```
print(@Readonly Object x) {
  List<@NonNull String> lst;
  …
}
```

# JSR 308: Type Annotations

## Making Java annotations more general and more useful

Spec leads: Michael Ernst & Alex Buckley
Implementation lead: Werner Dietl

## Java 7: declaration annotations

```
@Deprecated
public class Date {
   ...
   @Override
   int compareTo(Date other) { ... }
   ...
}
```

## Type annotations

**JSR 308** permits annotating any use of a type

```
@Untainted String query;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
class UnmodifiableList<T>
  implements @Readonly List<@Readonly T> {}
```

## Syntax vs. semantics

- JSR 308 specifies syntax but not semantics
  - No change to Java compile-time, load-time, or run-time semantics
- Motivation: type qualifiers and pluggable type-checking
  - Prevent/detect errors, improve code quality
- Semantics: An annotation processor (compiler plug-in) can handle type annotations
  - Detect run-time errors at compile time
  - Compiler issues additional errors/warnings

## Outline

- Java syntax
- Classfile representation
- Relationship to other projects:
  - Java SE 8 language changes
  - Checker Framework (pluggable type-checking)
  - JSR 269: annotation processing
  - JSR 305: standard annotations for code quality
  - New datatype libraries: JSR 310 Time, JSR 354 Money
- Logistics
- Conclusion

# Prefix syntax: annotation appears before the type

```
class Folder<F extends @Existing File> { ... }
class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }
void monitorTemperature() throws @Critical TemperatureException { ... }

Map<@NonNull String, @NonEmpty List<@Readonly Document>> files;
Collection<? super @Existing File> files;
MyClass<@Immutable ? extends Comparable<MyClass>> unchangeable;
Map . @NonNull Entry  keyAndValue;

... (@NonNull String) myObject
... o.<@NonNull String>m("...")
... new @Interned MyObject()
... new @NonEmpty @Readonly List<String>(myNonEmptyStringSet)
... new <String> @Interned MyObject()

class MyClass<@Immutable T> { ... }
```

# Array and receiver syntax

- Array annotations appear before the relevant part of the type

```
@English String @NonEmpty []

@Readonly Document [][] docs1
Document @Readonly [][] docs2
Document[] @Readonly [] docs3
```

- Explicit receiver ("**this**" parameter); these are equivalent:

```
String toString() { ... }
String toString(MyClass this) { ... }
```

  - Presence or absence of the this parameter has no effect on semantics
  - No special type annotation syntax

## Distinguishing declaration annotations from type annotations

```
@Target(ElementType.METHOD)
public @interface Override { }

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


@Override
@NonNull String toString() { ... }
```

## Class file format

- Java stores declaraton annotations in classfile attributes
  - For formal parameters: RuntimeVisibleParameterAnnotations and RuntimeInvisibleParameterAnnotations
  - For fields, methods, and classes:  RuntimeVisibleAnnotations and RuntimeInvisibleAnnotations
- JSR 308 defines two new classfile attributes: RuntimeVisibleTypeAnnotations and RuntimeInvisibleTypeAnnotations
  - structurally identical to corresponding declaration annotations
- Also stores local variable declaration annotations

End of JSR 308 specification.
Now, discuss relationship to other projects.

## Java SE 8 language changes

- Lambda expressions
  - 2 forms of argument list:  normal and compact
  - compact form (#{ x -> x.toString }):  no type annotations
    - compiler copies annotations during inference (like diamond)
  - normal form:  permits type annotations
  - No annotations on lambda expression, which is not a type
- Extension methods:  deferred until that spec is stable
- Modules:  no annotations at present

- Annotation changes unrelated to JSR 308:
  - Repeated/duplicate/multiple annotations
  - @ParamName or @ParameterName annotation

