Type Annotations Specification (JSR 308)

Michael D. Ernst

`mernst@cs.washington.edu`

January 23, 2012

The JSR 308 webpage is `http://types.cs.washington.edu/jsr308/`. It contains the latest version of this document, along with other information such as a FAQ, the reference implementation, and sample annotation processors.

Contents

1   Introduction

JSR 308 extends Java's annotation system [Blo04] so that annotations may appear on any use of a type. (By contrast, Java SE 7 permits annotations only on declarations; JSR 308 is backward-compatible and continues to permit those annotations.) Such a generalization removes limitations of Java's annotation system, and it enables new uses of annotations.

This document specifies the *syntax* of extended Java annotations, but it makes no commitment as to their *semantics*. As with Java's existing annotations [Blo04], the semantics is dependent on annotation processors (compiler plug-ins), and not every annotation is necessarily sensible in every location where it is syntactically permitted to appear. This proposal is compatible with existing annotations, such as those specified in JSR 250, "Common Annotations for the Java Platform" [Mor06], and proposed annotations, such as those to be specified in (the now-defunct) JSR 305, "Annotations for Software Defect Detection" [Pug06].

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 [Blo04].

This document has two parts: a normative part and a non-normative part. The normative part specifies the changes to the Java language syntax (Section 2) and the class file format (Section 3). The non-normative part motivates annotations on types by presenting one possible use, type qualifiers (Appendix A). It also lists tools that must be updated to accommodate the Java and class file modifications (Appendix C). The document concludes with related work (Section D).

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

A FAQ (Frequently Asked Questions) document complements this specification; see `http://types.cs.washington.edu/jsr308/jsr308-faq.html`.

2   Java language syntax extensions

2.1   Source locations for annotations on types

In Java SE 7, 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, and on type parameter declarations. JSR 308 does not extend Java to allow annotations on type names that are not type uses, such as the type names that appear in `import` statements, class literals, static member accesses, and annotation uses (see Section 2.1.1). JSR 308 uses a simple prefix syntax for type annotations, as illustrated in the below examples, with a special case for receiver types (item #6 below). In addition to supporting type annotations, JSR 308 also makes one extension to declaration annotations (item #7 below).

1. A type annotation appears before the type, as in `@NonNull String` or `@NonNull java.lang.String`. A type annotation appears before the package name, if any. Here are examples:

   - 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, including wildcard bounds:
     ```
     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;`.
   - for constructor invocation results (that is, for object creation):
     ```
     new @Interned MyObject()
     new @NonEmpty @Readonly List<String>(myNonEmptyStringSet)
     myVar . new @Tainted NestedClass
     ```
     For generic constructors (JLS §8.8.4), the annotation follows the explicit type arguments (JLS §15.9):
     ```
     new <String> @Interned MyObject()
     ```
   - for type tests:
     ```
     boolean isNonNull = myString instanceof @NonNull String;
     ```
     It is not permitted to omit the Java type, as in `myString instanceof @NonNull`.

2. For a nested type, the type annotation appears before the type's simple name, as in `Outer. @NonNull StaticNested`; a concrete example is `Map.@NonNull Entry`.

3. An annotation on a wildcard type argument appears before the wildcard, as in `MyClass<@Immutable ? extends Comparable<MyClass>>`.

4. The annotation on a given array level prefixes the brackets that introduce that level. To declare a non-empty array of English-language strings, write `@English String @NonEmpty []`. The varargs syntax "..." is treated analogously to array brackets and may also be prefixed by an annotation. Here are examples:

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

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

5. A type annotation is permitted in front of a constructor declaration, where declaration annotations are already permitted. In that location, a type annotation is treated as applying to the constructed object (which is different than the receiver, if any, of the constructor). Generic type parameter annotations and outer class annotations are not possible on the constructor result. Here is an example:

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

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

6. It is permitted to explicitly declare the method receiver as the first formal parameter. (Each non-static method has an implicit parameter, `this`, which is called the *receiver*.) Then, no special syntax is needed for expressing the receiver annotation: it is simply an annotation on the type of the `this` parameter.

   Only the first formal parameter may be named `this`, and such a parameter is only permitted on an instance method, or on a static method or constructor of an inner class. The type of the `this` parameter must be the same as the class that contains the method and should include type arguments if that class has any. In a method in an inner type, the receiver type can be written as (for example) either `Inner` or as `Outer.Inner`, and in the latter case annotations on both parts are possible, as in `@ReadOnly Outer . @Mutable Inner`.

   The optional `this` parameter has no effect on execution — it only serves as a place to write annotations on the receiver. The compiler generates the same bytecodes, and reflection returns the same results regarding number of method parameters, whether or not the optional `this` parameter is present.

   As an example, here are the standard definitions for `toString` and `equals`:

   ```
   class MyClass {
     ...
     public String toString() { ... }
     public boolean equals(Object other) { ... }
   }
   ```

   It is equivalent to write instead

   ```
   class MyClass {
     ...
     public String toString(MyClass this) { ... }
     public boolean equals(MyClass this, Object other) { ... }
   }
   ```

   and then it would be possible to annotate the receiver's type:

   ```
   class MyClass {
     ...
     public String toString(@Readonly MyClass this) { ... }
     public boolean equals(@Readonly MyClass this, @Readonly Object other) { ... }
   }
   ```

7. It is permitted to write an annotation on a type parameter declaration. This appears before the declared name, as in `class MyClass<@Immutable T> { ... }` or `interface WonderfulList<@Reified E> { ... }`. This is a declaration annotation, not a type annotation, but its omission in JSR 175 [Blo04] was an oversight, and JSR 308 is taking the opportunity to correct it.

2.1.1 Not all type names are annotatable

The Java language uses type names in three different ways: in type definitions/declarations; in type uses; and in other contexts that are not a type declaration or use. JSR 308 permits annotations on type uses (and also on type parameter declarations). JSR 308 does not support annotations on type names that syntactically look like, but are not, type uses. In the JLS grammar, type uses have a non-terminal name ending in *Type*.

   Here are examples of such type names that are not annotatable.

Annotation uses   An annotation use cannot itself be annotated. (An annotation *declaration* can be annotated by a so-called meta-annotation.) For instance, in

```
@MyAnnotation Object x;
```

there is no way to annotate the use of `MyAnnotation`.

