard bound generic/array	`0x1D`
class literal	`0x1E`
class literal generic/array	`0x1F`∗
method type parameter	`0x20`
method type parameter generic/array	`0x21`∗

Figure 1: Values of `target_type` for each possible target of a type annotation. Enumeration elements marked ∗ will never appear in a `target_type` field but are included for completeness and convenience for annotation processors. 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.

Typecasts, type tests, and object creation When the annotation's target is a typecast, an `instanceof` expression, or a `new` 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 argument 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 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 the end of this section (page 7) for handling of generic type arguments and arrays.

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 index in that method. These fields are similar to those of the optional `LocalVariableTable` attribute defined in JVMS3 §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).

Method Receivers When the annotation's target is a method receiver, `reference_info` is empty.

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.

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 ith superinterface type.

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.

`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.

Generic Type Arguments or 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.

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*

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 [Bra04b] 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://pag.csail.mit.edu/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 technical report [PAJ+07] 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. `@Readonly`'s immutability guarantee can help developers avoid

```
 1 @NonNullDefault
 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 `@NonNullDefault` 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 argument (a `Vertex`) or its receiver (a `DAG`). 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 `==`.

accidental modifications, which are often manifested as run-time errors. An immutability annotation can also improve performance. For example, a programmer can indicate that a particular method (or all methods) on an Enterprise JavaBean is readonly, using the Access Intents mechanism of WebSphere Application Server.

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]. 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.). An `interned` qualifier can indicate which objects have been converted to canonical form and thus may be compared via object 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 [Goe06] 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. 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].

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. 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, @NonEmpty List<@Readonly Document>> files;
  ```
- for generic type arguments in a generic method or constructor invocation:
  ```
  o.<@NonNull String>m("...");
  ```
- for type parameter bounds and wildcards:
  ```
  class Folder<F extends @Existing File> { ... }
  Collection<? super @Existing File>
  ```
- for class inheritance:
  ```
  class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }
  ```
- for `throws` clauses:
  ```
  void monitorTemperature() throws @Critical TemperatureException { ... }
  ```
- 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.4.1.
- 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.4.1.
- for object creation:
  ```
  new @NonEmpty @Readonly List<String>(myNonEmptyStringSet)
  ```
  For generic constructors (JLS §8.8.4), the annotation follows the explicit type arguments (JLS §15.9):
  ```
  new <String> @Interned MyObject()
  ```
- for method receivers:
  ```
  public String toString() @Readonly { ... }
  public void write() @Writable throws IOException { ... }
  ```
  A method can express constraints on the generic parameters of the receiver (just as is possible for other formal parameters, albeit with a slightly different syntax):
  ```
  public int size() @Readonly<@Readonly> { ... }
  public void requiresNonNullKeys() <@NonNull,> { ... }
  ```
- for class literals:
  ```
  Class<@NonNull String> c = @NonNull String.class;
  ```
- for static member access:
  ```
  @NonNull Type.field
  ```
- for arrays:
  ```
  Document[@Readonly][] docs4 = new Document[@Readonly 2][12];
  Document[][@Readonly] docs5 = new Document[2][@Readonly 12];
  ```
  This syntax permits independent annotations for each distinct level of array, and for the elements.

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.1, 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 an array `a` with non-null elements.
- 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, PAJ+07]. Array elements can often be null (both due to initialization, and for other reasons), necessitating annotations on them.

Receivers Method receivers (`this`) are formal parameters and thus are an implicit mention of a type. For example, the method `PrintStream.println(String)` has two formal parameters (and at run time, its invocation involves two actual arguments). In Java's syntax, one of the formal parameters (the receiver) is implicit, but for consistency and expressiveness the implicit use of the receiver type should be annotatable just as the explicit parameters are. Such annotations require new syntax to distinguish them from annotations on the return value.

For example, this receiver annotation

```
Dimension getSize() @Readonly { ... }
```

indicates that `getSize` does not modify its receiver. This 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). This is also different than saying that a client should not modify the return value, so it is not appropriate as a return value annotation.

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.

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.4.1.)