## JSR 269: Annotation processing API

1. Iterator/visitor
   Designed for declaration annotations
   Visits each declaration (but not their bodies)
   Type annotations appear on signatures and in bodies
   JSR 308 makes no change
2. API for accessing information about signatures
   JSR 308 updates this
   Update reflection and serialization APIs similarly

## Pluggable type-checking

- Goal: prevent run-time errors
- Technique: detect the errors at compile time
  - Create a new type system
  - Annotate your program with the type system
- Checker Framework:
  - A useful third-party tool
  - Not part of JSR 308 or Java SE

# Java’s type checking is too weak

- Type checking prevents many bugs

```
int i = “hello”;    // type error
```

- Type checking doesn’t prevent enough bugs

```
System.console().readLine();
```

$\Rightarrow$ NullPointerException

```
Collections.emptyList().add(“One”);
```

$\Rightarrow$ UnsupportedOperationException

## Some errors are silent

```
Date date = new Date(0);
myMap.put(date, “Java epoch”);
date.setYear(70);
myMap.put(date, “Linux epoch”);
```

$\Rightarrow$ Corrupted map

```
dbStatement.executeQuery(userInput);
```

$\Rightarrow$ SQL injection attack

Initialization, data formatting, equality tests, …

## Solution: Pluggable type systems

- Design a type system to solve a specific problem
- Write type qualifiers in code (or, use type inference)

```
@Immutable Date date = new Date(0);
date.setTime(70);      // compile-time error
```

- Type checker warns about violations (bugs)

```
% javac -processor NullnessChecker MyFile.java

MyFile.java:149: dereference of possibly-null reference bb2
       allVars = bb2.vars;
                 ^
```

## Pluggable types in practice

- In use academically and industrially
  - not imposed on any user

- Case studies at U. of Washington

  > 3,000,000 lines of code annotated

  > 200 errors found

  also, improved the design

  < 1 annotation per 50 lines of code, often much less

  much less overhead than generic types

## @Regex annotation

- Goal: prevent PatternSyntaxException
- @Regex: a string that is a legal regular expression
  - Pattern.compile will not throw PatternSyntaxException
  - **`"a*(bc)?"`** is @Regex
  - **`"1) first point; 2) second point"`** is not @Regex
- An undergraduate annotated 4 programs
    - plume-lib, Daikon, Apache Lucene, Chukwa
    - > 500K LOC
  - Wrote 62 annotations
  - Detected over 20 bugs
    - Suppressed 16 false warnings

## Standard type annotations in Java SE 8?

- A small number of standard annotations may help users
  - The mandate of JSR 305 (now defunct)
- Annotations are stylized documentation
  - Terser than Javadoc (shorter code)
  - Machine-checked to detect bugs
  - Comparable to @Deprecated, @Override, etc.
- Semantics are defined independently of any annotation processor
  - No effect on compilation unless an annotation processor is specified
- Needs JDK annotations

## Candidate standard annotations

- Nullness: @NonNull, @Nullable
- Regexps: @Regex
- Type strings: @FieldDescriptor, @BinaryName, @FullyQualifiedName
- Interning: @Interned
- Property file keys
- Concurrency annotations (JCIP book)
- Fake enumerations

## New data structure APIs

- JSR 310 Time
- JSR 354 Money

Can define annotations to avoid run-time errors

## Logistics

- JCP process version 2.6
- Spec is complete and stable
  - some changes are scheduled for release in early 2012
  - most recent previous changes were in 2009
- Implementation complete except:
  - parsing of annotations on nested classes
  - renumbering an enum (trivial)
  - reflection & serialization (waiting for finalized spec)
- EDR planned in early 2012, when reference implementation is complete

## JSR 308: annotations on types

- Extends Java annotations
  - permits annotations to be written on any use of a type
  - defines syntax but not semantics
  - simple prefix syntax
- Enables pluggable type-checking
  - detects and prevents errors, improves code quality
- Specificaton and implementation status
  - publicly available: http://types.cs.washington.edu/jsr308/
  - operational and in daily use