Class literals   It is not permitted to annotate the type name in a class literal, as in

```
@Even int.class         // illegal!
int @NonEmpty [].class   // illegal!
```

This type name refers to a class, not a type. The expression evaluates to a `Class`, which does not reflect the type of the annotation, so there is no point in being able to write the annotation, which would have no effect.

Import statement   It is not permitted to annotate the type name in an `import` statement.

```
import @NotAllowed java.util.Date;       // illegal!
```

This use of a type name is not a type use and is more properly viewed as a scoping mechanism.

Static member accesses   Static member accesses are preceded by a type name, but that type name may not be annotated:

```
@Illegal Outer . StaticNestedClass      // illegal!
@Illegal Outer . staticField            // illegal!
```

A static member access may itself be a type use, in which case the used type may be annotated by annotating the last component of the static member access, which is the simple name of the type being used.

```
Outer . @Legal StaticNestedClass x = ...;     // legal
```

The type name in front of a static member access is a scoping mechanism, not a use of a type — there's nothing of type `Outer` in the above example. Furthermore, since there is only one instance of any static member, what it is cannot possibly be affected by an annotation on the name of the class being used to access it. Affecting the type of that single thing, depending on the annotation on the class name being used to access it, feels unnatural.

2.2   Java language grammar changes

This section gives changes to the grammar of the Java language [GJS+11, ch. 18], organized in the same way as the rules of Section 2.1. Additions are underlined.

Infelicities in the Java grammar make this section somewhat long. Some improvements are possible (for instance, by slightly refactoring the Java grammar), but this version attempts to minimize changes to existing grammar productions.

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

   Type:
      [Annotations] UnannType

   UnannType:
      (byte | short | char | int | long | float | double | boolean) { *[Annotations]* [] }
      UnannReferenceType { *[Annotations]* [] }

   MethodOrFieldDecl:
      UnannType Identifier MethodOrFieldRest

   InterfaceMethodOrFieldDecl:
      UnannType Identifier InterfaceMethodOrFieldRest

   Resource:
      { *VariableModifier* } *UnannReferenceType VariableDeclaratorId* = *Expression*

   ForVarControl:
      { *VariableModifier* } *UnannType VariableDeclaratorId ForVarControlRest*

   QualifiedTypeList:
      [Annotations] QualifiedType { , *[Annotations] QualifiedType* }

2. Annotations are permitted on the simple name in a static nested class.

ReferenceType:
    [Annotations] UnannReferenceType

UnannReferenceType:
    Identifier [TypeArguments] { . *[Annotations] Identifier [TypeArguments]*}

3. Annotations may appear on the wildcard in any type argument. A wildcard is the declaration of an anonymous (unnamed) type parameter.

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

4. 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].)

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

VariableDeclaratorId:
    Identifier {*[Annotations]* []}

MethodDeclaratorRest:
    FormalParameters {*[Annotations]* []} *[*throws *QualifiedTypeList] (Block* | ; *)*

InterfaceMethodDeclaratorRest:
    FormalParameters {*[Annotations]* []} *[*throws *QualifiedTypeList]* ;

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

Annotations may also appear on varargs (...):

FormalParameterDecls:
    {*VariableModifier*} *UnannType FormalParameterDeclsRest*

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

5. No grammar changes are required to support type annotations on constructor result types.

6. The receiver may be expressed explicitly, by changing the definition of "*FormalParameters*". Then, annotations appear on the receiver type in the normal way. If the receiver is not expressed explicitly, then its type cannot be annotated.

FormalParameters:
    (*[FormalParameterOrReceiverDecls]*)

FormalParameterOrReceiverDecls:
    {*VariableModifier*} *UnannType* this *[,* *FormalParameterDecls]*
    FormalParameterDecls

7. An annotation may appear on the type parameter declaration in a class or method declaration.

> *TypeParameter*:
>   *[Annotations] Identifier [*`extends` *Bound]*

2.2.1 Syntax of array annotations

As discussed in Section A.3, it is desirable to be able to independently annotate both the base type and each distinct level of a nested array. Forbidding annotations on arbitrary levels of an array would simplify the syntax, but it would reduce expressiveness to an unacceptable degree. The syntax of array annotations follows the same general prefix rule as other annotations — it looks slightly different because the syntax of array types is different than the syntax of other Java types. (Arrays are less commonly used than generics, so even if you don't like the array syntax, it need not bother you in most cases.)

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

In other words, the order in which a programmer reads an array type or expression is left-to-right for the brackets, *then* left-to-right for the base type.

```
type:                    String          []           []

order of reading:  2------------->  1 ---------------------->
```

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

```
type:                    String  @Length(2) []  @Length(5) []

order of reading:  2------------->  1 ---------------------->
```

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

```
type:           @English String  @NonNull []  @Length(10) []

order of reading:  2------------->  1 ---------------------->
```

An important property of this syntax is that, in two declarations that differ only in the number of array levels, the annotations mean the same thing. For example, `var1` has the same annotations as the elements of `arr2`:

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

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

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

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

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

2.3 Target meta-annotations for type annotations

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

- `ElementType.TYPE_PARAMETER` stands for a type parameter — that is, the declaration of a type variable. Examples are in generic class declarations `class MyClass<T> {...}`, generic method declarations `<T> foo(...) {...}`, and wildcards `List<?>`, which declare an anonymous type variable.

- `ElementType.TYPE_USE` stands for all uses of types, plus two two special cases.

  1. A type annotation (one meta-annotated with `@Target(ElementType.TYPE_USE)`) is permitted to be written on a a class declaration or a type parameter declaration. Strictly speaking, these are declaration sites, not uses of a type. However, it is convenient to write a type annotation at a type declaration, as shorthand for applying it at all uses. For example, `@Interned class MyClass { ... }` could indicate that all uses of `MyClass` are interned, even though for other classes some instances may be interned and other instances not interned.

  2. A type annotation may appear before a constructor, in which case it represents the object that the constructor is creating (which is not the same as the receiver of the constructor).

`ElementType.TYPE_PARAMETER` and `ElementType.TYPE_USE` are distinct from the existing `ElementType.TYPE` enum element of Java SE 7, which indicates that an annotation may appear on a type declaration (a class, interface, or enum declaration). The program elements denoted by `ElementType.TYPE_PARAMETER`, `ElementType.TYPE_USE`, and `ElementType.TYPE` are disjoint (except for the special cases about class and type parameter declarations, noted above).

