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

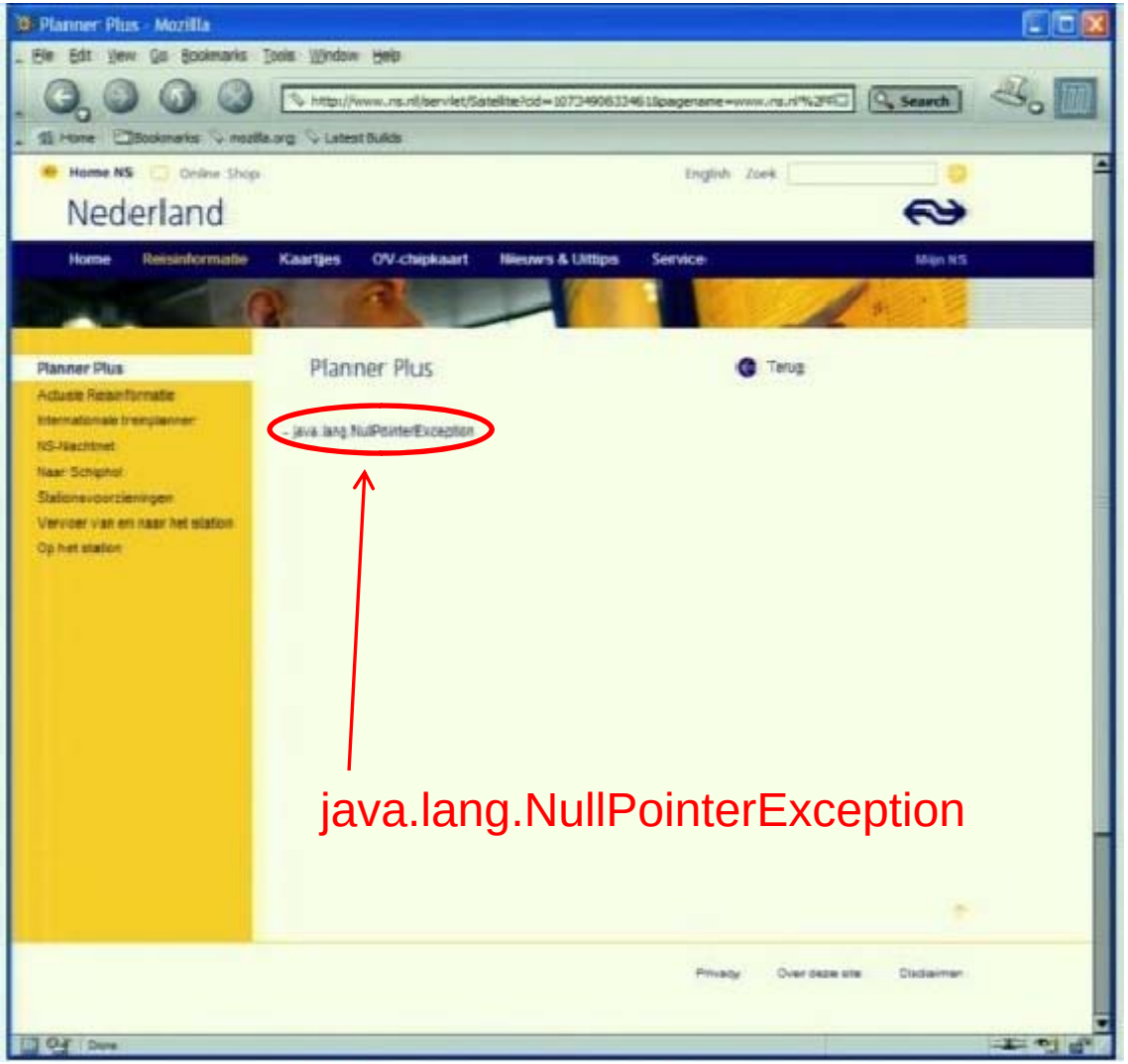
Preventing bugs with pluggable type checking

Michael Ernst

University of Washington

Joint work with Mahmood Ali and Matthew Papi

Motivation


Planner Plus - Mozilla
Nederland
Home
Reisinformatie
Kaartjes
OV-chipkaart
Nieuws & Uitlips
Service
Planner Plus
Terug
java.lang.NullPointerException
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);
```

⇒ UnsupportedOperationException

Equality tests, initialization, data formatting, …

Solution: Pluggable type systems

- Design a type system to solve a specific problem
- Write type qualifiers in your code (or, 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** annotation syntax

