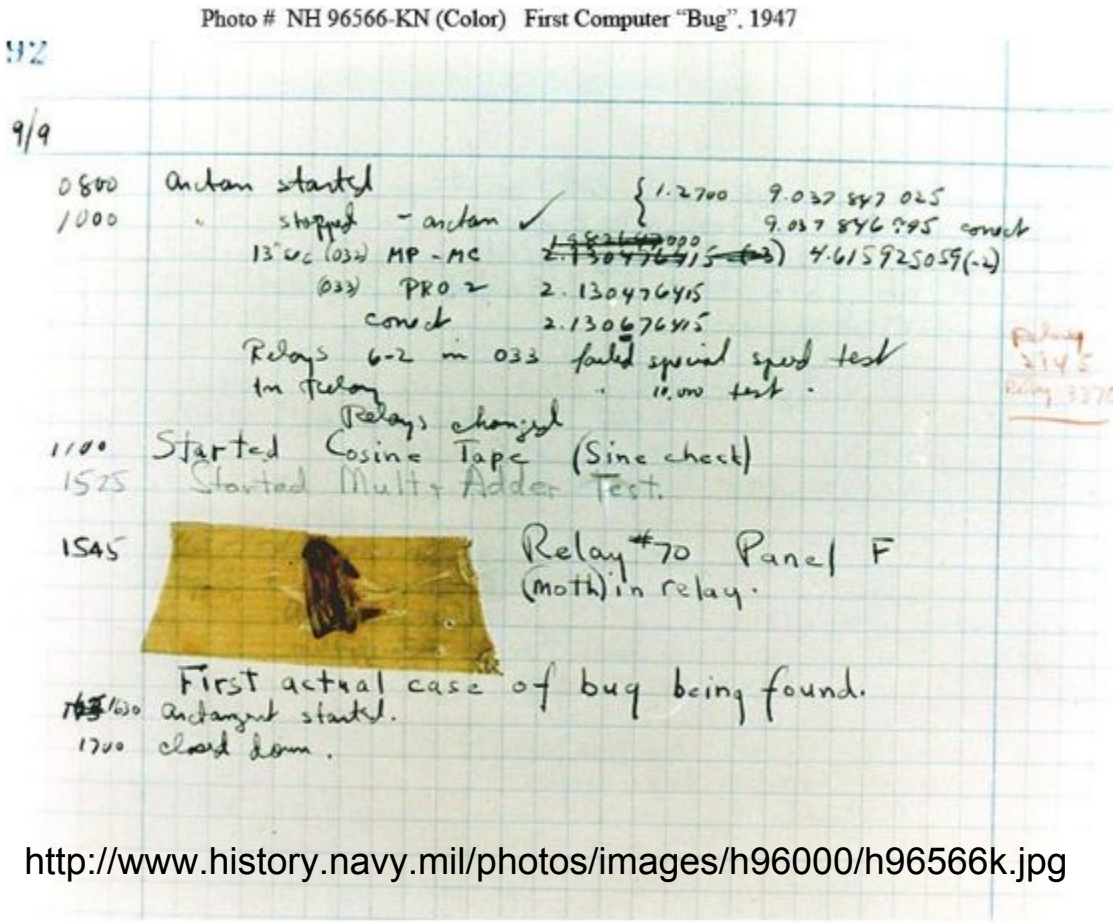
Werner Dietl | University of Waterloo

Hands-on with the *Checker Framework:* Preventing Null Pointer Exceptions at Compile Time