As an example, in this declaration:

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

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

If an annotation is not meta-annotated with `@Target` (which would be poor style!), then the compiler treats the annotation as if it is meta-annotated with all of the `ElementType` enum constants that appear in Java 7: `ANNOTATION_TYPE`, `CONSTRUCTOR`, `FIELD`, `LOCAL_VARIABLE`, `METHOD`, `PACKAGE`, `PARAMETER`, and `TYPE`.

As in Java SE 7, the compiler issues an error if a programmer places an annotation in a location not permitted by its `@Target` meta-annotation. (The compiler issues the error even if no annotation processor is being run.)

A programmer could write a `@Target` meta-annotation indicating that an annotation is *both* a type annotation and a declaration annotation. We have not found an example where such a meta-annotation is desirable; although it is legal, it is considered bad style. For example, suppose that a programmer wrote

```
@Target({ ElementType.TYPE_USE, ElementType.METHOD })
public @interface SillyAnnotation { ... }
```

If `@SillyAnnotation` were written before a non-`void` method, then the annotation would apply to both the return type and the method declaration. The annotation appears twice in the AST during annotation processing, and it appears twice in the classfile. A tool that reads a classfile and writes Java-like output, such as Javadoc or a decompiler, must take care not to write an annotation twice in the decompiled code, as in "`@SillyAnnotation @SillyAnnotation int m()...`". It would be illegal to write `@SillyAnnotation` on a `void` method.

3 Class file format extensions

This section defines how to store type annotations in a Java class file. It also defines how to store local variable annotations, which are permitted in Java SE 7 source code but are discarded by the compiler.

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

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

First, annotated *signatures* (public members) must be available to tools that read class files. For example, a type-checking compiler plug-in [Dar06, PAC+08] needs to read annotations when compiling a client of the class file. The Checker Framework (`http://types.cs.washington.edu/checker-framework/`) is one way to create such plug-ins.

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

How Java SE 7 stores annotations in the class file In Java SE 7, an annotation is stored in the class file in an *attribute* [Blo04, LY07]. An attribute associates data with a program element (a method's bytecodes, for instance, are stored in a `Code` attribute of the method). The `RuntimeVisibleParameterAnnotations` attribute stores formal parameter annotations that are accessible at runtime using reflection, and the `RuntimeInvisibleParameterAnnotations` attribute stores formal parameter annotations that are not accessible at runtime. `RuntimeVisibleAnnotations` and `RuntimeInvisible-Annotations` are analogous, but for annotations on fields, methods, and classes.

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

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

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

Changes in JSR 308 JSR 308 introduces two new attributes: `RuntimeVisibleTypeAnnotations` and `RuntimeInvisible-TypeAnnotations`. These attributes are structurally identical to the `RuntimeVisibleAnnotations` and `RuntimeInvisible-Annotations` attributes described above with one exception: rather than an array of `annotation` elements, `RuntimeVisible-TypeAnnotations` and `RuntimeInvisibleTypeAnnotations` contain an array of `type_annotation` elements, which are described in Section 3.1.

```
Runtime[In]VisibleTypeAnnotations_attribute {
    u2 attribute_name_index;  // "Runtime[In]VisibleTypeAnnotation"
    u2 attribute_length;
    u2 num_annotations;
    type_annotation annotations[num_annotations];
}
```

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

A type annotation written on a declaration's type (e.g., on a field type or on a method return type) appears in the `Runtime[In]visibleTypeAnnotations` attribute.

No changes are made to the StackMapTable that stores the types of elements on the stack. No changes are made to the Exception table. Local variable type annotations are stored in the class file, see Section 3.3.8.

Backward compatibility For backward compatibility, JSR 308 uses new attributes for storing the type annotations. In other words, JSR 308 merely reserves the names of a few new attributes and specifies their layout. JVMs ignore unknown attributes. JSR 308 does not alter the way that existing annotations on classes, methods, method parameters, and fields are stored in the class file. JSR 308 mandates no changes to the processing of existing annotation locations; in the absence of other changes to the class file format, class files generated from programs that use no new annotations will be identical to those generated by a standard Java SE 7 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 referring to specific instructions, or bytecode offsets).

3.1 The `type_annotation` structure

The `type_annotation` structure has the following format:

```
type_annotation {
    // Original fields from "annotation" structure:
    u2 type_index;
    u2 num_element_value_pairs;
    {
        u2 element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs];
    // New fields in JSR 308:
    u2 target_type;     // the type of the targeted program element, see Section 3.2
    union {
        typeparam_target;
        supertype_target;
        typeparam_bound_target;
        empty_target;
        methodparam_target;
        exception_target;
        wildcard_bound_target;
        localvar_target;
        catch_target;
        offset_target;
        typearg_target;
    } target_info;  // uniquely identifies the targeted program element, see Section 3.3
}
```

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

- `type_index` is an index into the constant pool indicating the annotation type for this annotation.

- `num_element_value_pairs` is a count of the `element_value_pairs` that follow.

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

Compared to `annotation`, the `type_annotation` structure contains two additional fields. These fields implement a discriminated (tagged) union type: field `target_type` is the tag (see Section 3.2), and its value determines the size and contents of `target_info` (see Section 3.3).

3.2 The `target_type` field

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

3.3 The `target_info` field

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

All indexes count from zero.

See Section 3.3.12 for handling of generic type arguments and arrays.

Targets for type parameter declarations (`ElementType.TYPE_PARAMETER`):

Annotation target	`TargetType` enum constant	`target_type` value	`target_info` definition
class type parameter	`CLASS_TYPE_PARAMETER`	0x00, 0x01*	§3.3.1
method type parameter	`METHOD_TYPE_PARAMETER`	0x02, 0x03*	§3.3.1

Targets for type uses that may be externally visible in classes and members (`ElementType.TYPE_USE`):

Annotation target	`TargetType` enum constant	`target_type` value	`target_info` definition
class `extends`/`implements`	`CLASS_EXTENDS`	0x04, 0x05	§3.3.2
class type parameter bound	`CLASS_TYPE_PARAMETER_BOUND`	0x06, 0x07	§3.3.3
method type parameter bound	`METHOD_TYPE_PARAMETER_BOUND`	0x08, 0x09	§3.3.3
field	`FIELD`	0x0A, 0x0B	§3.3.4
method return type	`METHOD_RETURN`	0x0C, 0x0D	§3.3.4
method receiver	`METHOD_RECEIVER`	0x0E, 0x0F	§3.3.4
method parameter	`METHOD_PARAMETER`	0x10, 0x11	§3.3.5
exception type in `throws`	`THROWS`	0x12, 0x13	§3.3.6
wildcard bound	`WILDCARD_BOUND`	0x14, 0x15	§3.3.7