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 error 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 T) {
  ... (T) x ...
}

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

to be used with the same annotated type `T` 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 Annotations on object creation (`new`) can indicate the type of the newly-created object, which could be statically (at compile time) verified 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.4.4 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 element type and each distinct level of a nested array. Forbidding annotations on arbitrary levels of an array would simplify the annotation system, though it would reduce expressiveness. The syntax of array types is rather different than the syntax of other Java types, so the annotation syntax must also be different. (Arrays are not very commonly used in Java, so perhaps the syntax need not be perfect, so long as it is usable and expressive.)

This section presents several proposals for array syntax.

For the array syntax, there are two choices to make. First, should an annotation on a set of brackets refer to the array (ARRAY) or the elements (ELTS)? Second, where should array annotations appear?

- IN: within the brackets (`[]`) of the array syntax: `@NonNull Document[@Readonly]`
- PRE: outside the brackets in prefix notation (before the brackets): `@NonNull Document @Readonly []`
- POST: outside the brackets in postfix notation (after the brackets): `@NonNull Document[] @Readonly`
  or, if postfix syntax is adopted for all type annotations: `Document @NonNull [] @Readonly`

Here is an example of the ARRAY-vs-ELTS distinction. Taking the IN syntax as an example, should `@NonNull Document[@Readonly]` mean that the array is `@Readonly` and contains `@NonNull` elements (ARRAY-IN), or that the array is `@NonNull` and contains `@Readonly` elements (ELTS-IN)? (For the fully postfix syntax, the ARRAY-vs-ELTS question is moot: the only sensible choice is for the annotation on the brackets to refer to the array, not the elements.)

Here are some (mutually incompatible) principles that an ideal syntax would satisfy.

P1 Adding array levels should not change the meaning of existing annotations. For example, it would be confusing to have a syntax in which

```
@A List<@B Object>        // @A refers to List
@A List<@B Object>[@C]    // @A refers to array, @C refers to List
```

Another way of stating this principle is that a textual subpart of a declaration should describe a type that is part of the declared type. Stating a subpart of the given type should not require shuffling around the annotations.

P2 When two variables appear in a variable declaration, the annotations should mean the same thing for both variables. In Java, arrays can be declared with brackets after the type, after the identifier, or both, as in `String[] my2dArray[];`. For example, `arr1` should have the same annotations as the elements of `arr2`:

```
@A T[@B] arr1, arr2[@C];
```

Likewise, the `T`s should have the same annotations for `v3` and `arr4`:

```
@A T v3, arr4[@B][@C];
```

And, these three declarations should mean the same thing:

```
@A T[@B] arr5[@C];
@A T[@B][@C] arr6;
@A T arr7[@B][@C];
```

P3 Type annotations before a declaration should refer to the full type, just as variable annotations (which occur in the same position — at the very beginning of the declaration) refer to the entire variable. This is also consistent with annotations on generics (though the syntax of generics and arrays is quite different in other ways), where `@NonNull List<String>` is a non-null `List` of possibly-null `String`s. This principle is inconsistent with principles P1 and P2, unless type annotations are forbidden before a declaration.

The ARRAY syntax (an annotation on brackets refers to the array) violates principle P3. The ELTS syntax (an annotation on brackets refers to the elements) violates principles P1 and P2.

Here are several proposals for the syntax of such array annotations.

The examples below use the following variables:

`array_of_rodocs` a mutable one-dimensional array of immutable `Documents`

`roarray_of_docs` an immutable one-dimensional array of mutable `Documents`

`array_of_array_of_rodocs` a mutable array, whose elements are mutable one-dimensional arrays of immutable `Documents`

`array_of_roarray_of_docs` a mutable array, whose elements are immutable one-dimensional arrays of mutable `Documents`

`roarray_of_array_of_docs` an immutable array, whose elements are mutable one-dimensional arrays of mutable `Documents`

ARRAY-IN: Within brackets, refer to the array being accessed

An annotation before the entire array type binds to the member type that it abuts; `@Readonly Document[][]` can be interpreted as `(@Readonly Document)[][]`.

An annotation within brackets refers to the *array* that is accessed using those brackets.

The type of elements of `@A Object[@B][@C]` is `@A Object[@C]`.