```
@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

- **Improve documentation**
- **Find bugs** in programs
- Guarantee the **absence of errors**
- Aid compilers and analysis tools
- Reduce the need for assertions and run-time checks

Outline

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

Sample checkers

- **`@NonNull`**: null dereference
- **`@Interned`**: incorrect equality tests
- **`@Immutable`**: incorrect mutation and side-effects
- Many other simple checkers
  - Security: encryption, tainting, access control
  - Encoding: SQL, URL, ASCII/Unicode
- Under construction:
  - CMU, ETH Zurich, MIT, Radboud U., U. of Buenos Aires, U. of California at Los Angeles, U. of Saarland, U. of Washington, U. of Wisconsin, Washington State U., ……

Nullness and mutation demo

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 a 4KLOC program
- False warnings are suppressed via an annotation or assertion

Checkers are featureful

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

Checkers are usable

- Integrated with toolchain
  - javac, Ant, Eclipse, Netbeans
- Few false positives

- Annotations are not too verbose
  - **`@NonNull`**: 1 per 75 lines
  - **`@Interned`**: 124 annotations in 220KLOC revealed 11 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
  - Inference tools: nullness, mutability

What a checker guarantees

- The program satisfies the type property
  - There are no bugs (of particular varieties)
- Caveat: 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

Annotating libraries

- Each checker includes JDK annotations
  - Typically, only 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 't'='t**
- Result:

```
query ⇒ SELECT * FROM users
          WHERE name='a' or 't'='t';
```

- Query returns information about all users

Tainting 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
```

Tainting checker demo

Defining a type system

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

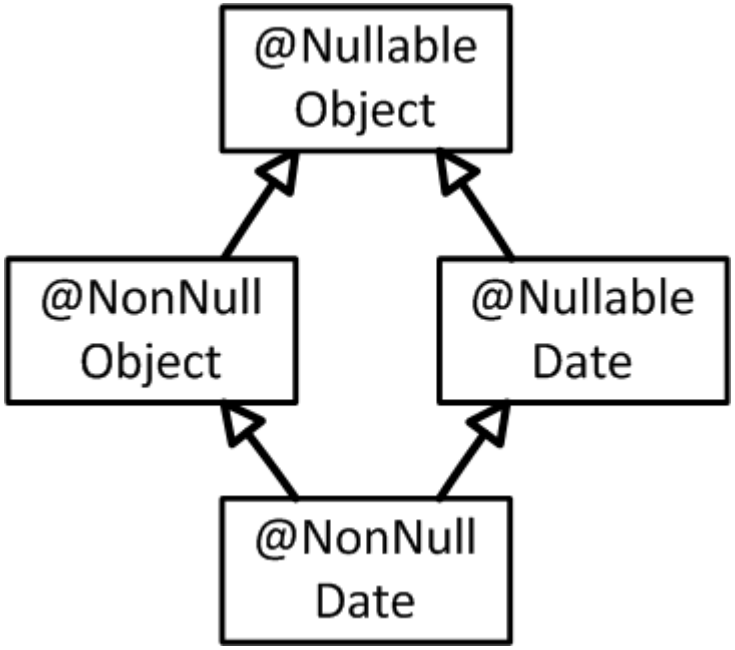
Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

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

Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules




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

Defining a type system

1. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

```
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. Type qualifier hierarchy
2. Type introduction rules
3. Other type rules

```
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**

Research results

- First practical system for pluggable types
  - This lack held back research and practice
- Significant case studies led to:
  - new type systems
  - new insights about old ones
- Linear-time inference algorithm
- See paper "Practical pluggable types for Java" (in ISSTA 2008)

My other research

Making it easier (and more fun!) to create reliable software

Security:

- Finding and exploiting web vulnerabilities
- Automatically patching vulnerabilities
- Quantitative information-flow

Programming languages:

- Type systems: immutability, canonicalization
- Type inference: abstractions, polymorphism, immutability

Testing:

- Creating complex test inputs
- Generating unit tests from system tests
- Classifying test behavior

More: Reproducing in-field failures; combined static & dynamic analysis; analysis of version history; refactoring; …

Contributions

- Checker Framework for creating type checkers
  - Featureful, effective, easy to use, scalable
- Prevent bugs at compile time
- Create custom type-checkers
- Download: http://pag.csail.mit.edu/jsr308