Targets for type uses that occur only within code blocks (`ElementType.TYPE_USE`):

Annotation target	`TargetType` enum constant	`target_type` value	`target_info` definition
local variable	`LOCAL_VARIABLE`	0x16, 0x17	§3.3.8
resource variable	**TODO**	0x18, 0x19	§3.3.8
exception parameter	**TODO**	0x1A, 0x1B*	§3.3.9
typecast	`TYPECAST`	0x1C, 0x1D	§3.3.10
type test (`instanceof`)	`INSTANCEOF`	0x1E, 0x1F	§3.3.10
object creation (`new`)	`NEW`	0x20, 0x21	§3.3.10
type argument in constructor call	`NEW_TYPE_ARGUMENT`	0x22, 0x23	§3.3.11
type argument in method call	`METHOD_TYPE_ARGUMENT`	0x24, 0x25	§3.3.11

Figure 1: Values of `target_type` for each possible target of JSR 308's new annotations. All of these appear in `type_annotation` attributes in the class file. Table elements such as "field", "method parameter", and "local variable" refer to the declaration, not the use, of such elements.

There are two `target_type` values for each annotation target: one for the construct itself, and one for type arguments, array types, or nested types that appear at the construct. The second `target_type` has the same structure as the first one, except it adds the fields described in §3.3.12 and its symbolic constant name adds the suffix `_COMPONENT`. The symbolic constants appear in the enums `com.sun.tools.javac.code.TargetType` and `com.sun.tools.classfile.ExtendedAnnotation.TargetType`.

Enumeration elements marked ∗ will never appear in a `target_type` field, because Java does not permit these constructs; see Section 3.2.

3.3.1 Type parameters

When the annotation's target is a type parameter of a class or method, `target_info` contains one `typeparam_target`:

```
typeparam_target {
    u1 param_index;
};
```

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

3.3.2 Class `extends` and `implements` clauses

When the annotation's target is a type in an `extends` or `implements` clause, `target_info` contains one `supertype_target`:

```
supertype_target {
    u2 type_index;
};
```

`type_index` specifies the index of the targeted type in the `interfaces` array field of the `ClassFile` structure; simply the value *i* is used if the annotation is on the *i*th superinterface type.

`-1` (`65535`) is used if the annotation is on the superclass type.

A `Runtime[In]visibleTypeAnnotations` attribute containing a `supertype_target` appears in the attributes table of a `ClassFile` structure.

3.3.3 Type parameter bounds

When the annotation's target is a bound of a type parameter of a class or method, `target_info` contains one `typeparam_bound_target`:

```
typeparam_bound_target {
    u1 param_index;
    u1 bound_index;
};
```

`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.

A `Runtime[In]visibleTypeAnnotations` attribute containing a `typeparam_bound_target` appears in the attributes table of a `method_info` structure if it targets a method type parameter bound, otherwise, in the attributes table of a `ClassFile` structure if it targets a class declaration type parameter bound.

3.3.4 Method return type, receiver, and fields

When the type annotation's target is a method return type, a constructor result, a receiver type (for an instance method or for an inner class constructor or static method), or a field, `target_info` is empty. More formally, it contains `empty_target`:

```
empty_target {
};
```

A `Runtime[In]visibleTypeAnnotations` attribute targeting a field type appears in the attributes table of a `field_info` structure. A `Runtime[In]visibleTypeAnnotations` attribute targeting a method return type, constructor result type, or receiver appears in the attributes table of a `method_info` structure.

3.3.5 Method parameters

When the annotation's target is a method parameter type, `target_info` contains one `methodparam_target`, which indicates which of the method's formal parameters is being annotated:

```
methodparam_target {
    u1 parameter_index;
};
```

A `Runtime[In]visibleTypeAnnotations` attribute containing a `methodparam_target` appears in the attributes table of a `method_info` structure.

3.3.6 `throws` clauses

When the annotation's target is a type in a `throws` clause, `target_info` contains one `exception_target`:

```
exception_target {
    u2 type_index;
};
```

`type_index` specifies the index of the exception type in the clause in `exception_index_table` of `Exceptions_attribute`; simply the value *i* denotes an annotation on the *i*th exception type.

A `Runtime[In]visibleTypeAnnotations` attribute containing a `exception_target` appears in the attributes table of a `method_info` structure.

3.3.7 Wildcard bounds

When the annotation's target is the bound of a wildcard, `target_info` contains one `wildcard_bound_target`:

```
wildcard_bound_target {
    u2 wildcard_location_type;   // The location of the wildcard, values as for target_type (Section 3.2)
    {
        ...
    } target_info;  // uniquely identifies the targeted program element, see Section 3.3
}
```

Here are examples (see Section 3.3.12 for an explanation of `location_length` and `location`):

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

```
target_type = WILDCARD_BOUND
target_info =
{
  wildcard_location_type = METHOD_PARAMETER_COMPONENT (0x0D)
  wildcard_location =
  {
    u1 parameter_index = 0;
    // the first type argument of List
    location_length = 1;
    location = { 0 };
  }
}
```

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

```
target_type = WILDCARD_BOUND_COMPONENT
target_info =
{
  // location of wildcard: type argument of the return type
  wildcard_location_type = RETURN_TYPE_COMPONENT;
  wildcard_location =
  {
    // the second type argument of Map
    location_length = 1;
    location = { 1 };
  }
  // the first type argument of List (the generic information of the annotation itself)
  location_length = 1;
  location = { 0 };
}
```

A `Runtime[In]visibleTypeAnnotations` attribute containing a `wildcard_bound_target` appears in the attributes table of:

1. a `ClassFile` structure, if the annotation appears on the class type parameter bound, a super class, or a super interface type,

2. a `field_info`, if the annotation appears on the field type arguments, or

3. a `method_info`, if the annotation appears on the method type parameter bound, return type, parameter types, exception types, or any type in its method body.

3.3.8 Local variables and resource variables

When the annotation's target is a local variable or resource variable, `target_info` contains one `localvar_target`:

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

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`, inclusive, to offset `start_pc + length`, exclusive). The `index` field stores the local variable's index in that method. These fields are similar to those of the optional `LocalVariableTable` attribute [LY07, §4.8.12].