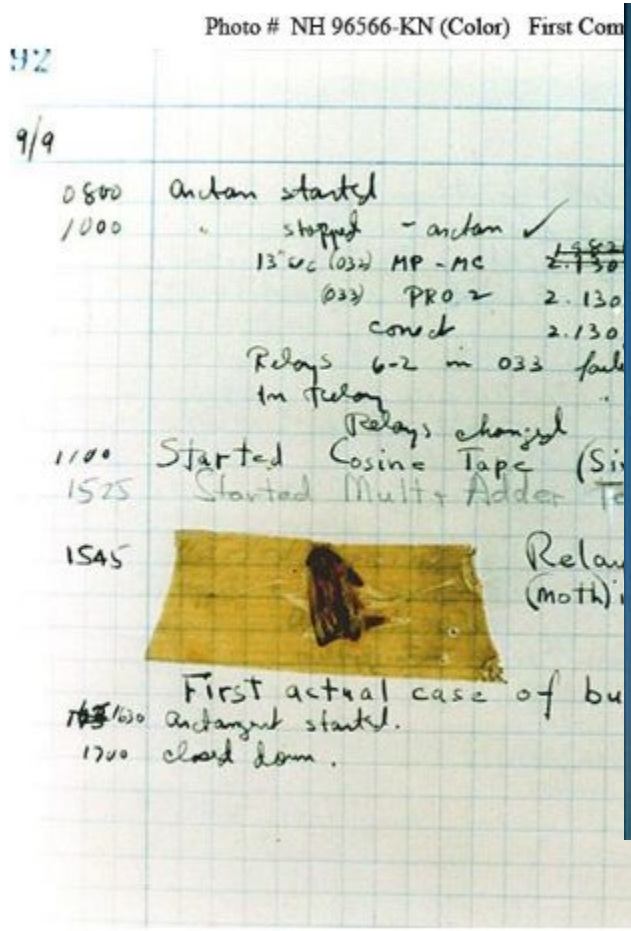
Bug Evolution



http://www.history.navy.mil/photos/images/h96000/h96566k.jpg

http://checkerframework.org/

Bug Evolution



http://dlimages.businessweek.com/imageserve/0arRbTIdRSg3e/630x418.jpg

Cost of software failures

$312 billion per year global cost of software bugs (2013)

$300 billion dealing with the Y2K problem

$440 million loss by Knight Capital Group Inc. in 30 minutes in August 2012

$650 million loss by NASA Mars missions in 1999; unit conversion bug

$500 million Ariane 5 maiden flight in 1996; 64 bit to 16 bit conversion bug

Software bugs can cost lives

225 deaths: jet crash caused by radar software (1997)

28 deaths: Patriot missile guidance system (1991)

11 deaths: blackout (2003)

>8 deaths: Radiation therapy (1985-2000)

2011: Software cause for 25% of all medical device recalls

Outline

- Solution: Pluggable type-checking
- Tool: Checker Framework
- Nullness Checker
- Creating your own type system
- Project components

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”;
```

Type checking doesn't prevent enough errors

```
System.console().readLine();

Collections.emptyList().add(“one”);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”;
```

Type checking doesn't prevent enough errors

```
System.console().readLine();

Collections.e
```

NullPointerException

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”;
```

Type checking doesn't prevent enough errors

```
System.console().readLine();

Collections.emptyList().add(“one”);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”;
```

Type checking doesn't prevent enough errors

```
System.console().readLine();

Collections.emptyList().add(“one”);
```

UnsupportedOperationException

Some errors are silent

```
Date date = new Date();
myMap.put(date, "now");
date.setSeconds(0);  // round to minute
myMap.get(date);
```

Some errors are silent

```
Date date = new Date();
myMap.put(date, "now");
date.setSeconds(0);  // round to minute
myMap.get(date);
```