The example variables would be declared as follows:

```
@Readonly Document[] array_of_rodocs;
Document[@Readonly] roarray_of_docs;
@Readonly Document[][] array_of_array_of_rodocs = new Document[2][12];
Document[@Readonly][] array_of_roarray_of_docs = new Document[@Readonly 2][12];
Document[][@Readonly] roarray_of_array_of_docs = new Document[2][@Readonly 12];
```

ELTS-IN: Within brackets, refer to the elements being accessed

An annotation before the entire array type refers to the (reference to the) top-level array itself; `@Readonly Document[][] docs4` indicates that the array is non-modifiable (not that the `Documents` in it are non-modifiable).

An annotation within brackets applies to the *elements* that are accessed using those brackets.

The type of elements of `@A Object[@B][@C]` is `@B Object[@C]`.

The example variables would be declared as follows:

```
Document[@Readonly] array_of_rodocs;
@Readonly Document[] roarray_of_docs;
Document[][@Readonly] array_of_array_of_rodocs = new Document[2][@Readonly 12];
Document[@Readonly][] array_of_roarray_of_docs = new Document[@Readonly 2][12];
@Readonly Document[][] roarray_of_array_of_docs = new Document[2][12];
```

ARRAY-PRE: Outside brackets, prefix; refer to the array being accessed

The type of elements of `@A Object @B [] @C []` is `@A Object @C []`.

The example variables would be declared as follows:

```
@Readonly Document[] array_of_rodocs;
Document @Readonly [] roarray_of_docs;
@Readonly Document[][] array_of_array_of_rodocs = new Document[2][12];
Document [] @Readonly [] array_of_roarray_of_docs = new Document[2] @Readonly [12];
Document @Readonly [][] roarray_of_array_of_docs = new Document @Readonly [2][12];
```

ELTS-PRE: Outside brackets, prefix; refer to elements

The type of elements of `@A Object @B [] @C []` is `@B Object @C []`.

The example variables would be declared as follows:

```
Document @Readonly [] array_of_rodocs;
@Readonly Document[] roarray_of_docs;
@Readonly Document[][] array_of_array_of_rodocs = new Document[2][12];
Document[] @Readonly [] array_of_roarray_of_docs = new Document[2] @Readonly [12];
Document @Readonly [][] roarray_of_array_of_docs = new Document @Readonly [2][12];
```

ARRAY-POST: Outside brackets, postfix; refer to the array being accessed

The type of elements of `@A Object [] @B [] @C` is `@A Object [] @C`.

The example variables would be declared as follows:

```
@Readonly Document[] array_of_rodocs;
Document [] @Readonly roarray_of_docs;
@Readonly Document[][] array_of_array_of_rodocs = new Document[2][12];
Document [] @Readonly [] roarray_of_array_of_docs = new Document[2] @Readonly [12];
Document [][] @Readonly array_of_roarray_of_docs = new Document[2][12] @Readonly;
```

ELTS-POST: Outside brackets, postfix; refer to elements

The type of elements of `@A Object[] @B [] @C` is `@B Object[] @C`.

In Java, array types are constructed using postfix syntax, so postfix annotation syntax for them is appealing.

Possible disadvantage: Prefix notation may be more natural to Java programmers, as it is used in other places in the Java syntax.

The example variables would be declared as follows:

```
Document[] @Readonly array_of_rodocs;
@Readonly Document[] roarray_of_docs;
Document[][] @Readonly array_of_array_of_rodocs = new Document[2][12] @Readonly;
Document[] @Readonly [] array_of_roarray_of_docs = new Document[2] @Readonly [12];
@Readonly Document[][] roarray_of_array_of_docs = new Document[2][12];
```

or, in a fully postfix syntax:

```
Document @Readonly [] array_of_rodocs;
Document[] @Readonly roarray_of_docs;
Document @Readonly [][] array_of_array_of_rodocs = new Document[2][12] @Readonly;
Document[] @Readonly [] array_of_roarray_of_docs = new Document[2] @Readonly [12];
Document[][] @Readonly roarray_of_array_of_docs = new Document[2][12];
```

The IN (within-the-brackets) syntax has problems with ambiguity, when an explicit size is provided in a `new` array construction expression. In this example the annotated element could be the array or the type `Y`:

```
new X[@ReadOnly Y.class.getMethods().length]
```

And in this example, the annotated element is the array, but the annotation could be the marker annotation `@ReadOnly` with a parenthesized expression `(2)` or could be the annotation `@ReadOnly(2)`.

```
new X[2][ @ReadOnly (2) ]
```

It is also possible to imagine array annotations that do not require new locations for the annotations. The advantage of this is that there is no new syntax. A disadvantage is that the array level annotations are syntactically separated from the array levels themselves, so the meaning may not be as clear.

