# Possible extensions to Java annotations

Michael D. Ernst  
mernst@cs.washington.edu  
January 22, 2013

This document is a companion to the Type Annotations Specification (JSR 308). This document discusses some potential enhancements to annotations that would be worthwhile, but were not included in JSR 308. The JSR 308 webpage is [http://types.cs.washington.edu/jsr308/](http://types.cs.washington.edu/jsr308/).

## Contents

1. **Introduction**  
2. **Duplicate (multiple, repeated) annotations at a location**  
   2.1 Desugar into wrapper annotations  
   2.2 Add new `getAnnotations` method  
   2.3 JSR 175 design justification  
3. **Locations for annotations**  
   3.1 Annotations on statements  
   3.2 Expression annotations  
   3.3 Implicit Java types in casts  
   3.4 Disjunctions  
   3.5 Only certain statements  
4. **Changes to the annotation type**  
   4.1 Inheritance among annotations (subclassing/subtyping annotations)  
   4.2 Potentially recursive annotations (annotations as members of annotations)  
   4.3 Null as an annotation field value (nullable fields)  
   4.4 Positional arguments  
5. **Semantics of annotations**  
   5.1 Annotations for specific purposes  
   5.2 Annotation inheritance  
   5.3 Default annotations  
6. **Type abbreviations and typedefs**  
7. **Class file syntax**  
   7.1 Type annotations for stack elements  
   7.2 Reducing class file size via use of the constant pool  
   7.3 Optional/default fields in annotations  
8. **Access to method bodies in annotation processing API**
1 Introduction

The Type Annotations (JSR 308) specification extends Java’s annotation syntax. Java’s annotations may be enhanced again in the future, both to improve type annotations and for other purposes. It is a goal of the Type Annotations specification not to unnecessarily close off realistic future avenues of extension.

This document (which was previously an appendix to the Type Annotations specification) gives examples of further extensions to Java’s annotation system. Inclusion here is not an endorsement; this list is intended to be a resource for current and future language designers, explaining pros and cons of possible changes.

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

2 Duplicate (multiple, repeated) annotations at a location

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

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

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

A cleaner syntax is desirable for both purposes:

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

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

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

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

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

It is already possible in Java SE 6 for a location to have both an inherited and an explicit annotation of the same type. The `@Inherited` meta-annotation specification states that a `getAnnotations[]` query returns only the explicit annotation. This requirement should perhaps be relaxed, even if inheritance among annotations (Section 4.1) is not permitted. For some annotation processors, it is best to check that the explicit annotation is consistent with the overridden, inherited one and possibly merge information in the two annotations. The current design makes that inconvenient to do.
2.1 Desugar into wrapper annotations

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

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

into

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

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

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

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

This proposal would need to specify what happens if the user queries for annotation @A: would that always throw an exception (probably the best choice), or succeed if one @A annotation is present, or succeed if an @AContainer annotation with a length-1 array is present, or return the first @A annotation?

A problem with this proposal is that it requires defining an @AContainer annotation for each annotation @A, or else annotation @A cannot be duplicated. It would be better not to burden programmers with writing and maintaining this boilerplate code.

Another problem with this proposal is that it makes other possible extensions to annotations, such as inheritance among annotations (Section 4.1), more difficult and clumsy to implement and use. JSR 308 should aim to preserve, not to preclude, future avenues of extension.

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

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

2.2 Add new getAnnotations method

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

1. Create a new method

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

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

2. Slightly modify the specification of the existing method

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

When duplicate annotations exist, method getAnnotation could either give the first one (convenient for backward compatibility) or throw an exception (convenient to prevent erroneous processing). The latter seems like a better choice.
The arrays returned by the methods `getAnnotations(Class), getAnnotations(), and getDeclaredAnnotations()` are required to preserve the ordering of the annotations as they were declared in the original .java source file.

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

The problem with this proposal is that it requires adding a single method, `getAnnotations`, to two interfaces. While the changes would not break upward binary-compatibility, they would break source-compatibility: existing implementations of `Element` and `AnnotatedElement` could not be re-compiled under Java 8 without adding a `getAnnotations` method. Oracle’s policy generally frowns upon such changes that break source-compatibility, though Oracle sometimes makes them nonetheless. For example, the SQL interfaces have far more implementations than the compiler or reflection APIs, but Java SE 6 added three methods to the `java.sql.Statement` interface and made it extend `Wrapper`.

An alternative to changing the `Element` (and `AnnotatedElement`) interface is to create a new `Element2` interface that augments `Element` with the `getAnnotations` method, and to mark `Element` as deprecated. Then, programmers should rewrite their existing code to use the `Element2` interface instead of the `Element` interface. However, their code continues to compile until they make the change. A problem with this change is that it requires making a copy (with names suffixed “2”) of the entire `javax.lang.model` package, which is ugly.

