JSR-305: Annotations for Software Defect Detection

William Pugh

Professor

Univ. of Maryland

pugh@cs.umd.edu

http://www.cs.umd.edu/~pugh/

Why annotations?

- Static analysis can do a lot
  - can even analyze interprocedural paths
- Why do we need annotations?
  - they express design decisions that may be implicit, or described in documentation, but not easily available to tools

Where is the bug?

```
if (spec != null) fFragments.add(spec);
if (isComplete(spec)) fPreferences.add(spec);
```

```
boolean isComplete(AnnotationPreference spec) {
    return spec.getColorPreferenceKey() != null
        && spec.getColorPreferenceValue() != null
        && spec.getTextPreferenceKey() != null
        && spec.getOverviewRulerPreferenceKey() != null;
    }
```

Finding the bug

- Many bugs can only be identified, or only localized, if you know something about what the code is supposed to do
- Annotations are well suited to this...
  - e.g., @Nonnull

JSR-305

- At least two tools already have defined their own annotations:
  - FindBugs and IntelliJ
- No one wants to have to apply two sets of annotations to their code
  - come up with a common set of annotations that can be understood by multiple tools

JSR-305 target

- JSR-305 is intended to be compatible with JSE 5.0+ Java

JSR-308

- Annotations on Java Types
- Designed to allow annotations to occur in many more places than they can occur now
  - ArrayList<@Nonnull String> a = ...
- Targets JSE 7.0
- Will add value to JSR-305, but JSR-305 cannot depend upon JSR-308

Nullness

- Nullness is a great motivating example
- Most method parameters are expected to always be nonnull
  - some research papers support this
- Not always documented in JavaDoc

Documenting nullness

- Want to document parameters, return values, fields that should always be nonnull
  - Should warn if null passed where nonnull value required
- And which should not be presumed nonnull
  - argument to equals(Object)
  - Should warn if argument to equals is immediately dereferenced

Only two cases?

- What about Map.get(...)?
- Return null if key not found
  - even if all values in Map are nonnull
- So the return value can’t be @Nonnull
- But lots of places where you “know” that the returned value will be nonnull

Null in some situation

- There are places where a value might be null, under some circumstance
  - return value from Map.get(Object)
  - first argument to Method.invoke(Object obj, Obj.. args)
    - should be null if method is a static method

Just dereference it

- If you use that value that might be null in some circumstance, but you have reason to believe should not be null (perhaps due to the values of other parameters, or program state)
  - you might just dereference it without null check
  - If it is null, something is wrong and throwing a NPE is the right thing to do

Data on use of Map.get(...)

- In JDK 1.6.0, build 105
- 196 calls to Map.get(...)
- where the return value is dereferenced without a null check

Thus, 3 cases

- I think there have 3 nullness annotations
  - @Nonnull
  - @NullFeasible
  - @UnknownNullness
    - same as no annotation
- Would *love* better name suggestions
  - might use @Nullable for one of last two these, but which one?

@Nonnull

- Should not be null
  - For fields, interpreted as should be nonnull after object is initialized
- Tools will try to generate a warning if they see a possibly null value being used where a nonnull value is required
  - same as if they see a dereference of a possibly null value

@NullFeasible

- Code should always worry that this value might be null
  - e.g., argument to equals
- Tools should flag any dereference that isn't preceded by a null check

@UnknownNullness

- *Same* as no annotation
  - Needed because we are going to introduce default and inherited annotations
  - Need to be able to get back to original state
- Null under some circumstances
  - might vary in subtypes, or depend on other parameters or state
- Interprocedural analysis might give us better information

@NullFeasible requires work

- If you mark a return value as @NullFeasible, you will likely have to go make a bunch of changes
  - kind of like const in C++
- My experience has been that there are lots of methods that could return null
  - but that in a particular calling context, you may know that it can’t

Proving no NPEs

- Some static analysis tools might want to prove that no null pointers can be dereferenced
- Such tools would likely warn if a value with unknown nullness is dereferenced without a null check
  - other tools would not generate a warning in this case

Type Qualifiers

- Many of the JSR-305 annotations will be type qualifiers: additional type constraints on top of the existing Java type system
  - aka Pluggable type system

@Nonnegative and friends

- Fairly clear motivation for @Nonnegative
- More?
  - @Positive
- Where do we stop?
  - @NonZero
  - @PowerOfTwo
  - @Prime

Three-way logic again

- If we have @Nonnegative, do we also need:
  - @Signed
    - similar to @NullFeasible
    - returned by hashCode(), Random.nextInt()
  - @UnknownSign
    - similar to unknown nullness

