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

Detecting and preventing bugs with pluggable type-checking

CSE 331

Joint work with Mahmood Ali

http://types.cs.washington.edu/jsr308
http://types.cs.washington.edu/checker-framework

Motivation

java.lang.NullPointerException

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();
```

⇒ NullPointerException

```
Collections.emptyList().add("One");
```

⇒ UnsupportedOperationException

Some errors are silent

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

⇒ Corrupted map

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

⇒ SQL injection attack

Initialization, data formatting, equality tests, …

Problem: Your code has bugs

- Who discovers the problems?

I'm Feeling Lucky

  - If you are very lucky, testing discovers (some of) them
  - If you are unlucky, your customer discovers them
  - If you are very unlucky, hackers discover them
  - If you are smart, the compiler discovers them
- It's better to be smart than lucky

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;
                 ^
```

Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- Conclusion

Type qualifiers

- **Java 7**: annotations on types

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

- **Backward-compatible**: compile with any Java compiler

```
List</*@NonNull*/ String> strings;
```

Benefits of type qualifiers

- **Find bugs** in programs
- Guarantee the **absence of errors**
- **Improve documentation**
- Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
- Reduce number of assertions and run-time checks

- Possible negatives:
  - Must write the types (or use type inference)
  - False positives are possible (can be suppressed)

Outline

- Type qualifiers
- **Pluggable type checkers**
- Writing your own checker
- Conclusion

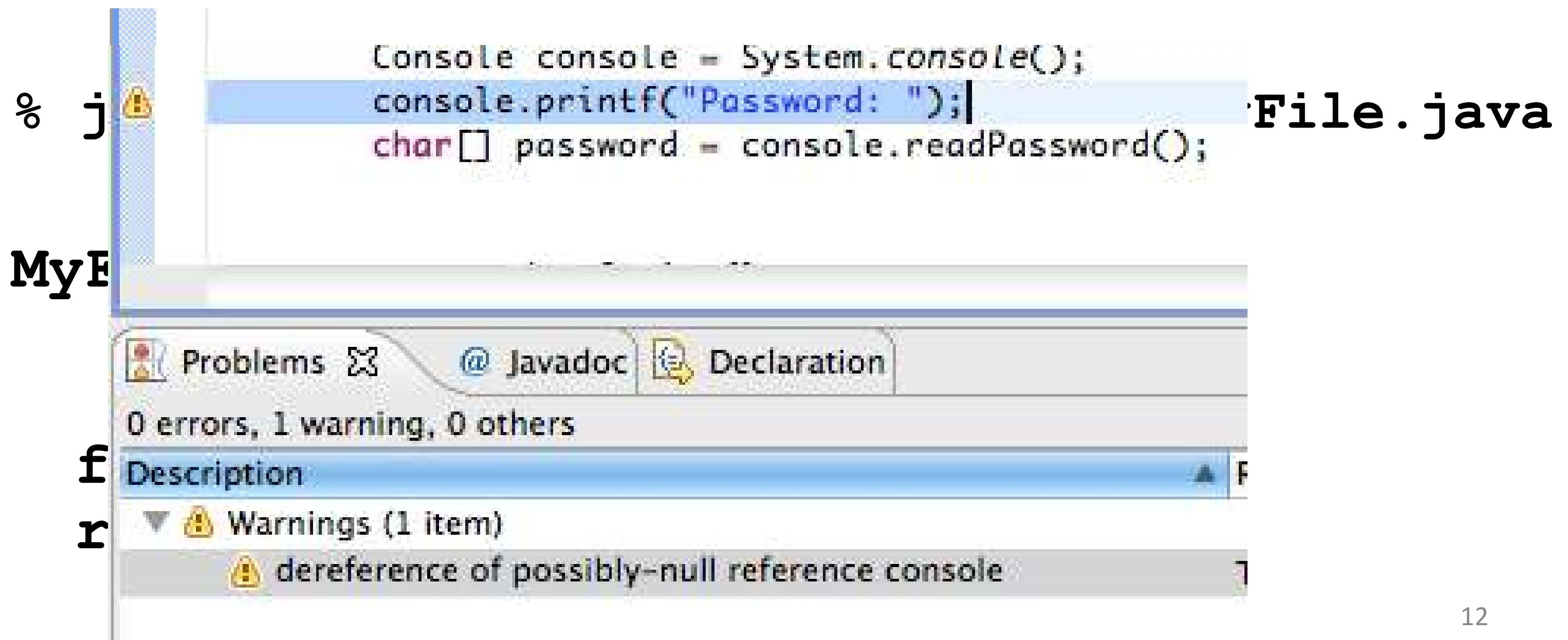
What bugs can you find & prevent?

	The annotation you write:
• Null dereferences	`@NonNull`
• Mutation and side-effects	`@Immutable`
• Concurrency: locking	`@GuardedBy`
• Security: encryption, tainting	`@Encrypted` `@Untainted`
• Aliasing	`@Linear`
• Equality tests	`@Interned`
• Strings: localization, regular expression syntax	`@Localized` `@Regex`
• Typestate (e.g., open/closed files)	`@State`

- You can write your own checker!

