status report

JSR-305: Annotations for Software Defect Detection
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);
Where is the bug?

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

```java
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
- Hoped to have usable drafts and preliminary tool support out by the end of the summer
  - missed that, but close
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 value will be nonnull
  - you know key is in table
  - you know value is nonnull
3 cases?

- May need to have 3 cases for nullness
  - @NonNull
  - @NullFeasible
  - @UnknownNonNullness
- 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 benonnull after object is initialized

• Tools will try to generate a warning if they see a possibly null value being used where anonnull 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
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
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

```java
@Documented
@TypeQualifier
@Retention(RetentionPolicy.RUNTIME)
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 $\land \neg$ 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, 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`
Type qualifier nicknames

• No one wants to be typing
  ```java
  @Nonnull(when=When.MAYBE)
  public @interface NullFeasible {}
  ```
  all over the place

• Define a type qualifier nickname
  ```java
  @TypeQualifierNickname
  @Nonnull(when=When.MAYBE)
  public @interface NullFeasible {}
Annotations other than type qualifier

Many of these not yet supported in FindBugs
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;
}
What is wrong with this code?

```java
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: constructed object decorates the parameter, and will close it when the constructed object is closed
Miscellaneous

- `@CheckReturnValue`
- `@InjectionAnnotation`
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 as mutating methods return the new value.
@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

- Over the summer, David Hovemeyer and I have largely implemented what is described here for type qualifiers (mostly Dave)
  - not as far as long as we had hoped, but we are getting there