Storing local variable annotations in the class file raises certain challenges. For example, live ranges are not isomorphic to local variables. Note that a local variable with no live range might not appear in the class file; that is OK, because it is irrelevant to the program.

A `Runtime[In]visibleTypeAnnotations` attribute containing a `localvar_target` appears in the attributes table of a `method_info` structure.

3.3.9 Exception parameters (`catch` clauses)

When the annotation's target is an exception parameter (the exception type in a `catch` statement), `target_info` indicates an offset in the exception table (which appears in the `exception_table` slot of the `Code` attribute).

```
catch_target {
    u2 exception_table_index;
};
```

A `Runtime[In]visibleTypeAnnotations` attribute containing a `catch_target` appears in the attributes table of a `method_info` structure.

3.3.10 Typecasts, type tests, and object creation

When the annotation's target is a typecast, an `instanceof` expression, or a `new` expression, `target_info` contains one `offset_target`:

```
offset_target {
    u2 offset;
};
```

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 the `new` bytecode emitted for the object creation expression. Typecast annotations are attached to a single bytecode, not a bytecode range (or ranges): the annotation provides information about the type of a single value, not about the behavior of a code block. A similar explanation applies to type tests and object creation.

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

If the compiler eliminates an annotated cast, it is required to retain the annotations on the cast in the class file. When a cast is removed, the compiler may need to adjust (the locations of) the annotations, to account for the relationship between the expression's type and the casted-to type. Consider:

```
class C<S, T> { ... }
class D<A, B> extends C<B, A> { ... }
...
... (C<@A1 X, @A2 Y>) myD ...
```

The compiler may leave out the upcast, but in that case it must record that `@A1` is attached to the second type argument of `D`, even though it was originally attached to the first type argument of `C`.

A `Runtime[In]visibleTypeAnnotations` attribute containing a `offset_target` appears in the attributes table of a `method_info` structure.

3.3.11 Constructor and method call type arguments

When the annotation's target is a type argument in a constructor call or a method call, `target_info` contains one `typearg_target`:

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

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

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

A `Runtime[In]visibleTypeAnnotations` attribute containing a `typearg_target` appears in the attributes table of a `method_info` structure.

3.3.12 Type arguments, array element types, and static nested types

In a parameterized, array, or nested type, there are multiple places that an annotation may appear. For example, consider the difference among `@X Map<String, Object>`, `Map<@X String, Object>`, and `Map<String, @X Object>`; or the difference among `@X String [] []`, `String @X [] []`, and `String [] @X []`; or the difference among `@X Outer . Inner` and `Outer . @X Inner`. The classfile must distinguish among these locations.

When the annotation's target is within a parameterized type, array type, or nested type, `target_info` contains what it normally would for a non-parameterized, non-array type *plus* the following fields at the end:

```
u2 location_length;
u1 location[location_length];
```
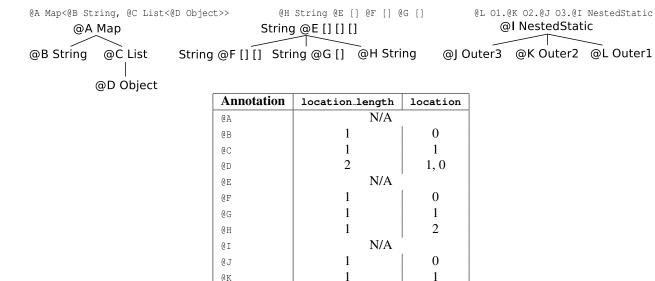
The `location` field is a list of numbers that represents the annotation location among the type arguments, array levels, and type nestings. If the type is viewed as a tree, then the `location` field represents a path in that tree. The `location_length` field is the depth of the annotation in the tree, which is also the length of the `location` field. Each element of the `location` field indicates which child to select at the given depth in the tree; the leftmost child is numbered 0.

A parameterized, array, or nested type can be viewed as a tree in the following way. For a parameterized type, the children of the parameterized type are the type arguments, numbered from the left. For a nested type, the children are the type arguments (if any), followed by the outer types, numbered from innermost to outermost. The type argument positions count even for a raw type; for example, the annotations on `@A Outer.@B MyClass` and `@A Outer<T>.@B MyClass<U>` are stored identically. For an array type, the children are the types of elements at each level, numbered from the first dimension. At each level, the element type is itself an array type (of lesser dimensions than the main type), except for the last one which is a non-array type. For a 1-dimensional array, there is only one child, which is a non-array.

Figure 2 shows some examples.

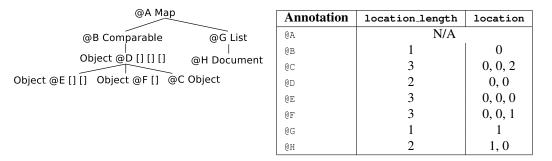The numbering in the class file is with respect to the full type, not the source code representation. In the source code, a type can be abbreviated or partially written out, such as a nested type written without the outer type or a raw type written without the type arguments.

In the class file, all annotations on the full type appear. For example, the classfile would record the type in this `new` expression as `@A Outer. @B Inner`:

```
@A Map<@B String, @C List<@D Object>>        @H String @E [] @F [] @G []        @L O1.@K O2.@J O3.@I NestedStatic
```
@A Map   String @E [] [] []   @I NestedStatic

@B String   @C List   String @F [] []   String @G []   @H String   @J Outer3   @K Outer2   @L Outer1

@D Object

Annotation	location_length	location
@A	N/A	
@B	1	0
@C	1	1
@D	2	1, 0
@E	N/A	
@F	1	0
@G	1	1
@H	1	2
@I	N/A	
@J	1	0
@K	1	1
@L	1	2

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

@A Map

@B Comparable   @G List

Object @D [] [] []   @H Document

Object @E [] []   Object @F []   @C Object

Annotation	location_length	location
@A	N/A	
@B	1	0
@C	3	0, 0, 2
@D	2	0, 0
@E	3	0, 0, 0
@F	3	0, 0, 1
@G	1	1
@H	2	1, 0

```
@A O1.@B O2<@C S,@D T>.@E O3.@F Nested<@G U, @H V>
```