Element not found

Some errors are silent

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

Some errors are silent

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

SQL injection attack

Initialization, data formatting, equality tests, …

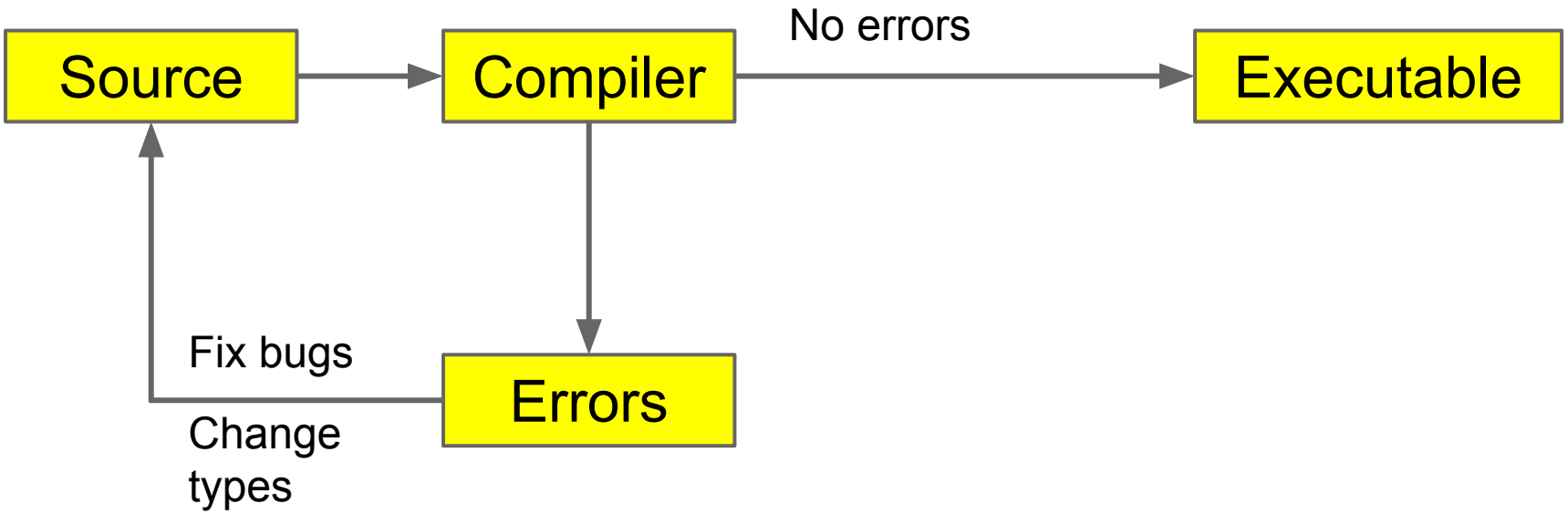
Solution: Pluggable Type Checking

1. Design a type system to solve a specific problem

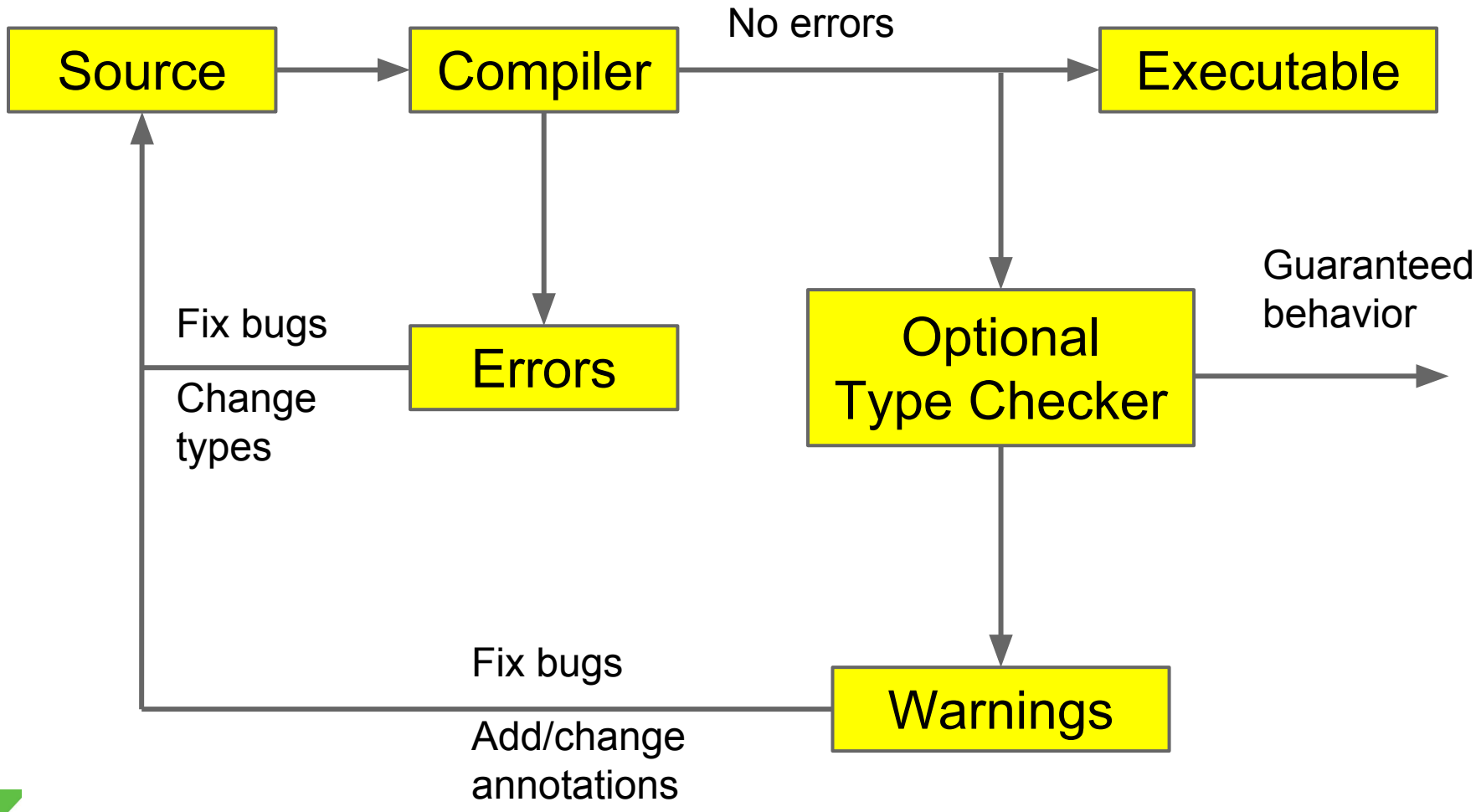
2. Write type qualifiers in code (or, use type inference)