User defined type qualifiers

- In (too many) places, Java APIs use integer values or Strings where enumerations would have been better
  - except that they weren't around at the time
- Lots of potential errors, uncaught by compiler

Example in java.sql.Connection

createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)

Creates a Statement object that will generate ResultSet objects with the given type, concurrency, and holdability.

resultSetType: one of the following ResultSet constants:
ResultSet.TYPE_FORWARD_ONLY,
ResultSet.TYPE_SCROLL_INSENSITIVE, or
ResultSet.TYPE_SCROLL_SENSITIVE

resultSetConcurrency: one of the following ResultSet constants:
ResultSet.CONCUR_READ_ONLY or ResultSet.CONCUR_UPDATABLE

resultSetHoldability: one of the following ResultSet constants:
ResultSet.HOLD_CURSORS_OVER_COMMIT or
ResultSet.CLOSE_CURSORS_AT_COMMIT

The fix

- Declare
  - public @TypeQualifier
    @interface ResultSetType {}
  - public @TypeQualifier
    @interface ResultSetConcurrency {}
  - public @TypeQualifier
    @interface ResultSetHoldability {}
- Annotate static constants and method parameters

User defined Type Qualifiers

- JSR-305 won't define @ResultSetType
- Rather JSR-305 will define the meta-annotations
  - that allow any developer to define their own type qualifier annotations
  - which they can apply and will be interpreted by defect detection tools

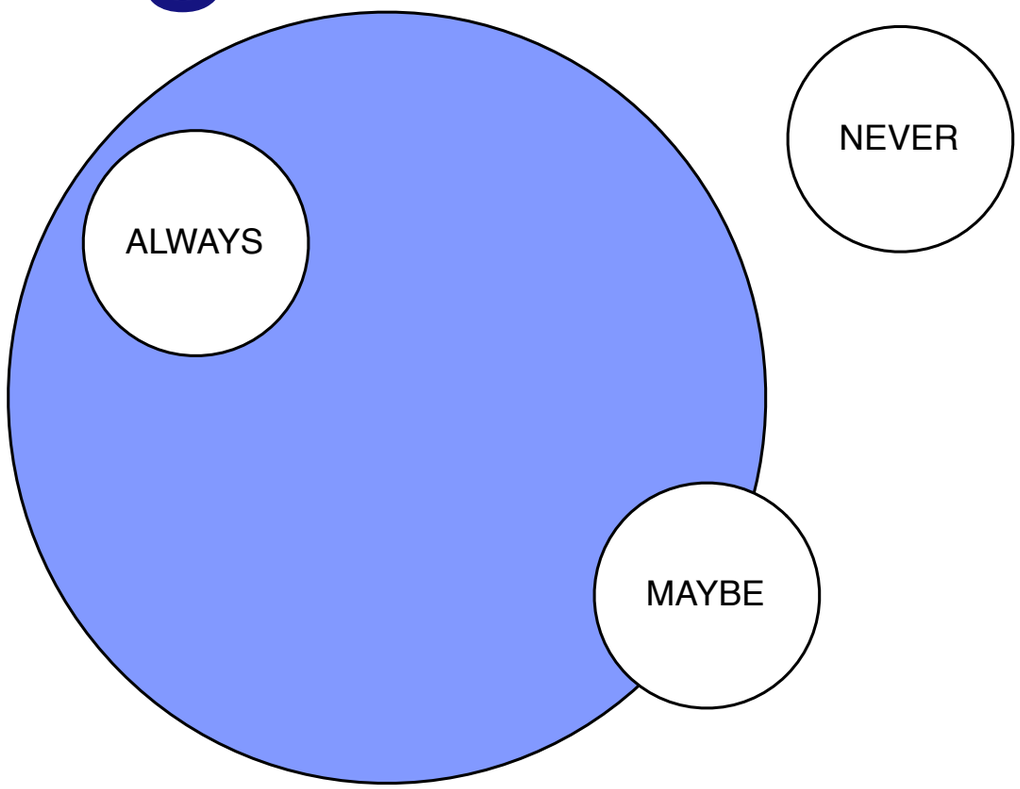
Defining a type qualifier

```
@TypeQualifier
public @interface Nonnull {
        When when() default When.ALWAYS;
        }
```

The When element

- An enum that describes the relationship between
  - the values S allowed at a location and
  - the set T of values described by the type qualifier
- values: ALWAYS, NEVER, MAYBE, UNKNOWN

Meanings



- ALWAYS: $S \subseteq T$
- NEVER: $S \subseteq \overline{T}$
- MAYBE: $\neg$ ALWAYS $\wedge \neg$ NEVER
- UNKNOWN: true

Applied to Nonnull

- Say we start by defining @Nonnull
- @Nonnull(when=When.MAYBE)
  - null feasible
- @Nonnull(when=When.UNKNOWN)
  - unknown nullness

Why so many when's?