1. Use an array-valued annotation that gives the annotations for the different dimensions. It could give the dimensions explicitly:

   ```
   // dimension 1 and 2 of the array are annotated
   @ArrayAnnots({
     @ArrayAnnot(i=1, value={Readonly.class}),
     @ArrayAnnot(i=2, value={Readonly.class})
   })
   Object[][][] arr;
   ```

   or use the order in which the annotations are given.

   ```
   @ArrayAnnots({
     @ArrayAnnot({Readonly.class}),
     @ArrayAnnot({Readonly.class})
   })
   Object[][] arr2;

   @ArrayAnnots({
     @Readonly,
     @Readonly
   })
   Object[][] arr2;
   ```

   The latter syntax is less convenient when not every level of the array is being annotated, or when multiple annotations are put on an array. (This document should give examples of those situations.)

2. Use an annotation that lists the dimensions that have a property:

   ```
   // In each case, the elements in the array are readonly
   // dimension 0 has no annotation
   // dimensions 1 and 2 are also readonly

   @ReadonlyDims({1,2}) @Readonly Object[][][] roa;
   @Dims({1,2}, @Readonly) @Readonly Object[][][] roa;
   ```

   One advantage of this syntax over the one that gives an array of annotations is that each annotation is given independently, so it will be easier for tools to insert, delete, or conditionally display a given annotation. However, the array of annotations more closely mirrors the syntax of the array declaration itself.

B.4 Disambiguating type and declaration annotations

An annotation before a method declaration annotates either the return type, or the method declaration; similarly for field declarations. The `@Target` meta-annotation indicates the programmer intention.

Consider the following two field declarations.

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

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

As another example, in

```
@Override
@NonNull Dimension getSize() { ... }
```

`@Override` applies to the method and `@NonNull` applies to the return type. This is because `Override` is meta-annotated with `ElementType.METHOD`, and `NonNull` is meta-annotated with `ElementType.TYPEREF` (see Section 2.3).

As explained in Section 2.3, the compiler applies the annotation to every target that is consistent with its meta-annotation. This means that, for certain syntactic locations, which target (Java construct) is being annotated depends on the annotation, or an annotation might even be applied to two targets.

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.TYPEREF)` (since `ElementType.TYPEREF` does not exist in Java SE 5/6), but with no meta-annotation or with one that permits annotations on any declaration.

C.2 Annotation processing

The Tree API, which exposes the AST (including annotations) to authors of annotation processors (compile-time plug-ins), must be updated to reflect the modifications made to the internal AST node classes described in Section 2.

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 annotations on class member declarations (also see Section D.4.6). 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.

For example, new method `Method.getReceiverAnnotation` (for the receiver `this`) would parallel the existing `Method.getAnnotations` (for the return value) and `Method.getParameterAnnotations` (for the formal parameters). 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.

Suppose that a method is declared as:

```
@NonEmpty List<@Interned String> foo(@NonNull List<@Opened File> files) @Readonly {...}
```

Then `Method.getAnnotations()` returns the `@NonEmpty` annotation, just as in Java SE 6, and likewise `Method-.getParameterAnnotations()` returns the `@NonNull` annotation. New method `Method.getReceiverAnnotations()` returns the `@Readonly` annotation. We have not yet decided how to provide reflective access to annotations on generic types in a method's signature, such as the instances of `@Interned` and `@Opened` above.

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

C.4 Virtual machine and class file analysis tools

No modifications to the virtual machine are necessary.