3. Type checker warns about violations (bugs)

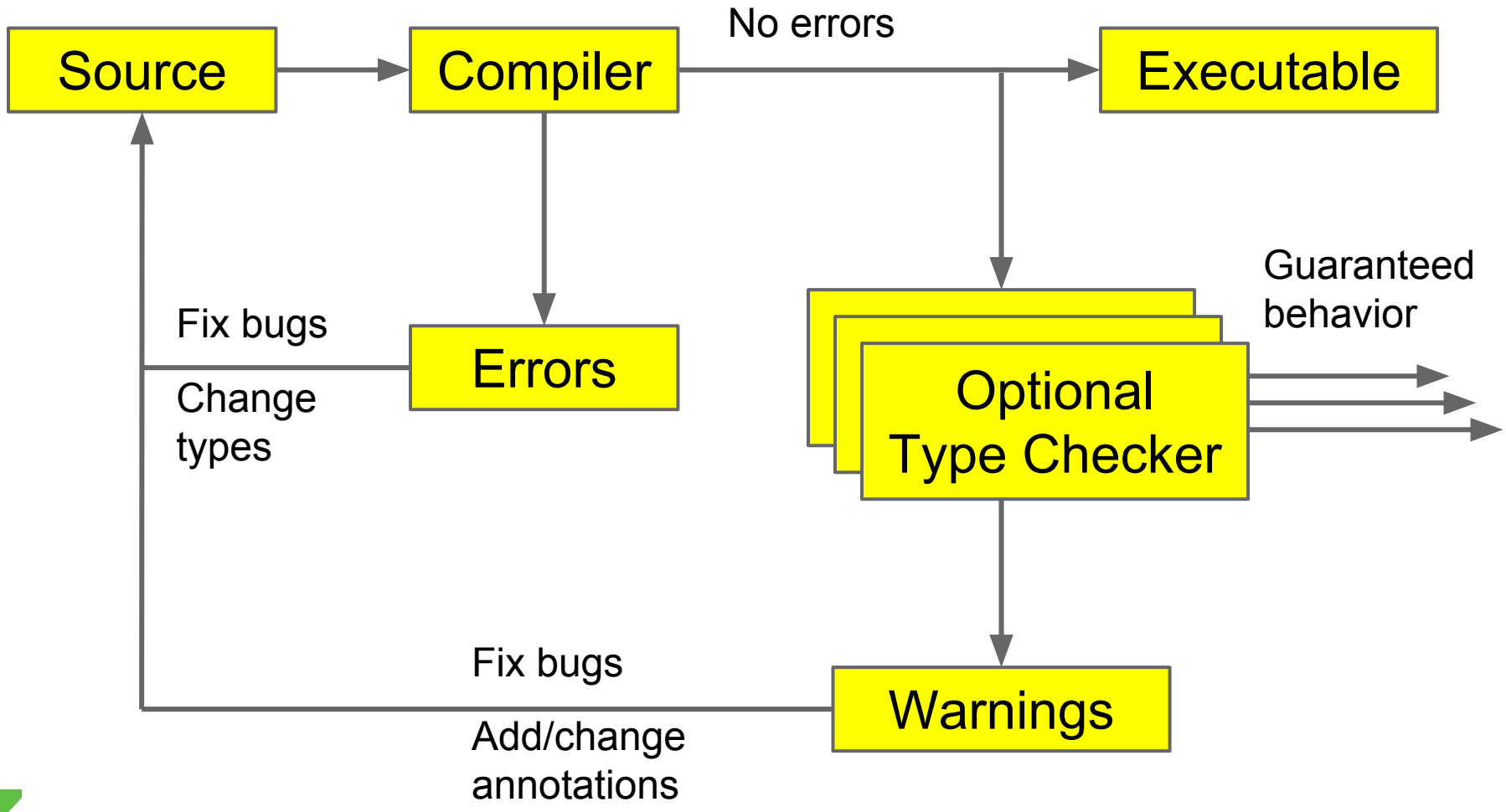
Type Checking



Optional Type Checking



Optional Type Checking



Prevent null pointer exceptions

Type system that statically guarantees that

the program only dereferences known non-null references

Types of data

`@NonNull`	reference is never null
`@Nullable`	reference may be null

Null pointer exception

```
String op(Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Null pointer exception

Where is the defect?

```
String op(Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Null pointer exception

Where is the defect?

```
String op(Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Null pointer exception

Where is the defect?

```
String op(Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Can’t decide without specification!

Let's use the Nullness Checker!

```
http://checkerframework.org/

wget http://types.cs.washington.
edu/checker-framework/current/checker-
framework-1.9.13.zip
```

Specification 1: non-null parameter

```
String op(@NonNull Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Specification 1: non-null parameter

```
String op(@NonNull Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);      // error
```

Specification 2: nullable parameter

```
String op(@Nullable Data in) {
  return “transform: ” + in.getF();
}
…
String s = op(null);
```

Specification 2: nullable parameter

```
String op(@Nullable Data in) {
  return “transform: ” + in.getF();
}                                  // error
…
String s = op(null);
```

Benefits of type systems

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

Benefits of type systems

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

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

Input Format Validation

Demo: ensure that certain strings contain **valid regular expressions**.

Regular Expression Example

```
public static void main(String[] args) {
  String regex = args[0];
  String content = args[1];
  Pattern pat = Pattern.compile(regex);
  Matcher mat = pat.matcher(content);
  if (mat.matches()) {
    System.out.println("Group: " + mat.group(1));
  }
}
```

Regular Expression Example

