

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

Preventing bugs with pluggable type checking

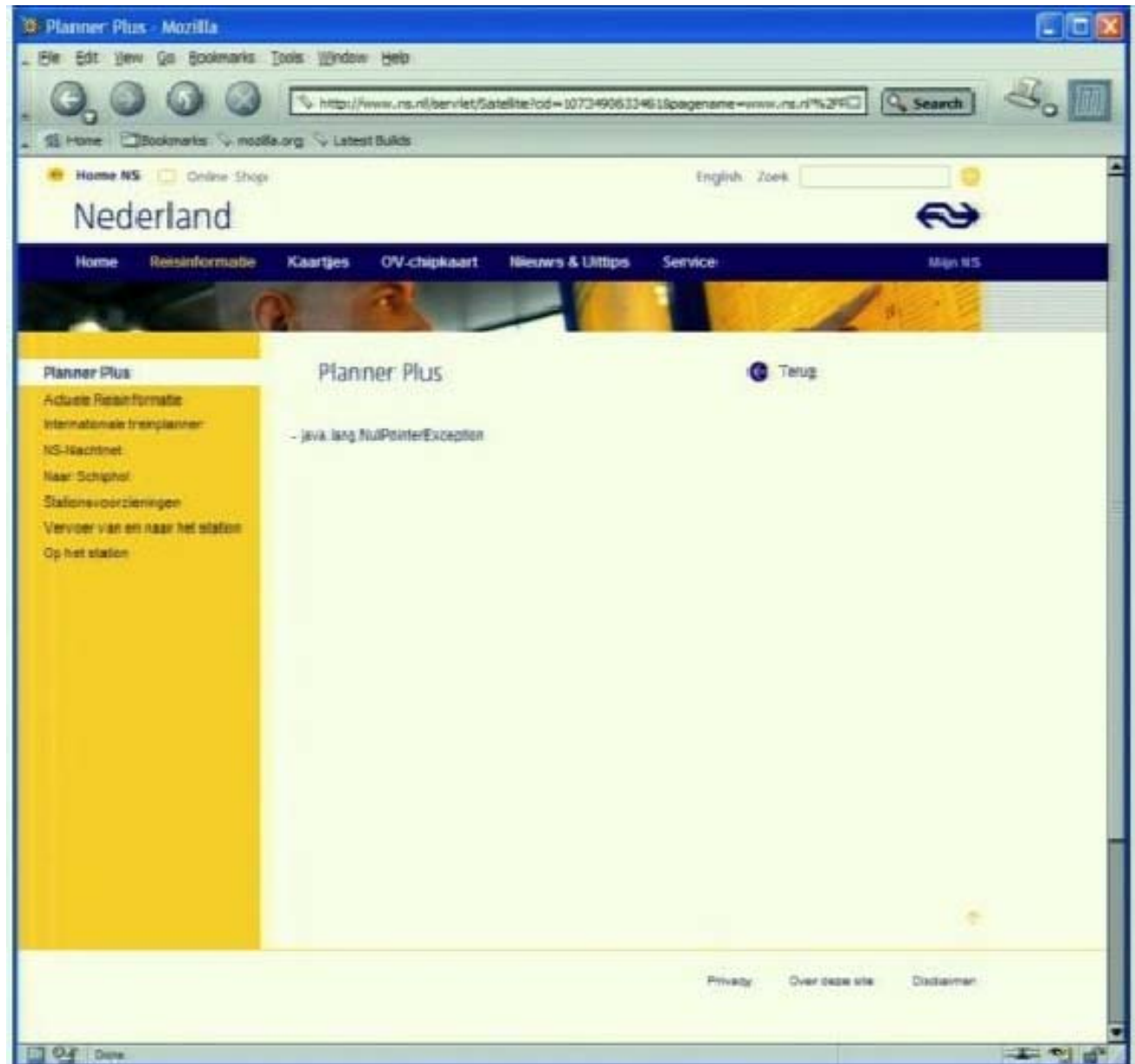
Michael D. Ernst

University of Washington

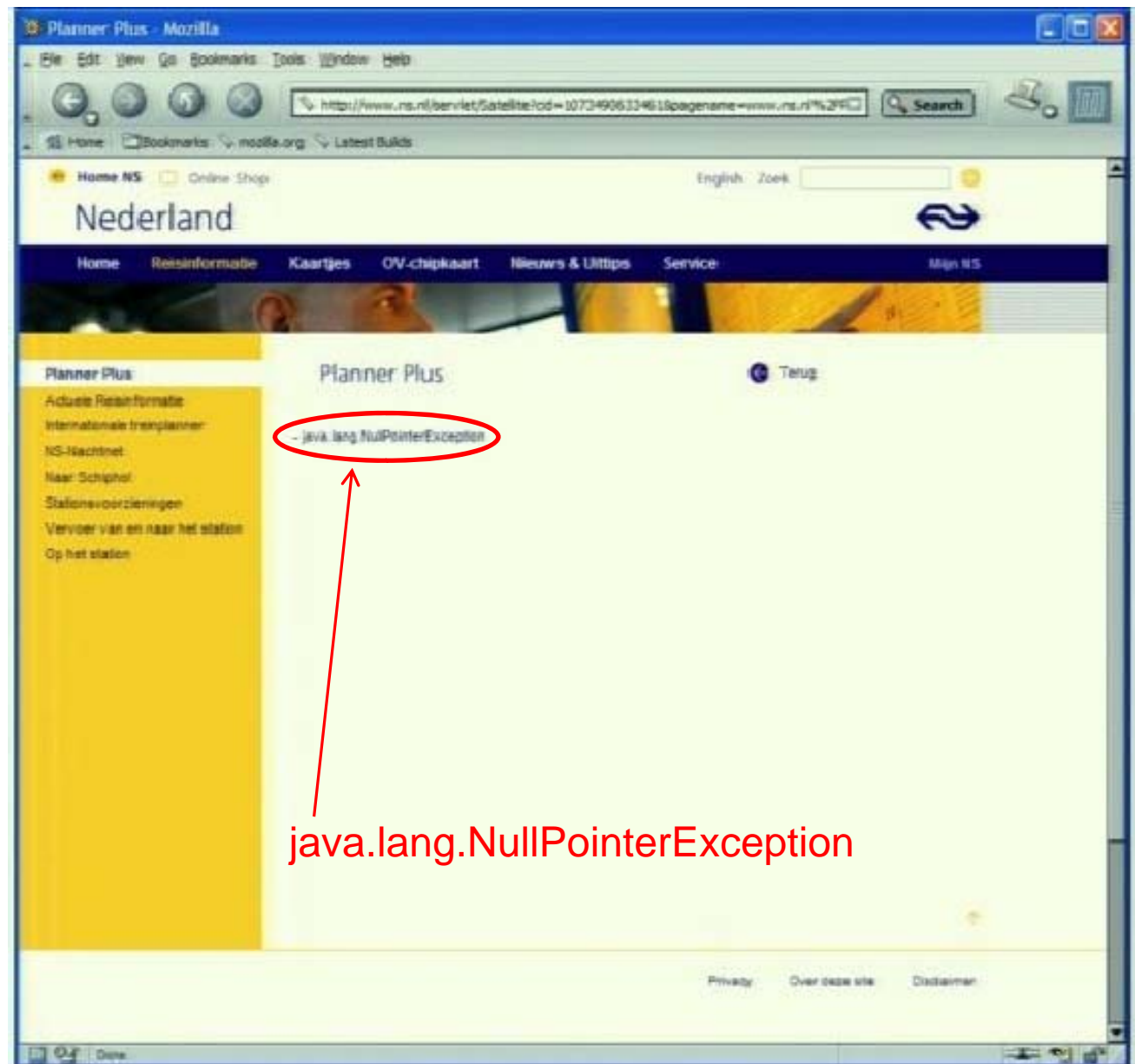
Joint work with Mahmood Ali

<http://pag.csail.mit.edu/jsr308>

Motivation



Motivation



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, ...

Solution: Pluggable type systems

- > Design a type system to solve a specific problem
- > Write type qualifiers in 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, 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

Sample checkers

- > **@NonNull**: null dereference
- > **@Immutable**: incorrect mutation and side-effects
- > **@Interned**: incorrect equality tests
- > Many other simple checkers
 - Security: encryption, tainting, access control
 - Encoding: SQL, URL, ASCII/Unicode
- > Even more are under construction
 - You can write your own!

Using checkers

- > Designed as compiler plug-ins (i.e., annotation processors)
- > Use familiar error messages