Another alternative is to change the `Elements` utility interface. The new `getAnnotations` methods logically belong in `Element`, but they can be added to the `Elements` interface because it is explicitly documented with “Compatibility Note: Methods may be added to this interface in future releases of the platform.” Thus, adding static methods in `Elements` is a path toward extending functionality without changing core interfaces/classes such as `Element`. A problem with this proposal is that clients must remember to use the methods of `Elements` when functionality is not present in `Element`. A programmer browsing the `Element` interface might not think to check for static methods in `Elements`.

### 2.3 JSR 175 design justification

The JSR 175 Design FAQ [Blo04] briefly addresses why this feature was not originally included:

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

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

The argument that “the resulting system is simpler and cleaner” is perplexing. A client must check for both the singular and plural versions of the annotation and must handle the case when both the singular and the plural versions are present. By contrast to the stated claim, a system for duplicate annotations with an interface that always returns a list (or always returns an array) is what is needed to relieve the client of “worry about whether you’re going to get back a single instance”.

### 3 Locations for annotations

#### 3.1 Annotations on statements

Annotations on statements (or on some subset of statements, such as blocks or loops) would be useful for a variety of purposes, including atomicity/concurrency. Supporting annotations on statements would require defining both Java syntax and a convention for storing the information in the class file. See [http://code.google.com/p/jsr308-langtools/wiki/AnnotationsOnStatements](http://code.google.com/p/jsr308-langtools/wiki/AnnotationsOnStatements) for a proposal that summarizes why statement annotations are desirable, and that proposes a Java syntax, a classfile storage format, and how other tools will accommodate them. If you would like to help this feature become a reality, then please pitch in! You can join the [jsr308-discuss@googlegroups.com](mailto:jsr308-discuss@googlegroups.com) mailing list (via [http://groups.google.com/group/jsr308-discuss](http://groups.google.com/group/jsr308-discuss)), expand the partial design on the wiki, or work on the implementation.

The JSR 175 Design FAQ [Blo04] briefly addresses why this feature was not originally included:
Why can’t you annotate arbitrary program elements such as blocks and individual statements?
This would greatly complicate the annotation syntax: We would have to sacrifice the simplicity of saying
that annotations are simply modifiers, which can be used on declarations.

3.2 Expressions annotations

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

3.3 Implicit Java types in casts

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

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

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

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

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

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

An even shorter shorthand would drop the parentheses:

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

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

3.4 Disjunctions

It may be desirable to write an annotation that 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:

```java
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)
}
```

```java
catch { @A @B NullPointerException | @A @C IllegalArgumentException e} {
```
3.5 Only certain statements

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

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

4 Changes to the annotation type

4.1 Inheritance among annotations (subclassing/subtyping annotations)

In Java SE 6, annotations cannot subclass one another, and an annotation is not allowed to extend/implement any interfaces. These restrictions make it difficult to share behavior or to express similarities or relationships among annotation types. For example, it is inconvenient to define annotations with choices in their structure, such as a discriminated union that uses field names that act as explicit tags. It is impossible to create annotation processors or APIs that work with a specific set of annotations (say, all those with a given set of element names).

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

A potential objection is that multiple annotation processors might try to claim the same annotation, one directly and one by claiming its supertype. This can be accommodated. In general, annotation processor designers should, and can, avoid this situation by running one processor at a time for a family of annotations that are related by inheritance.

Without other changes, inheritance among annotations would not enable recursive or self-referential annotations (see Section 4.2).

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

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

The specification of the getAnnotation(Class) method must be updated. Suppose that annotation types Sub1 and Sub2 are direct subtypes of Super, and Super has a value field of type int. The specification must indicate the behavior of getAnnotations(Super.class) when called on locations with all possible combinations of the three annotations, with all possible combinations of arguments. Here are a few examples:

- a location that is annotated with @Sub1(1) but not annotated with @Super
- a location that is annotated with both @Super(0) and @Sub1(1)
- a location that is annotated with both @Sub1(0) and @Sub2(0), but not annotated with @Super
- a location that is annotated with @Sub1(1) and @Sub2(2), but not annotated with @Super

Reflection APIs may also need to be updated.

As is clear from the examples above, subclassing annotations raises the possibility of multiple effective annotations applying at a location. Therefore, subclassing annotations makes most sense if the specification already permits duplicate annotations (see Section 2). In that case, the new getAnnotations(Class) method returns all of the annotations. (Then, the client can decide what to do with them, such as raise an error, merge the annotations, or take some other action.)