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://pag.csail.mit.edu/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://pag.csail.mit.edu/jsr308/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

JSR 308 "Annotations on Java Types" [EC06] has the goal of refining the ideas presented here. This proposal serves as a starting point for the JSR 308 expert group, but the expert group has the freedom to modify this proposal or to explore other approaches. (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`.)

The Expert Group will consider whether the proposal should extend annotations in a few other ways that are not directly related to annotations on types. This is especially true if the additional changes are small, that there is no better time to add such an annotation, and the new syntax would permit unanticipated future uses. Two examples follow, for which the proposal does not currently include a detailed design. Then, the rest of this section presents extensions that are out of the scope of JSR 308.

D.1 Duplicate annotations at a location

Currently, array-valued annotations can be clumsy to write:

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

Likewise, it may be desirable for some (but not all) annotations to be specified more than once at a single location, but "It is a compile-time error if a declaration is annotated with more than one annotation for a given annotation type." [GJSB05, §9.7]. (C# supports multiple annotations on a given program element.)

A cleaner syntax may be 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.

1. Use a meta-annotation that declares the type of the container, and to desugar duplicate annotations into the current array syntax. This approach treats duplicate annotations as purely a syntactic convenience; it does not change annotations in any deep way.

   One problem with this proposal is that it loses the ordering differently-named annotations. For example, it cannot distinguish these declarations:

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

2. Add new methods that return multiple annotations. Each method of the form

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

   would be augmented by one of the form

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

   No other changes would be necessary.

D.2 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; join the `jsr308-statements@lists.csail.mit.edu` mailing list (via `https://lists.csail.mit.edu/mailman/listinfo/jsr308-statements/`) to participate in discussions of the proposal.

D.3 Alternative annotation syntaxes

Sections 5 and 2 describe a simple prefix syntax for annotations on types. Alternatives are possible, and this section notes some possibilities.

D.3.1 Postfix syntax for type annotations

The current proposal uses a simple prefix syntax for type annotations: the annotation appears before the type, as in `@NonNull String`. There are two exceptions to this general rule: the syntax for arrays and the syntax for method receivers.

An alternative would use a simple *postfix* syntax for type annotations: type annotations would appear *after* the type, as in `String @NonNull` or `List <String> @NonNull`. This syntactically separates type annotations from all other annotations, putting them in a different place in the syntax. In

```
@A Type @B var;
```

`@A` would refer to the variable and `@B` would refer to the type.

A summary of grammar changes are the following additions:

Type:
    Type Annotation
Statement:
    Annotation Statement
VariableDeclaratorRest:
    Annotation VariableDeclaratorRest
MethodOrFieldRest:
    MethodOrFieldRest Annotation

Plus the following extra rules: When an annotation appears at the beginning of a declaration (using the existing syntax) and in one of these new contexts in the same declaration, it is an error. In that case the programmer must migrate the annotation to the new syntax. This prevents annotations from appearing "out of order". For instance, a construct such as `@A Type @B var;` is illegal; in this construct, the annotations appear "out of order", in that `@A` refers to `var` and `@B` refers to `Type`.

The intention of these rules is to permit/forbid the following syntactic forms:

```
@Deprecated @NonNull List<String> getStrings1() { ... }  // legal
List<String> @NonNull getStrings2() @Deprecated { ... }  // legal
@Deprecated List<String> @NonNull getStrings3() { ... }  // illegal
@NonNull List<String> getStrings4() @Deprecated { ... }  // illegal
```

The partial proposal sketched here needs to be extended to handle annotations on the receiver type, on varargs, and possibly other locations.

Advantages: Postfix syntax reduces the number of special cases in the syntax from 2 to 1: no special case is needed for arrays, but one is still needed for receiver types. This may be more convenient for compiler writers. It may also be less confusing to programmers — though their impression of simplicity may be affected by the fact that it introduces more new annotation locations in a program than the prefix syntax does.

Disadvantages: Java is a generally prefix language with respect to modifiers, so postfix notations may offer consistency and simplicity problems; real use is required to determine whether such problems are speculative or real. JSR 305 [Pug06] proposes that type annotations, such as `@NonNull`, be written in prefix style (on the declaration or variable, in the absence of JSR 308's extensions); the postfix syntax would require such code to be rewritten (or would permit type annotations to appear in either postfix or in prefix location, which might lead to inconsistency, confusion, or complications for tools), and likewise for statement annotations. As a different (and less important) but related issue, in a construct such as `List <String> @NonNull`, some programmers report that it looks like `@NonNull` is associated with `String` rather than with `List`.