@F Nested

@G U   @H V   @E O3   @B O2   @A O1

@C S   @D T

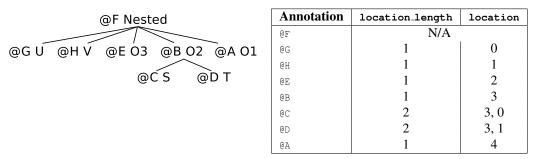| Annotation	location_length	location
@F	N/A	
@G	1	0
@H	1	1
@E	1	2
@B	1	3
@C	2	3, 0
@D	2	3, 1
@A	1	4

Figure 2: Example values of the location_length and location fields.
The top of the figure shows examples of each of parameterized types, array types, and nested types. The middle shows the interaction between parameterized types and array types. The bottom shows the interaction between parameterized types and nested types.
We assume that each annotation is a type annotation. A location array is used only for type annotations that are not on the main type. Type annotations on the main type are stored without a location array. For example, a type annotation on the outermost array level (@E in the top part of the figure) is not stored as an array level annotation but as a regular type annotation, see Figure 1.

```
@A Outer f1 = ...
... f1.new @B Inner() ...
```

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 7 annotations are inadequate. Similarly to type qualifiers, other pluggable type systems [Bra04] and similar lint-like checkers also require these extensions to Java's annotation system.

Our key goal is to create a type qualifier system that is compatible with the Java language, VM, and toolchain. Previous proposals for Java type qualifiers are incompatible with the existing Java language and tools, are too inexpressive, or both. The use of annotations for custom type qualifiers has a number of benefits over new Java keywords or special comments. First, Java already implements annotations, and Java SE 7 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://types.cs.washington.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 paper [PAC+08] discusses experience in which these plug-ins exposed bugs in real programs.

A.1   Examples of type qualifiers

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

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

`@Readonly` and `@Immutable` are other examples of useful type qualifiers [ZPA+07, BE04, TE05, GF05, KT01, SW01, PBKM00]. Similar to C's `const`, an object's internal state may not be modified through references that are declared `@Readonly`. A type qualifier designer would create a compiler plug-in (an annotation processor) to check the semantics

17

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

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. Typical code uses less than 1 type annotation per 50 lines [PAC+08], but this example was chosen to illustrate places where annotations do appear.
(1) Nullness: The @NonNull annotation is the default, so no reference in the DAG class may be null unless otherwise annotated. It is equivalent to writing line 3 as "@NonNull Set<@NonNull Edge> edges;", for example. This guarantees that the uses of edges on line 9, and of e on lines 10 and 11, cannot cause a null pointer exception. Similarly, the return type of getNeighbors() (line 7) is @NonNull, which enables its clients to depend on the fact that it will always return a List, even if v has no neighbors.
(2) Immutability: The two @Readonly annotations on method getNeighbors (line 7) guarantee to clients that the method does not modify, respectively, its Vertex argument or its DAG receiver (including its edges set or any edge in that set). The lack of a @Readonly annotation on the return value indicates that clients are free to modify the returned List.
(3) Interning: The @Interned annotation on line 7 (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 10 is valid. In the absence of such annotations, use of the equals method is preferred to ==.

---

of @Readonly. For instance, a method may only be called on a @Readonly object if the method was declared with a @Readonly receiver. (Each non-static method has an implicit parameter, this, which is called the *receiver*.) @Readonly's immutability guarantee can help developers avoid accidental modifications, which are often manifested as run-time errors. An immutability annotation can also improve performance. The Access Intents mechanism of WebSphere Application Server already incorporates such functionality: a programmer can indicate that a particular method (or all methods) on an Enterprise JavaBean is readonly.

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

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

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

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

A.3 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 a possibly-null array a with non-null elements.

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

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

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

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

For example, consider the following method:

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

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

The syntax also permits expressing constraints on the generic parameters of the receiver. Here are some examples:

```
class Collection<E> {
  // The elements of the result have the same annotation as the elements
  // of the receiver.  (In fact, they are the same elements.)
  public @PolyNull Object[] toArray(Collection<@PolyNull E> this) { ... }
}
interface List<T> {
  // The size() method changes neither the receiver nor any of the elements.
  public int size(@Readonly List<@Readonly T> this) { ... }
}
class MyMap<T,U> {
  // The map's values must be non-null, but the keys may be arbitrary.
  public void requiresNonNullValues(MyMap<T, @NonNull U> this) { ... }
}
```

In an inner class, there are two `this` parameters: that for the inner class, and that for the currently-executing method in the outer class. It is desirable to specify the types for both. One use case for this is a read-only type system, where you want to make the distinction between the outer and inner object. This can be specified as

```
void m(@ReadOnly Outer.@ReadWrite Inner this) {}
```

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

- There may be different receiver annotations on different methods that cannot be factored out into the containing class.

- Stating that a method does not modify its receiver is different than saying the method has no side effects at all, so it is not appropriate as a method annotation (such as JML's `pure` annotation [LBR06]).

- A receiver annotation is also distinct from a return value annotation: a method might modify its receiver but return an immutable object, or might not modify its receiver but return a mutable object.

Since a receiver annotation is distinct from other annotations, new syntax is required for the receiver annotation.

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

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

Casts   There are two distinct reasons to annotate the type in a type cast: to fully specify the casted type (including annotations that are retained without change), or to indicate an application-specific invariant that is beyond the reasoning capability of the Java type system. Because a user can apply a type cast to any expression, a user can annotate the type of any expression. (This is different than annotating the expression itself, which is not legal.)

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

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

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

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

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

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

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

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

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

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

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

Annotation of the type test permits the idiom

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

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

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

Inheritance   Annotations on class inheritance (`extends` and `implements`) are necessary to allow a programmer to fully specify a supertype. It would otherwise be impossible to extend the annotated version of a particular type t (which is often a valid subtype or supertype of t) without using an anonymous class.
These annotations also provide a convenient way to alias otherwise cumbersome types. For instance, a programmer might declare

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

so that `MyStringMap` may be used in place of the full, unpalatable supertype. However, this does somewhat limit reusability since a method declared to take a `MyStringMap` cannot take a `Map` of the appropriate type.

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 that can catch a `@Critical` exceptions are not empty.

There is no need for special syntax to permit annotations on the type of a caught exception, as in

```
catch (@NonCritical Exception e) { ... }
```

In this example case, a tool could warn if any `@Critical` exception can reach the `catch` clause.

B Interactions with other Java SE 8 language changes

This section discusses the interactions between type annotations and other language changes in Java SE 8.

Lambda There are two different syntaxes for a lambda argument list: normal and compact.

- The compact form does not explicitly specify a type, so no type annotation is possible. An example of this syntax is `# x -> x.toString` .