The JSR 175 Design FAQ [Blo04] briefly addresses why this feature was not originally included:
Why don’t you support annotation subtyping (where one annotation type extends another)? It complicates the annotation type system, and makes it much more difficult to write “Specific Tools”.

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

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

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

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

### 4.2 Potentially recursive annotations (annotations as members of annotations)

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

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

However, an annotation type cannot have a member of its own type. “It is a compile-time error if an annotation type \( T \) contains an element of type \( T \), either directly or indirectly” [GJSB05, §9.6]. (If inheritance among annotations is permitted (see Section 4.1), then the natural interpretation of this is that no element’s declared type may be either the annotation itself or any supertype thereof.)

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

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

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

- It is impossible to define annotations that take an arbitrary annotation as an argument.
  
  Two examples of such annotations are the `@DefaultAnnotation` example above, and an annotation that expresses that a method is polymorphic over annotations (as opposed to polymorphic over types, as generics do).
  
  Another example is annotation parameters that do not correspond to a type parameter. Consider a container class that is designed to hold things of a specific type (say, it interacts with the elements in domain-specific ways), not of an arbitrary type. The container has no type parameter, but it may still make sense to have annotations on the contained thing. Most examples of generics and arrays also serve as examples here. The argument applies not just to container classes but also to algorithms that process data of a specific type.
  
  These examples require the annotation declaration to be potentially recursive. However, they do not require that any recursive annotation ever be created in practice. They could be achieved even if creation of recursive annotation instances was prohibited.

- It is impossible to define annotations with recursive structure.

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

To express the base case, recursive annotations also require either inheritance among annotations (Section 4.1),
or nullable members (Section 4.3), or a requirement that recursion takes place through an array (an empty array
would terminate the recursion).

A more modest approach that makes annotations somewhat more expressive would be to permit inheritance among
annotations (see Section 4.1) without permitting possibly self-referential annotations.

The JSR 175 Design FAQ [Blo04] briefly addresses why this feature was not originally included:

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

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

```java
@interface Foo {
    Foo foo() default @Foo;
}
@Foo int i;
```

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

4.3 Null as an annotation field value (nullable fields)

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

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

We note some possible objections to the proposal.

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

The programmer-defined special value provides better documentation than null, which might mean “none”, “unini-
tialized”, null itself, etc.

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

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

4.4 Positional arguments

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

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

5.1 Annotations for specific purposes

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

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

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

The JSR 305 team does not plan any reference implementation. This hinders both programmers who want to use the annotations, and also people trying to interpret the meaning of the specification. By contrast, the JSR 308 reference implementation and the Checker Framework for compile-time type-checking have been available since January 2007. The Checker Framework contains checkers for many of the most useful annotations proposed in JSR 305.

5.2 Annotation inheritance

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

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

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

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

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

5.3 Default annotations

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

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

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

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

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

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

In Java, it is not possible to define an annotation that takes an arbitrary annotation as a parameter; see Section 4.2.
An issue for JSR 260 (Javadoc) and JSR 305 (Annotation semantics) is how inherited and defaulted annotations are handled in Javadoc: whether they are written out in full, or in some abbreviated form. Just as too many annotations may clutter source code, similar clutter-reduction ideas may need to be applied to Javadoc.

6 Type abbreviations and typedefs

An annotated type may be long and hard to read; compare `Map<String, Object>` to `@NonNull Map<@NonNull String, @NonNull Object>`. Class inheritance annotations and subclassing provides a partial solution; consider the following example:

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

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

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

7 Class file syntax

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

7.1 Type annotations for stack elements

The Java class file stores the types of stack elements, at entry to each basic block in the control flow graph (which may differ from the Java blocks in the program). As a result, byte code verification need not perform type inference for local variables and stack elements. The class file format should eventually similarly store the annotations on those types, to aid in annotation verification of class files.

We will defer this change until after a maintenance review fixes some other issues in StackMap. The fixes will make this enhancement much easier to implement.

7.2 Reducing class file size via use of the constant pool

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

7.3 Optional/default fields in annotations

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

8 Access to method bodies in annotation processing API

A type-checking compiler plug-in (annotation processor) must process annotations (including those in method bodies), and it also must check each use of a variable/method whose declared type is annotated. For example, if a variable `x` is declared as `@NonNull Object x`, then every assignment to `x` must be checked, because any assignment `x = null;` would be illegal.
The JSR 269 annotation processing API does not process annotations on local variables, as it is not designed to access method bodies. This limitation makes JSR 269 insufficient for creating a type-checking compiler plug-in.

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

In the absence of such an API, a type checker can be written using compiler-specific APIs. This is how the Checker Framework ([http://types.cs.washington.edu/checker-framework/](http://types.cs.washington.edu/checker-framework/)) works.

References