D.3.2 Generics-like syntax for type annotations

Place annotations in angle brackets after the type being annotated, just as type arguments are (either all after or all before the type arguments):

```
// Choose one of the following TypeArguments productions
TypeArgumentsAnnotationsLast:
       < [TypeArgument {, TypeArgument}] [; Annotations] >
       < Annotations >
TypeArgumentsAnnotationsFirst::
       < [Annotations ;] TypeArgument {, TypeArgument} >
       < Annotations >
BasicType:
     RawBasicType [<Annotations>]
RawBasicType:
     ...   // current content of the BasicType production
```

Here is how examples from JSR-308 document would look in such a syntax (with both annotations-first (Afirst) and annotations-last (Alast) alternatives shown).

```
Map<String<@NonNull>, List<@NonEmpty; Document<@Readonly>>> files; // Afirst
Map<String<@NonNull>, List<Document<@Readonly>; @NonEmpty>> files; // Alast

o.<String<@NonNull>>m("...");

class Folder<F extends File<@Existing>> { ... }
Collection<? super File<@Existing>>

class UnmodifiableList<T> implements List<@Readonly; T<@Readonly>> { ... } // Afirst
class UnmodifiableList<T> implements List<T<@Readonly>; @Readonly> { ... } // Alast

void monitorTemperature() throws TemperatureException<@Critical> { ... }

myString = (String<@NonNull>) myObject;

boolean isNonNull = myString instanceof String<@NonNull>;

new List<@NonEmpty; @Readonly; String>(myNonEmptyStringSet) // Afirst
new List<String; @NonEmpty; @Readonly>(myNonEmptyStringSet) // Alast
new <@Interned; String> MyObject() // Afirst
new <String; @Interned> MyObject() // Alast

public <@Readonly> int size() { ... }  // method receiver

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

The IGJ type system [ZPA+07] has been implemented using both a generics-like syntax and also the JSR 308 annotation syntax. In a case study, a programmer preferred the JSR 308 syntax to the generics-like syntax [ZPA+07].

D.4 Out-of-scope issues

This section of the document discusses several issues that are not in scope for JSR 308.

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 [Bra04a]) did. It is a goal not to unnecessarily close off realistic future avenues of extension.

D.4.1 Locations for annotations

Expression annotations. Annotating a type cast indicates a property of a value (the result of an expression). This is different than annotating the expression itself, which indicates some property of the entire 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.)

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 @codex instanceof @NonNull), for several reasons. Erasing the annotations should leave a valid Java program. Stating the type reinforces that the annotation is a type annotation rather than an expression annotation. The benefit of omitting the type in the cast seems relatively minor. Especially in a type test, stating the type reinforces that the run-time effect has is to check and change the Java type; no run-time check of the annotation is possible in general.

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).

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.4.2 Changes to the annotation type

Subclassing annotations. Annotations cannot subclass one another, so it is difficult to share behavior or to express similarities or relationships among annotation types. (To work around this, 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.)

Subannotations raise a trust problem. Suppose annotation `@A` is tied to a framework. If someone creates `@B`, a subclass of annotation `@A`, then by the Liskov Substitution Principle, `@B` must function as `@A`. But the framework will not want to load the subclass `@B` into its VM, as `@B` is alien and untrusted code from the framework's viewpoint.

A more prosaic problem with subclassing is the limitation of one annotation of a given type per location (see Section D.1). Allowing subtyping among annotations requires solving that problem, and in particular coming up with reasonable semantics for the situation where you annotate with two subtypes of a given annotation type, and then try to read the annotation of the parent type.

Annotations as arguments to annotations In Java, it is not possible to define an annotation that takes an arbitrary annotation as a parameter, as in