```
% javac -processor NullnessChecker MyFile.java
```

```
MyFile.java:9: incompatible types.
```

```
    nonNullVar = nullableValue;
```

```
                ^
```

```
found    : @Nullable String
```

```
required: @NonNull String
```

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 the Lookup program for file system searching (4KLOC)
 - Distributed with Daikon (>100KLOC verified by our checker)
- 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, Maven, Eclipse, Netbeans
- > 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

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
 - 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

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

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 { }
```

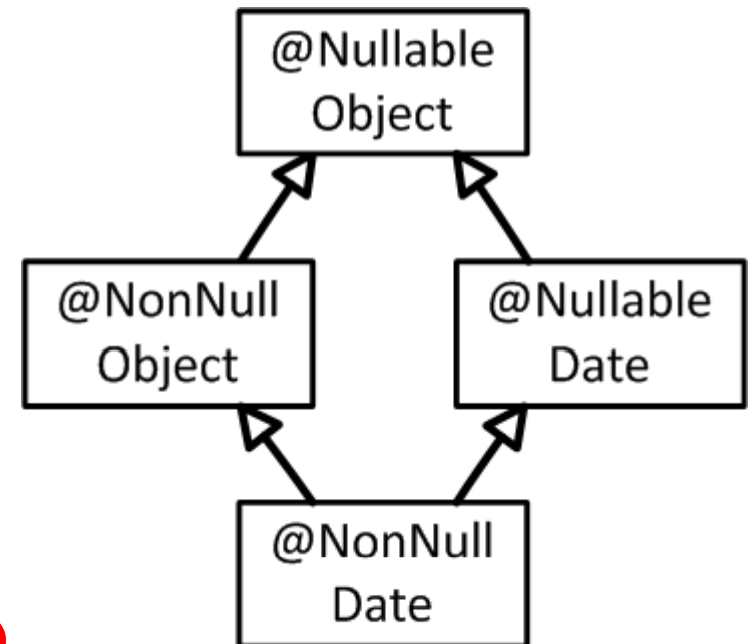
Defining a type system

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**

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://pag.csail.mit.edu/jsr308>
(or, web search for “Checker Framework” or “JSR 308”)