Using a checker

- Run in IDE or on command line
- Works as a compiler plug-in (annotation processor)
- Uses familiar error messages



Nullness and mutation demo

- Detect errors
- Guarantee the absence of errors
- Verify the correctness of optimizations

Checkers are effective

- Scales to > 200,000 LOC
- Each checker found errors in each code base it ran on
  - Verified by a human and fixed

Comparison: other Nullness tools

	Null pointer errors		False warnings	Annotations written
	Found	Missed		
Checker Framework	8	0	4	35
FindBugs	0	8	1	0
Jlint	0	8	8	0
PMD	0	8	0	0

- Checking the Lookup program for file system searching (4KLOC)
  - Distributed with Daikon (~200KLOC verified by our checker)
- False warnings are suppressed via an annotation or assertion
- Also, errors in Google Collections (>20,000 tests, FindBugs)

Checkers are featureful

- Full type systems: inheritance, overriding, etc.
- Generics (type polymorphism)
  - Also qualifier polymorphism
- Flow-sensitive type qualifier inference
  - Infers types for local variables
- Qualifier defaults
- Warning suppression

Checkers are usable

- Integrated with toolchain
  - javac, Eclipse, Ant, Maven
- Few false positives
- Annotations are not too verbose
  - `@NonNull`: 1 per 75 lines
    - with program-wide defaults, 1 per 2000 lines
  - `@Interned`: 124 annotations in 220KLOC revealed 11 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
- Inference tools: nullness, mutability
  - Adds annotations throughout your program

What a checker guarantees

- The program satisfies the type property. There are:
  - no bugs (of particular varieties)
  - no wrong annotations
- Caveat 1: only for code that is checked
  - Native methods
  - Reflection
  - Code compiled without the pluggable type checker
  - Suppressed warnings
    - Indicates what code a human should analyze
  - Checking part of a program is still useful
- Caveat 2: The checker itself might contain an error

Annotating libraries

- Each checker comes with JDK annotations
  - For signatures, not bodies
  - Finds errors in clients, but not in the library itself
- Inference tools for annotating new libraries

Outline

- Type qualifiers
- Pluggable type checkers
- **Writing your own checker**
- Conclusion

SQL injection attack

- Server code bug: SQL query constructed using unfiltered user input

```
query = "SELECT * FROM users "
      + "WHERE name='" + userInput + "';";
```

- User inputs: **a' or '1'='1**
- Result:

```
query ⇒ SELECT * FROM users
          WHERE name='a' or '1'='1';
```

- Query returns information about all users

Taint checker

```
@TypeQualifier
@SubtypeOf(Unqualified.class)
@ImplicitFor(trees = {STRING_LITERAL})
public @interface Untainted { }
```

To use it:

1. Write `@Untainted` in your program

   ```
   List getPosts(@Untainted String category) {…}
   ```

2. Compile your program

   ```
   javac -processor BasicChecker -Aquals=Untainted
     MyProgram.java
   ```

Taint checker demo

- Detect SQL injection vulnerability
- Guarantee absence of such vulnerabilities

Defining a type system

```
@TypeQualifier
public @interface NonNull { }
```

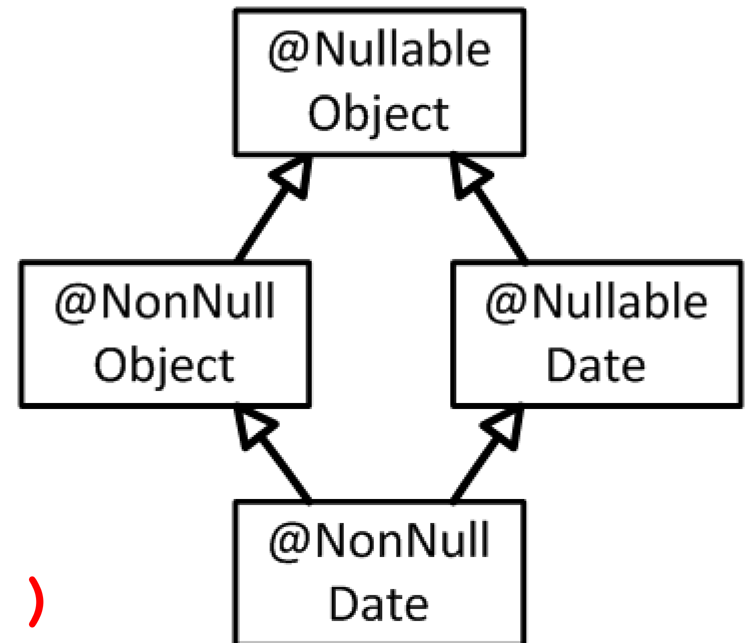
Defining a type system

1. Qualifier hierarchy – rules for assignment
2. Type introduction – types for expressions
3. Type rules – checker-specific errors

```
@TypeQualifier
public @interface NonNull { }
```

Defining a type system

What assignments are legal:



1. Qualifier hierarchy
2. Type introduction
3. Type rules

```
@TypeQualifier
@SubtypeOf( Nullable.class )
public @interface NonNull { }
```

Defining a type system

1. Qualifier hierarchy
2. Type introduction
3. Type rules

Gives the type of expressions:

```
new Date()
"hello " + getName()
Boolean.TRUE
```

```
@TypeQualifier
@SubtypeOf( Nullable.class )
@ImplicitFor(trees={ NEW_CLASS,
                     PLUS,
                     BOOLEAN_LITERAL, ... } )
public @interface NonNull { }
```

Defining a type system

1. Qualifier hierarchy
2. Type introduction
3. Type rules

Errors for unsafe code:

```
synchronized(expr) {
  …
}
```

Warn if expr may be null

```
void visitSynchronized(SynchronizedTree node) {
  ExpressionTree expr = node.getExpression();
  AnnotatedTypeMirror type = getAnnotatedType(expr);
  if (! type.hasAnnotation(NONNULL))
    checker.report(Result.failure(...), expr);
}
```

Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- **Conclusion**

Pluggable type-checking

- Java 7 syntax for type annotations
  - Write in comments during transition to Java 7
- Checker Framework for creating type checkers
  - Featureful, effective, easy to use, scalable
- Prevent bugs at compile time
- Create custom type-checkers
- Learn more, or download the Checker Framework:
  http://types.cs.washington.edu/jsr308
  (or, web search for “Checker Framework” or “JSR 308”)