Preventing Errors Before They Happen

The Checker Framework

http://CheckerFramework.org/
Twitter: @CheckerFrmwrk
Live demo: http://CheckerFramework.org/live/

Werner Dietl, University of Waterloo
Michael Ernst, University of Washington
Motivation

java.lang.NullPointerException
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

1997: **225 deaths**: jet crash caused by radar software
1991: **28 deaths**: Patriot missile guidance system
2003: **11 deaths**: blackout
1985-2000: **>8 deaths**: Radiation therapy

2011: Software caused 25% of all medical device recalls
Outline

- Verification approach: Pluggable type-checking
- Tool: Checker Framework
- How to use it
- Creating a custom type system
Java's type system is too weak

Type checking prevents many errors

```java
int i = "hello";
```

Type checking doesn't prevent **enough** errors

```java
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
   NullPointerException
   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.unsupportedOperationException
Collections.emptyList().add("one");
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);

Corrupted map
Some errors are silent

dbStatement.executeQuery(userInput);
Some errors are silent

dbStatement.executeQuery(userInput);

SQL injection attack

Initialization, data formatting, equality tests, …
SQL injection attack

Goal: don’t execute user input as a SQL command

private String wrapQuery(String s) {
    return "SELECT * FROM User WHERE userId='' + s + ''";
}

If a user inputs his name as: ' or 'x'='x
the SQL query is: ... WHERE userID='' or 'x'='x'

To prevent errors: sanitize user data before use
Vulnerable code

void op(String in) {
    ...
    executeQuery(in);
}

...
Vulnerable code

void op(String in) {
    ...
    executeQuery(in);
}
...
...
op(userInput);
Vulnerable code

```
void op(String in) {
    ...
    executeQuery(in);
}
...
...
op(userInput);
```

Where is the defect?
Vulnerable code

void op(String in) {
    ... 
    executeQuery(in);
} 

Can't decide without specification!

op(userInput);

Where is the defect?
void op(@Untainted String in) {
    ... 
    executeQuery(in);
}

... 
... 

op(userInput);
void op(@Untainted String in) {
    ... 
    executeQuery(in);
}
...
op(userInput);  // error
Specification 2: tainted parameter

```java
void op(@Tainted String in) {
    ...;
    executeQuery(in);
}
...
...
op(userInput);
```
void op(@Tainted String in) {
    ...
    executeQuery(in); // error
}
...
op(userInput);
Demo: Preventing SQL injection

Goal: don’t execute user input as a SQL command

private String wrapQuery(String s) {
    return "SELECT * FROM User WHERE userId='" + s + "'";
}

If a user inputs his name as: ' or 'x'='x
the SQL query is: ... WHERE userID='' or 'x'='x'

@Tainted = might be untrusted user input
@Untainted = sanitized, safe to use
Verification approach: Pluggable Type Checking

1. Design a type system to solve a specific problem
2. Write type qualifiers in code (or, use type inference)
   
   ```java
   @Immutable Date date = new Date();
   date.setSeconds(0);  // compile-time error
   ```

3. Type checker warns about violations (bugs)

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

   ```java
   MyFile.java:149: dereference of possibly-null reference bb2
   allVars = bb2.vars;
   ^
   ```
Type Checking

Source → Compiler → Executable

No errors

Errors

Fix bugs
Change types
Optional Type Checking

Source → Compiler → No errors → Executable

Errors:
- Fix bugs
- Change types

Warnings:
- Fix bugs
- Add/change annotations

Optional Type Checker

Guaranteed behavior
Optional Type Checking

Source → Compiler → Executable

Errors → Optional Type Checker → Warnings

Fix bugs
Change types

No errors
Guaranteed behavior

Fix bugs
Add/change annotations
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

```java
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?

```java
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();
}

Can’t decide without specification!

... String s = op(null);
Specification 1: non-null parameter

String op(@NonNull Data in) {
    return "transform: " + in.getF();
}

... String s = op(null);
Specification 1: non-null parameter

```java
String op(@NonNull Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);       // error
```
Specification 2: nullable parameter

```java
String op(@Nullable Data in) {
    return "transform: " + in.getF();
}

String s = op(null);
```
String op(@Nullable Data in) {
    return "transform: " + in.getLog();
} // error

... String s = op(null);
Nullness demo

- Detect errors
- Guarantee the absence of errors
- Verify the correctness of optimizations
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)
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
Ant, Maven, Gradle integration