- Don't want to bias type qualifiers as to whether you start with the positive or negative version
  - @Nonnull(when=When.NEVER)
  - represents a value that is always null
- But what if we had started by defining @Null
  - @Null(when=When.NEVER)
    - nonnull
  - @Null(when=When.MAYBE)
    - null feasible

More examples

- Start by defining @Negative
  - @Negative(when=When.NEVER)
    - nonnegative
  - @Negative(when=When.MAYBE)
    - signed

Checking type qualifiers

- If we detect a feasible path on which a
  - ALWAYS or MAYBE source
  - flows to a NEVER sink
  - generate a warning
  - And the converse
    - NEVER or MAYBE source flowing to an ALWAYS sink

Strict type qualifiers

- If you don't define a when element for a type qualifier, the type qualifier is *strict*
  - applying it is treated as ALWAYS
  - anything else is treated as UNKNOWN
  - report a warning if an UNKNOWN source reaches an ALWAYS sink
- Great for stuff like @ResultSetHoldability

The value element

- Type qualifiers can define a value element

```
@TypeQualifier public @interface Foo {
    int value();
}
```

  - Defines a family of type qualifiers: @Foo(1), @Foo(2), ...
  - Orthagonal/Independent
    - Whether a value is @Foo(1) is independent of whether it is @Foo(2)

Exclusive values

```
public @TypeQualifier(exclusive=true)
 @interface ClassName {
    Kind value();
    enum Kind { SLASHED, DOTTED };
}
```

- Defines two exclusive type qualifiers:
  `@ClassName(ClassName.Kind.SLASHED)`
  `@ClassName(ClassName.Kind.DOTTED)`
- Anything marked as
  `@ClassName(ClassName.Kind.SLASHED) ALWAYS`
  is also
  `@ClassName(ClassName.Kind.DOTTED) NEVER`

Exhaustive values?

```
public @TypeQualifier(exhaustive=true)
@interface Bar {
      Color value();
      enum Color { RED, GREEN, BLUE };
}
```

- If something is `Bar(Color.RED) NEVER` and `Bar(Color.GREEN) NEVER`
  it must be `Bar(Color.BLUE) ALWAYS`
- If a value is exhaustive, it is also exclusive.

Syntax

- Two potential ways to denote exclusive/ exhaustive type qualifiers
  1) Define boolean exclusive/exhaustive attributes to the TypeQualifier
  2) Define marker annotations that are applied to the value() member of a TypeQualifier

Possible syntax

- Syntax (1)

```
public @TypeQualifier(exclusive=true)
@interface Bar {
      Color value();
      enum Color { RED, GREEN, BLUE };
   }
```

- Syntax (2)

```
public @TypeQualifier @interface Bar {
      @Exclusive Color value();
      enum Color { RED, GREEN, BLUE };
   }
```

Example usage

- ```
  public @TypeQualifier(exhaustive=true)
  @interface Sign {
        NumericSign value();
        When when() default When.ALWAYS;
        enum NumericSign
              { NEGATIVE, ZERO, POSITIVE };
  }
  ```
- Can then define `@Nonnegative`, `@Nonzero`, etc.
  ```

Limiting the scope of a type qualifier

- When the @TypeQualifier annotation is applied, can define the types it can be applied to:

```
@TypeQualifier(
    applicableTo=CharSequence.class)
public @interface Foo {}
```

- primitive types allowed if wrapper types allowed

Type qualifier nicknames

- No one wants to be typing

```
@Nonnull(when=When.MAYBE)
```

  all over the place

- Define a type qualifier nickname

```
@TypeQualifierNickname
@Nonnull(when=When.MAYBE)
public @interface NullFeasible {}
```

Default and Inherited Type Qualifiers

Most parameters are nonnull

- Most reference parameters are intended to be non-null
  - many return values and fields as well
- Adding a @Nonnull annotation to a majority of parameters won't sell
- Treating all non-annotated parameters as nonnull also won't sell

Default type qualifiers

- Can mark a method, class or package as having nonnull parameters by default
  - If a parameter doesn't have a nullness annotation
    - climb outwards, looking at method, class, outer class, and then package, to find a default annotation
- Can mark a package as nonnull parameters by default, and change that on a class or parameter basis as needed

Inherited Annotations

- We want to inherit annotations
  - Object.equals(@CheckForNull Object obj)
  - int compareTo(@Nonnull E e)
  - @Nonnull Object clone()

Inherited qualifiers take precedence over default

- Default qualifiers shouldn’t interfere with or override inherited type qualifiers

Do defaults apply to most JSR-305 type qualifiers?

- Case for default and inherited nullness type qualifiers is very compelling
- Should it be general mechanism available for many/all type qualifiers?

Defining default type qualifiers

- @TypeQualifierDefault marks an annotation that can be used to specify default type qualifiers