```
public static void main(String[] args) {
  String regex = args[0];
  String conten
  Pattern pat =
  Matcher mat = pat.matcher(content);
  if (mat.matches()) {
    System.out.println("Group: " + mat.group(1));
  }
}
```

PatternSyntaxException

IndexOutOfBoundsExceptionon

Fixing the Errors

`Pattern.compile` only on valid regex
`Matcher.group(i)` only if > i groups

```
...
if (!RegexUtil.isRegex(regex, 1)) {
  System.out.println("Invalid: " + regex);
  System.exit(1);
}
...
```

The Checker Framework

A framework for pluggable type checkers

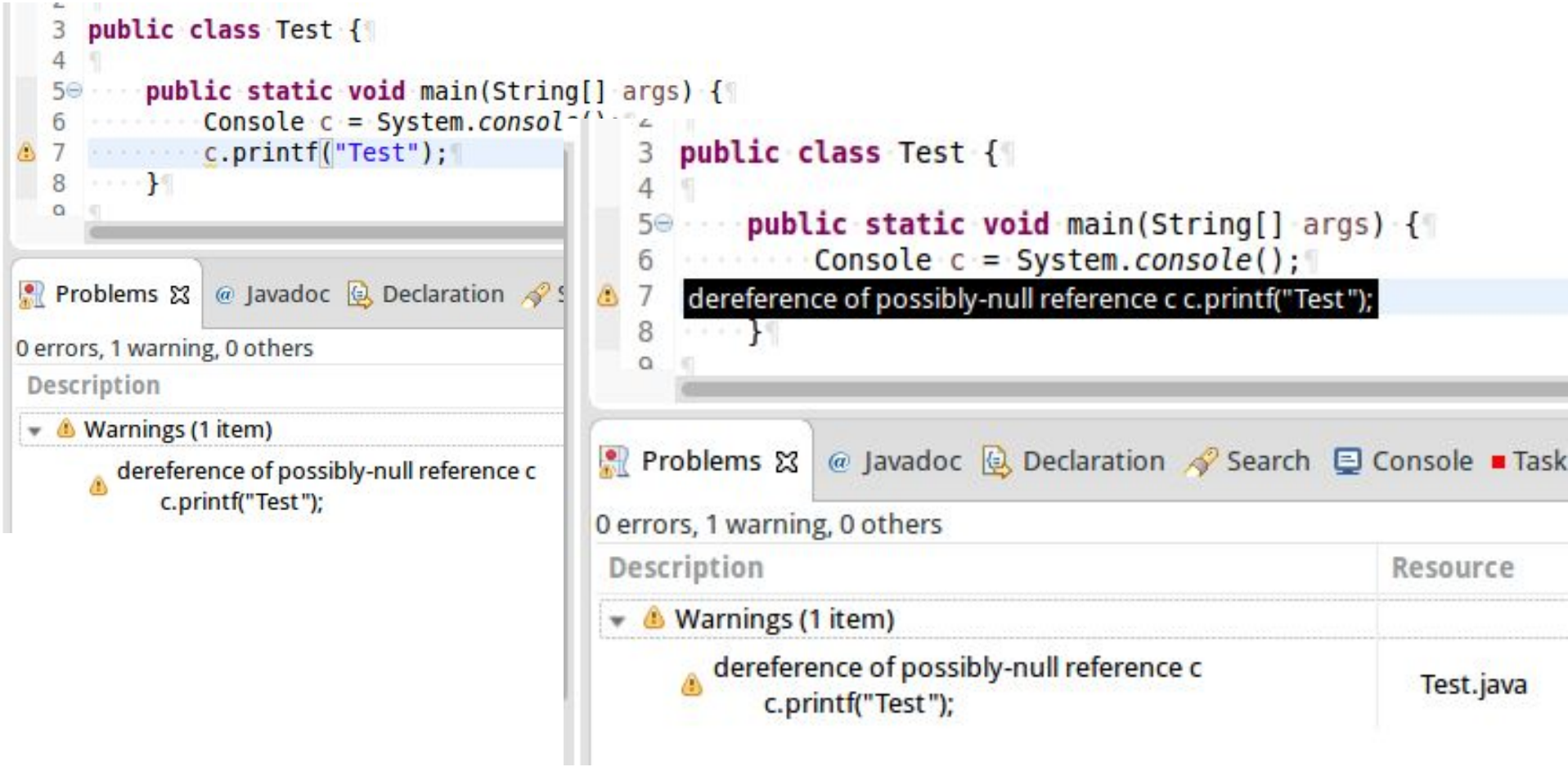
"Plugs" into the OpenJDK or OracleJDK compiler