- The normal form permits type annotations; a programmer who wants to write type annotations must write the normal form.

No annotations are permitted on the lambda expression itself, because it is an expression, not a type.

Annotations cannot be written on the lambda's type, because the lambda's type cannot be written: lambda expressions are always immediately SAM-converted.

Modules The modules specification states that modules cannot be annotated.

Extension methods It might be desirable to annotate signatures for defaults. The syntax and semantics are still being worked out; when those are finalized, we will revisit the semantics of annotations on extension methods

Other Java SE 8 features Other new features do not raise any new interactions with type annotations.

It may be desirable to write an annotation that should should apply to all the disjuncts in a disjunctive union type, such as those in a multi-catch statement. Here is a possibility, though it is not currently part of the specification:

```
catch ( @A | @B NullPointerException | @C IllegalArgumentException e) {
```

An annotation before the first `|` is treated as applying to every disjunct, so the above would be equivalent to (but shorter than)

```
catch ( @A | @B NullPointerException | @C IllegalArgumentException e) {
```

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 3.

C.1 Compiler

The Java compiler must accept type annotations and add them to the program's AST. The relevant AST node classes must also be modified to store these annotations. When generating `.class` files, the compiler must emit the attributes described in Section 3.

This specification 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.)

The compiler sometimes creates new methods that did not appear in the source code. The compiler should annotate these compiler-generated methods in an appropriate way. For example, a bridge method is 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]. The bridge method does nothing but call an existing method, and in this case, annotations should be copied from the method being invoked.

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 type annotations as well. Also see Section C.4.

C.1.1 Backward and forward compatibility

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 8 -target 5 MyFile.java`. A Java 8 compiler produces a class file with the same attributes for type annotations as when the target is a version 8 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 7 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 7. Furthermore, the compiler must be provided with a definition of the annotation that is meta-annotated not with `@Target(ElementType.TYPE_USE)` (since `ElementType.TYPE_USE` does not exist in Java SE 7), but with no meta-annotation or with a meta-annotation that permits annotations on any declaration.

C.1.2 Optimizations

The compiler is required to preserve annotations in the class file. More precisely, if an expression has an annotated type (and the annotation has class file or runtime retention), 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. For expressions within a field initializer or a static initializer block, the compiled representation might be in an `<init>` or `<clinit>` method, or in a helper method called by them.

If the compiler optimizes away an expression, then it may also optimize away the annotation. (Exception: when a type cast is optimized away without optimizing away its argument, the annotation remains on the argument; see Section 3.2.)

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.

C.1.3 Annotations in comments

To ease the transition from standard Java SE 7 code to code with type annotations, the reference implementation recognizes the type annotations when surrounded by comment markers:
```
List</*@Readonly*/ Object> myList;
```

This permits use of both standard Java SE 7 tools and the new annotations even before Java SE 8 is released. However, it is not part of the proposal; that is, it is not required that every Java compiler parses comments. Oracle's OpenJDK implementation does not support a switch that makes it recognize the new annotations when embedded in comments. The Spec# [BLS04] extension to C# can be made compilable by a standard C# compiler in a similar way, by enclosing its annotations in special `/*^...^*/` comment markers. The `/*@` comment syntax is a standard part of the Splint [Eva96], ESC/Java [FLL+02], and JML [LBR06] tools (that is, not with the goal of backward compatibility).

C.2 ASTs and annotation processing

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

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

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

C.3 Reflection

To do: Complete this design.

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 type annotations in the top of Figure 1.

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

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

C.3.1 Non-changes to reflection

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

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

Method `Method.getParameterAnnotations()` returns the annotations on the parameter *declarations*, just as in Java SE 7. It does not return type annotations. There is no point in new methods that parallel it, such as `Method.getReceiverAnnotation` (for the receiver `this`) and `Method.getReturnAnnotation` (for the return value). Rather, the interface will provide a uniform mechanism for querying annotations on types.

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

C.4 Virtual machine and class file analysis tools

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

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

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

The compiler and other tools that read class files are trivially compatible with class files produced by a Java SE 7 compiler. However, the tools would not be able to read the impoverished version of type qualifiers that is expressible in Java SE 7 (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 SE 7, 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 Oracle 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://types.cs.washington.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://types.cs.washington.edu/annotation-file-utilities/annotation-file-format.pdf`), describes a textual format for annotations that is independent of `.java` or `.class` files. This textual format can represent annotations for libraries that cannot or should not be modified. We have built tools for manipulating annotations, including extracting annotations from and inserting annotations in `.java` and `.class` files. That file format is not part of this proposal for extending Java's annotations; it is better viewed as an implementation detail of our tools.

D   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 [PAC[+]08].

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

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

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

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

Mathias Ricken's LAPT-javac (`http://www.cs.rice.edu/~mgricken/research/laptjavac/`) is a version of javac (version 1.5.0_06) that encodes annotations on local variables in the class file, in new `Runtime{Inv,V}isibleLocalVariable-Annotations` attributes. The class file format of LAPT-javac differs from that proposed in this document. Ricken's xajavac (Extended Annotation Enabled javac) permits subtyping of annotations (`http://www.cs.rice.edu/~mgricken/research/xajavac/`).

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

Ownership types [CPN98, Boy04, Cla01, CD02, PNCB06, NVP98, DM05, LM04, LP06] permit programmers to control aliasing and access among objects. Ownership types can be expressed with type annotations and have

been applied to program verification [LM04, Mül02, MPHL06], thread synchronization [BLR02, JPLS05], memory management [ACG+06, BSBR03], and representation independence [BN02].

JavaCOP [ANMM06] is a framework for implementing pluggable type systems in Java. Whereas JSR 308 uses standard interfaces such as the Tree API and the JSR 269 annotation processing framework, JavaCOP defines its own incompatible variants. A JavaCOP type checker must be programmed in a combination of Java and JavaCOP's own declarative pattern-matching and rule-based language. JavaCOP's authors have defined parts of over a dozen type-checkers in their language. Their paper does not report that they have run any of these type-checkers on a real program; this is due to limitations that make JavaCOP impractical (so far) for real use.

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

Acknowledgments

Matt Papi, Mahmood Ali, and Werner Dietl designed and implemented the JSR 308 compiler as modifications to Oracle's OpenJDK javac compiler, and contributed to the JSR 308 design.