```
@Nonnull
@TypeQualifierDefault(ElementType.PARAMETER)
public @interface ParametersAreNonnullByDefault {}
```

Meaning

- If TypeQualifierDefault is used to annotate an annotation
- Any type qualifiers also applied to the annotation are taken as defaults for the element types provided as arguments to the TypeQualifierDefault

Defined default annotations

- JSR-305 will only define `ParametersAreNonnullByDefault`, but more can be defined outside of JSR-305
  - and can be interpreted by static analyzers that interpret JSR-305 annotations

Using default annotations

- Just apply the annotation to a package, class, or method

```
@ParametersAreNonnullByDefault
class FooBar { ... }
```

Type qualifier validators

- A type qualifier can define a validator
  - typically, a static inner class to the annotation
- Checks to see if a particular value is an instance of the type qualifier
  - Static analysis tools can execute the validator at analysis time to check constant values
  - Dynamic instrumentation could check validator at runtime

CreditCard example

```
@Documented @TypeQualifier @Retention(RetentionPolicy.RUNTIME)
public @interface CreditCardNumber {
 class Validator implements TypeQualifierValidator<CreditCardNumber> {
  public boolean forConstantValue(CreditCardNumber annotation, Object v) {
    if (v instanceof String) {
     String s = (String) v;
      if (java.util.regex.Pattern.matches("[0-9]{16}", s)
          && LuhnVerification.checkNumber(s))
        return true;
      }
  return false;
}}}
```

Need a way to cast

- Need to provide a way to cast to a type qualifier
  - e.g., force the analysis to assume that the result returned by this method is always `@CreditCardNumber`, even if it can't prove it
- Use `when=When.ASSUME_ALWAYS` ?

Standard type qualifiers

@Syntax

- Used to indicate String values with particular syntaxes
  - @Syntax(“RegEx”)
  - @Syntax(“Java”)
  - @Syntax(“SQL”)
- Allows for error checking and used by IDE’s in refactoring

@MatchesPattern

- Provides a regular expression that describes the legal String values
  - @MatchesPattern(“\\d+”)
    - Indicates that the allowed values are non-empty strings of digits

@Untainted / @Tainted

- Needed for security analysis
- Information derived directly from web form parameters is tainted
  - can be arbitrary content
- Strings used to form SQL queries or HTML responses must be untainted
  - otherwise get SQL Injection or XSS

Question about @Taint

- Is one kind of taint sufficient, or do we want to allow specification of different kinds of taint?
  - Is HTTP request parameter taint distinct from command line taint or SQL result taint?

Annotations other than type qualifier

Thread/Concurrency Annotations

- Annotations to denote how locks are used to guard against data races
- Annotations about which threads should invoke which methods
- See annotations from *Java Concurrency In Practice* as a starting point

JCP Annotations

```
@ThreadSafe

@NotThreadSafe

@Immutable

@GuardedBy(“this”)

@GuardedBy(“lock”)

@GuardedBy(...)
```

What is wrong with this code?

```
Properties getProps(File file)
throws ... {
   Properties props = new Properties();
   props.load(new FileInputStream(file));
   return props;
   }
```

Doesn't close file

Resource Closure

- @WillNotClose
  - this method will not close the resource
- @WillClose
  - this method will close the resource
- @WillCloseWhenClosed
  - Usable only in constructors and factories: constructed object decorates the parameter, and will close it when the constructed object is closed

Miscellaneous

- @CheckReturnValue
- @OverridingMethodsMustInvokeSuper
- @InjectionAnnotation

@CheckReturnValue

- Indicates a method that should always be invoked as a function, not a procedure.
- Example:
  - String.toLowerCase()
  - BigInteger.add(BigInteger val)
- Anywhere you have an immutable object and methods that might be thought of a a mutating method return the new value

@OverridingMethodsMust InvokeSuper

- Applied to a method, it indicates that if this method is overridden in a subclass, the overriding method should invoke the annotated methods via an invocation using super

@InjectionAnnotation

- Static analyzers get confused if there is a field or method that is accessed via reflection/ injection, and they don't understand it
- Many frameworks have their own annotations for injection
- Using @InjectionAnnotation on an annotation @X tells static analysis tools that @X denotes an injection annotation

JSR-305 status

- David Hovemeyer and I have largely implemented in FindBugs what is described here for type qualifiers (mostly Dave)
  - still need to work on validators and type qualifiers with exclusive/exhaustive values