```
javac -processor MyChecker ...
```

Standard error format allows tool integration

http://checkerframework.org/

Eclipse plug-in



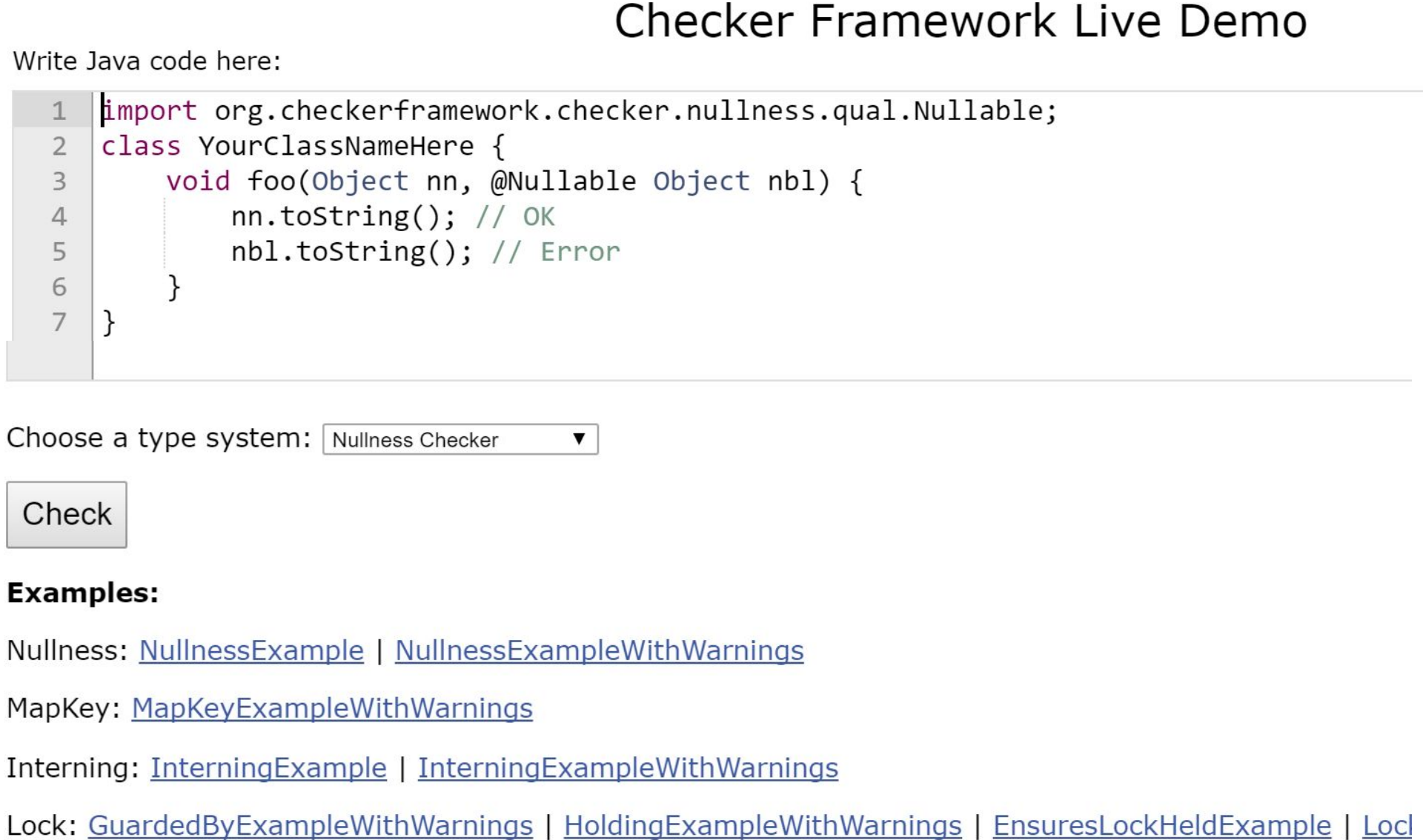
http://checkerframework.org/

Ant and Maven integration

```
<presetdef name="jsr308.javac">
  <javac fork="yes"
    executable="${checkerframework}/checker/bin/${cfJavac}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

```
<dependencies>
  ... existing <dependency> items ...
  <!-- annotations from the Checker Framework:
       nullness, interning, locking, ... -->
    <dependency>
      <groupId>org.checkerframework</groupId>
      <artifactId>checker-qual</artifactId>
      <version>1.9.7</version>
    </dependency>
</dependencies>
```

Web interface
http://eisop.uwaterloo.ca/live/



Example type systems

Null dereferences (`@NonNull`)

- >200 errors in Google Collections, javac, ...

Equality tests (`@Interned`)

- >200 problems in Xerces, Lucene, ...