```
@DefaultAnnotation(@MyAnnotation)
```

More generally, annotation types cannot have members of their own type. (An annotation whose parameter is an annotation of a specific type is explicitly permitted (JLS §9.6 and §9.7).)

These limitations reduce the expressiveness of annotations. 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 over types, as generics do). It is impossible to define annotations with recursive structure. It is inconvenient to define annotations with choices in their structure: a discriminated union can be simulated via field names that act as explicit tags.

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.

D.4.3 Semantics of annotations

Annotations for specific purposes. JSR 308 does not define any annotations nor take any position on their semantics. 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 set of annotations for specific purposes, along with their semantics. 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. For more details, see `http://jcp.org/en/jsr/detail?id=305` and `http://groups.google.com/group/jsr-305/`.

Annotation inheritance. 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 provide a more fine-grained mechanism that applies different rules to classes, methods, fields, etc., or even to specify inheritance of annotations from interfaces. These semantic issue are out of the scope of JSR 308 but may be taken up by JSR 305 ("Annotations for Software Defect Detection" [Pug06]).

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 @NonNullDefault 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 class literal, as the argument to `@DefaultAnnotation`, as in

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

but in Java, it is not possible to define an annotation that takes an arbitrary annotation as a parameter; see Section D.4.2.

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.4.4 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 14 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 [Goe06] 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 increase code readability.

D.4.5 Class file syntax

Changes to the class file syntax are out of the scope of JSR 308, which, for backward compatibility, does not change the way that existing annotations are stored in the class file.

However, some changes to the class file syntax have significant benefits, and could be the subject of another, small, JSR whose focus is only the class file format. Class file syntax changes require modification of compilers, JVMs, javap, and other class file tools (see Sections C.4 and C.5).

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 with the annotations enabled by JSR 308 and those proposed in JSR 305.

D.4.6 Annotation processing API

The JSR 269 annotation processing API specifies that the `process` method is invoked on class, field, and method annotations. It does not process annotations on local variables, as it is not designed to access method bodies. JSR 269's limitations make it insufficient for a type-checking compiler plug-in (annotation processor), which must both process all annotations and also check at each use of a variable/method whose declaration 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. Extending JSR 269 to process all annotations, for consistency and to support other types of annotation processors, is beyond the scope of JSR 308 but may be desirable in the future, after more experience is gained with JSR 308 annotation processors.

E Logistical matters

JSR 308 ("Annotations on Java types") 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 prototype 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 prototype implementation recognizes the extended annotations when surrounded by comment markers:

```
/*@Readonly*/ Object x;
```

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, and the final Java SE 7 implementation will not 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] 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): 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 revisions to chapter 4, "The class file Format".) Similar definitions are required for RuntimeV,InvisibleTypeAnnotations.

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.

F Related work

Section A.1 gave many examples of how type qualifiers have been used in the past. Also see the related work section of [PAJ$^+$07].

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}isible-LocalVariableAnnotations` attributes. The class file format of LAPT-javac differs from that proposed in this document.

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. They have not run any of these type-checkers on a program, due to limitations that make JavaCOP impractical for real use.

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.

At the 5th annual JCP Program Awards (in May 2007), JSR 308 received the Most Innovative Java SE/EE JSR of the Year award.

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 *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.

[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.

[Bra04a] Gilad Bracha. JSR 175: A metadata facility for the Java programming language. `http://jcp.org/en/jsr/detail?id=175`, September 30, 2004.

[Bra04b] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, Vancouver, BC, Canada, October 2004.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *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, 20th 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 *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.

[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 *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 (JOT)*, 4(8):5–32, October 2005.

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

[ECM06] Ecma 334: C# language specification, 4th edition. ECMA International, June 2006.

[EFA99] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. *Carillon — A System to Find Y2K Problems in C Programs*, July 30, 1999.

[Eva96] David Evans. Static detection of dynamic memory errors. In *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.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *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/greenfiledboyce.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.

[Goe06] Brian Goetz. The pseudo-typedef antipattern: Extension is not type definition. `http://www.ibm.com/developerworks/library/j-jtp02216.html`, February 21, 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.

[LY] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. 3rd edition. To appear.

[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.

[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.

[PAJ$^+$07] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical Report MIT-CSAIL-TR-2007-047, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 17, 2007.

[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 Kieżun, 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*, Dubrovnik, Croatia, September 5–7, 2007.