The members of the JSR 308 mailing list (`http://groups.google.com/group/jsr308-discuss`) provided valuable comments and suggestions. Additional feedback is welcome.

JSR 308 received the Most Innovative Java SE/EE JSR of the Year award in 2007, at the 5th annual JCP Program Awards. JSR 308's spec leads (Michael Ernst and Alex Buckley) were nominated as Most Outstanding Spec Lead for Java SE/EE in 2008, at the 6th annual JCP Program Awards. Michael Ernst won a Java Rock Star award for a presentation on the Checker Framework, which builds on the Type Annotations syntax, at JavaOne 2009.

References

[AAA06]   Marwan Abi-Antoun and Jonathan Aldrich. Bringing ownership domains to mainstream Java. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 702–703, Portland, OR, USA, October 24–26, 2006.

[ACG+06]   Chris Andrea, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time systems. In *ECOOP 2006 — Object-Oriented Programming, 20th European Conference*, pages 124–147, Nantes, France, July 5–7, 2006.

[AFKT03]   Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI 2003, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, CA, USA, June 9–11, 2003.

[ANMM06]   Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 57–74, Portland, OR, USA, October 24–26, 2006.

[BE04]   Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[BHP07]   Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering*, pages 215–229, Braga, Portugal, March 27–30, 2007.

[Blo04]   Joshua Bloch. JSR 175: A metadata facility for the Java programming language. `http://jcp.org/en/jsr/detail?id=175`, September 30, 2004.

[BLR02]   Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 211–230, Seattle, WA, USA, October 28–30, 2002.

[BLS04]   Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Marseille, France, March 10–13, 2004.

[BN02]   Anindya Banerjee and David A. Naumann. Representation independence, confinement, and access control. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–177, Portland, Oregon, January 16–18, 2002.

[Boy04]   Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 2004.

[Bra04]   Gilad Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages*, Vancouver, BC, Canada, October 25, 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 *PLDI 2003, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 324–337, San Diego, CA, USA, June 9–11, 2003.

[CCC05]   Walter Cazzola, Antonio Cisternino, and Diego Colombo. Freely annotating C#. *Journal of Object Technology*, 4(10):31–48, December 2005. Special Issue: OOPS Track at SAC 2005.

[CD02]   Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 292–310, Seattle, WA, USA, October 28–30, 2002.

[CJ07]   Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 227–247, Berlin, Germany, August 1–3, 2007.

[Cla01]   David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[CMM05]   Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 85–95, Chicago, IL, USA, June 13–15, 2005.

[CPN98]   David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 48–64, Vancouver, BC, Canada, October 20–22, 1998.

[Cro06]   Jose Cronembold. JSR 198: A standard extension API for Integrated Development Environments. `http://jcp.org/en/jsr/detail?id=198`, May 8, 2006.

[Dar06]   Joe Darcy. JSR 269: Pluggable annotation processing API. `http://jcp.org/en/jsr/detail?id=269`, May 17, 2006. Public review version.

[Det96]   David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[DF01]   Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 2001, Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, USA, June 20–22, 2001.

[DLNS98]   David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[DM05]   Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

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

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

[EM04]   Michael Eichberg and Mira Mezini. Alice: Modularization of middleware using aspect-oriented programming. In *4th International Workshop on Software Engineering and Middleware (SEM04)*, pages 47–63, Linz, Austria, December 2004.

[Eva96]   David Evans. Static detection of dynamic memory errors. In *PLDI 1996, Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, USA, May 21–24, 1996.

[FL03]   Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA, November 6–8, 2003.

[FLL+02]   Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[FTA02]   Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 17–19, 2002.

[GF05]   David Greenfieldboyce and Jeffrey S. Foster. Type qualifiers for Java. `http://www.cs.umd.edu/Grad/scholarlypapers/papers/greenfiledboyce.pdf`, August 8, 2005.

[GJS+11]   James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. `http://download.oracle.com/javase/7/specs/jls/JLS-JavaSE7.pdf`, Java SE 7 edition, 2011. "2011-07-15 Full" version.

[GJSB00]   James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, second edition, 2000.

[GJSB05]   James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.

[GPB+06]   Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[HK07]   Trevor Harmon and Raymond Klefstad. Toward a unified standard for worst-case execution time annotations in real-time Java. In *WPDRTS 2007, Fifteenth International Workshop on Parallel and Distributed Real-Time Systems*, Long Beach, CA, USA, March 2007.

[JPLS05]   Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Koblenz, Germany, September 7–9, 2005.

[JW04]   Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *13th USENIX Security Symposium*, pages 119–134, San Diego, CA, USA, August 11–13, 2004.

[KT01]   Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[LBR06]   Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3), March 2006.

[LM04]   K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, pages 491–, Oslo, Norway, June 16–18, 2004.

[LP06]   Yi Lu and John Potter. Protecting representation with effect encapsulation. In *Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–371, Charleston, SC, USA, January 11–13, 2006.

[LY99]   Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1999.

[LY07]   Tim Lindholm and Frank Yellin. The `class` file format. `http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf`, December 2007. Revision to chapter 4 of [LY99] for JDK 1.5.

[Mor06]   Rajiv Mordani. JSR 250: Common annotations for the Java platform. `http://jcp.org/en/jsr/detail?id=250`, May 11, 2006.

[MPHL06]   Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, October 2006.

[MPPD08]   Chris Male, David Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Compiler Construction: 14th International Conference, CC 2008*, pages 229–244, Budapest, Hungary, April 3–4, 2008.

[Mül02]   Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[NVP98]   James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 158–185, Brussels, Belgium, July 20-24, 1998.

[PAC+08]   Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[PBKM00]   Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Mississauga, Ontario, Canada, November 13–16, 2000.

[Pec03]   Igor Pechtchanski. *A Framework for Optimistic Program Optimization*. PhD thesis, New York University, September 2003.

[Pfe92]   Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.

[PNCB06]   Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 311–324, Portland, OR, USA, October 24–26, 2006.

[PØ95]   Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proceedings of the Second International Symposium on Static Analysis, SAS '95*, pages 314–329, Glasgow, UK, September 25–27, 1995.

[Pug06]   William Pugh. JSR 305: Annotations for software defect detection. `http://jcp.org/en/jsr/detail?id=305`, August 29, 2006. JSR Review Ballot version.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington, DC, USA, August 15–17, 2001.

[SW01] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP'2001: 3rd Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, June 18, 2001.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[VS97] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 607–621, Lille, France, April 14–18, 1997.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, December 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 20–22, 1999.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September–November 1998.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam 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*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.