Concurrency / locking (`@GuardedBy`)

- >500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Fake enumerations (`@Fenum`)

- problems in Swing, JabRef

String type systems

Regular expression syntax (`@Regex`)

- 56 errors in Apache, etc.; 200 annos

printf format strings (`@Format`)

- 104 errors, only 107 annotations required

Signature format (`@FullyQualified`)

- 28 errors in OpenJDK, ASM, AFU

Compiler messages (`@CompilerMessageKey`)

- 8 wrong keys in Checker Framework

Security type systems

Command injection vulnerabilities (`@OsTrusted`)

- 5 missing validations in Hadoop

Privacy (`@Source`)

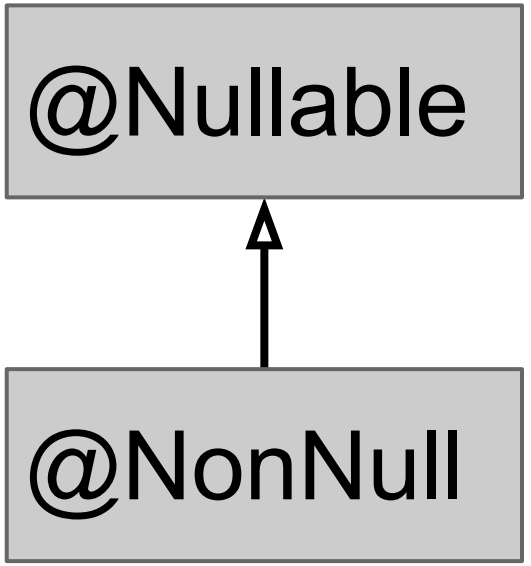
- SPARTA detected malware in Android apps

You can write your own checker!

Preventing Null-Pointer Exceptions

Basic type system:

`@Nullable`	might be null
`@NonNull`	non-null



Default is `@NonNull`

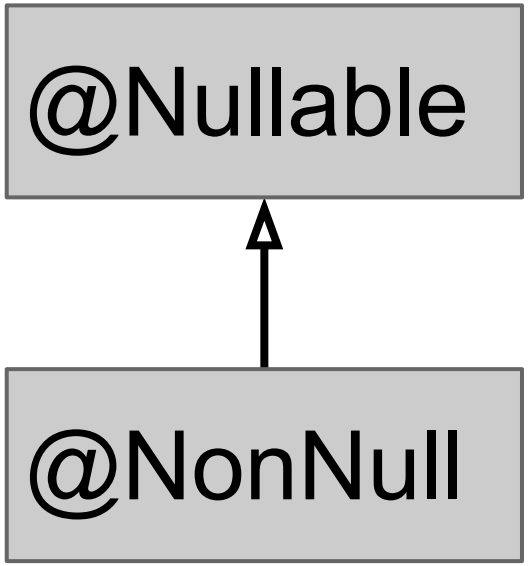
(Opposite of Java's assumption)

- Makes the dangerous case explicit
- Requires fewer annotations

CLIMB-to-top defaulting rule (applies to all type systems)

Top type is the default for:

- **C**asts
- **L**ocal variables
- **I**nstanceof
- i**M**plicit **B**ounds



Type refinement from assignments

```
myVar = new Foo();
```

Likewise for cast/instanceof expression

Top for implicit types allows every instantiation

Dynamic checks

```
if (x != null) {
  x.hashCode();
}

if (!RegexUtil.isRegex(userInput)) {
   throw new RuntimeException(...);
}
Pattern p = Pattern.compile(userInput);
```

One check for `null` is not enough

```
if (myField != null) {
  method1();
  myField.hashCode();
}
```

3 ways to express persistence across side effects:

```
@SideEffectFree void method1() { … }
@EnsuresNonNull(“myField”) method1()
{…}
@MonotonicNonNull myField;
```

Side effects

`@SideEffectFree`

Does not modify externally-visible state

`@Deterministic`

If called with == args again, gives == result

`@Pure`

Both side-effect-free and deterministic

The side-effect annotations are trusted, not checked

Lazy initialization and persistence across side effects

`@MonotonicNonNull`

Might be null or non-null
May only be (re-)assigned a non-null value

Purpose: avoid re-checking
    Once non-null, always non-null

Method pre- and post-conditions

Preconditions:

`@RequiresNonNull`

Postconditions:

`@EnsuresNonNull`

`@EnsuresNonNullIf`

```
@EnsuresNonNullIf(expression="#1", result=true)
public boolean equals(@Nullable Object obj) { ... }
```

Polymorphism over qualifiers

`@PolyNull`

Each occurrence is a use of an implicitly-defined type qualifier variable

`@PolyAll`

Same for all type systems

A non-null field might contain `null`

```
@NonNull String name;
... myObject.name ...
```

Initialization

`@Initialized` (constructor has completed)

`@UnderInitialization(Frame.class)`

Its constructor is currently executing

`@UnknownInitialization`

Might be initialized or under initialization

Map keys and `Map.get`

```
Map<String, @NonNull Integer> gifts;
… gifts.get(“pipers piping”) …
```

`Map.get` can return null unless

- value type is non-null, **and**
- argument key appears in the map

`@KeyFor` [rarely written, usually inferred]

Suppressing warnings

Because of Checker Framework false positives

```
@SuppressWarnings(“nullness”)
```

- Use smallest possible scope (e.g., local var)
- Write the rationale

```
assert x != null : “@AssumeAssertion(nullness)”;
```

More: http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#suppressing-warnings

Optional checks

`-Alint=redundantNullComparison`

Warns if comparing a non-null value to null

`-Alint=uninitialized`

Warns if the constructor does not initialize all fields (even primitives that have a default)

Building a checker is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {…}
@Encrypted String msg1 = ...;
send(msg1);   // OK
String msg2 = ....;
send(msg2);   // Warning!
```

Building a checker is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {...}
@Encrypted String msg1 = ...;
send(msg1);   // OK
String msg2 = ....;
send(msg2);   // Warning!
```

The complete checker:

```
@Target(ElementType.TYPE_USE)
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}
```

Encrypted Checker Demo

Let’s build it!

Defining a type system

1. Qualifier hierarchy
   - defines subtyping
2. Type introduction rules
   - types for expressions, declarations
3. Type rules
   - checker-specific errors
4. Flow-refinement
   - better types than the programmer wrote

Defining a type system

1. Qualifier hierarchy
   - subtyping, assignments

```
@SubtypeOf(UnknownRegex.class)
public @interface Regex {
```

Defining a type system

2. Type introduction rules
   - types for expressions, declarations

```
@ImplicitFor( trees = {
      Tree.Kind.NEW_CLASS,
      Tree.Kind.NEW_ARRAY, ... })
@DefaultQualifierInHierarchy
@DefaultForUnannotatedCode({
      DL.PARAMETERS, DL.LOWER_BOUNDS })
```

Defining a type system

3. Type rules
   - checker-specific errors

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

Defining a type system

4. Flow-refinement
   - better types than the programmer wrote

```
if (ElementUtils.matchesElement(method,
        IS_REGEX_METHOD_NAME,
        String.class, int.class)) {
    …
}
```

Testing infrastructure

jtreg-based testing as in OpenJDK

Lightweight tests with in-line expected errors:

```
String s = "%+s%";
//:: error: (format.string.invalid)
f.format(s, "illegal");
```

Tips

- Start by type-checking part of your code
- Only type-check properties that matter to you
- Use subclasses (not type qualifiers) if possible
- Write the spec first (and think of it as a spec)
- Avoid warning suppressions when possible
- Avoid raw types such as `List`; use `List<String>`

Checkers are usable

- Type-checking is familiar to programmers
- Modular: fast, incremental, partial programs
- Annotations are not too verbose
  - **`@NonNull`**: 1 per 75 lines
  - **`@Interned`**: 124 annotations in 220 KLOC revealed 11 bugs
  - **`@Format`**: 107 annotations in 2.8 MLOC revealed 104 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
- Few false positives
- First-year CS majors preferred using checkers to not
- **Practical**: in daily use at Google, on Wall Street, etc.

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 (but handles 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

Java 8 extends annotation syntax

Annotations on all occurrences of types:

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

Stored in classfile

Handled by javac, javap, javadoc, …

Java 6 & 7 compatibility

Annotations in comments:

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

(Requires use of jsr308-langtools compiler.)

Annotating external libraries

When type-checking clients, need library spec

Can write manually or automatically infer

Two syntaxes:

- As separate text file (stub file)
- Within its .jar file (from annotated partial source code)

Checker Framework facilities

- Full type systems: inheritance, overriding, ...
- Generics (type polymorphism)
  - Also qualifier polymorphism
- Qualifier defaults
- Dataflow framework
- Pre-/post-conditions
- Warning suppression
- Testing infrastructure

Verification

- **Goal**:
  prove that no bug exists
- **Specifications**:
  user provides
- **False negatives**:
  none
- **False positives**:
  user suppresses warnings
- **Downside**:
  user burden

Bug-finding

- **Goal**:
  find some bugs at low cost
- **Specifications**:
  infer likely specs
- **False negatives:**
  acceptable
- **False positives**:
  heuristics focus on most important bugs
- **Downside**:
  missed bugs

Neither is “better”; each is appropriate in certain circumstances.

Community

Open source project:

`https://github.com/typetools/checker-framework`

Community:

- uWashington: Michael Ernst, Suzanne Millstein, Javier Thaine, Dan Brown …
- uWaterloo: Werner Dietl, Jeff Luo, Jason Li, Mier Ta, Charles Chen, …
- Bug reports, test cases, patches, … from users

Conclusions

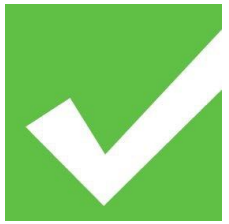
Checker Framework for creating type checkers

- Featureful, effective, easy to use, scalable

Prevent bugs at compile time

Create custom type-checkers

Improve your code!



http://CheckerFramework.org/

@CheckerFrmwrk on Twitter
CheckerFramework on Facebook & Google+