```xml
<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>
```
Eclipse, IntelliJ, NetBeans integration
Comparison: other nullness tools

<table>
<thead>
<tr>
<th></th>
<th>Null pointer errors</th>
<th>False warnings</th>
<th>Annotations written</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Found</td>
<td>Missed</td>
<td></td>
</tr>
<tr>
<td>Checker Framework</td>
<td>9</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>FindBugs</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Jlint</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>PMD</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Eclipse, in 2017</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Intellij (@NotNull default), in 2017</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Checking the Lookup program for file system searching (4kLOC)
Live demo:  http://CheckerFramework.org/live/

Checker Framework Live Demo

Write Java code here:

```java
import org.checkerframework.checker.nullness.qual.Nullable;

class YourClassNameHere {
    void foo(Object nn, @Nullable Object nbl) {
        nn.toString(); // OK
        nbl.toString(); // Error
    }
}
```

Choose a type system: Nullness Checker

**Examples:**

Nullness: NullnessExample | NullnessExampleWithWarnings

MapKey: MapKeyExampleWithWarnings

Interning: InterningExample | InterningExampleWithWarnings

Lock: GuardedByExampleWithWarnings | HoldingExampleWithWarnings | EnsuresLockHeldExample | Loc
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 / typedefs (@Fenum)
   problems in Swing, JabRef
String type systems

Regular expression syntax (@Regex)
56 errors in Apache, etc.; 200 annos required

Printf format strings (@Format)
104 errors, only 107 annotations required

Method 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
Information flow privacy (@Source)
  SPARTA detected malware in Android apps

It’s easy to write your own type system!
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 use in Silicon Valley, 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 (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
Formalizations

\[
\begin{align*}
P & \in \text{Program} ::= \text{Class, ClassId, Expr} \quad \text{Ps} \in \text{Fields} \\
\text{Cls} & \in \text{Class} ::= \text{class ClassId<TVarId> Fs} \in \text{Fields} \\
\text{extends ClassId<TVarId>} & \in \text{OwnerAddr ClassId<TVarId>} \\
\{ FieldId \text{<Type}; \text{Met} \} & \in \text{Addr} \cup \{ \text{null} \} \\
\text{rT} & \in \text{rType} = \text{omi} \\
o & \in \text{Obj} = \text{rType, Fields} \\
\text{rEnv} & = \text{rType; ParId Addr} \\
\text{OS-Read} & = \text{h, r\Gamma, e0 \sim h', \nu0} \\
\nu0 & \neq \text{null} \\
h0 & \sim \text{h2} \\
h' & = \text{h2}[(\nu0.f := \nu)] \\
h, r\Gamma, e0.f := e2 & \sim h', \nu \\
\text{GT-Read} & = \Gamma \vdash e0 : N0 \\
\text{N0} & = u0 \text{ C0}</> \\
\text{T1} & = \text{fType(C0, f)} \\
\Gamma & \vdash e2 : N0>T1 \\
u0 & \neq \text{any} \quad \text{rp(u0, T1)} \\
\Gamma & \vdash e0.f := e2 : N0>T1 \\
\text{DYN} & = \text{free_\text{rType}(C_\text{N})} \\
\text{free(sT)} & \subseteq \text{dom(C_N)} \\
dyn(sT, \nu, rT, (X \ rT'; \nu)) & = sT' | /\text{this}, \nu' | /\text{peer}, \nu' | /\text{rep}, \text{any}_a | /\text{any}_a, rT'/X \ rT'/X' \\
h, r\Gamma, e0 \sim h', \nu0
\end{align*}
\]
Demo: regular expression errors

@Regex = valid regular expression
OK: “colou?r”
NOT: “1) first point”

@Regex(2) = has 2+ capturing groups
OK: "((Linked)?Hash)?Map"
OK: "(http|ftp)://([^/]+)(/.*))?"
NOT: "(brown|beige)"
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

```java
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));
    }
}
```
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);  
}
 ...

Since Java 5: declaration annotations

Only for declaration locations:

```java
@Deprecated
class Foo {
    @Getter @Setter private String query;
    @SuppressWarnings("unchecked")
    void foo() {
        ...
    }
}
```
But we couldn’t express

A **non-null** reference to my data

An **interned** string

A **non-null** List of **English** strings

A **non-empty** array of **English** strings
With Java 8 Type Annotations we can!

A non-null reference to my data
   @NonNull Data mydata;
An interned String
   @Interned String query;
A non-null List of English Strings
   @NonNull List<@English String> msgs;
A non-empty array of English strings
   @English String @NonEmpty [] a;
Java 8 extends annotation syntax

Annotations on all occurrences of types:

```java
@Untainted String query;
List<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
(or avoid dependency on Checker Framework)

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
- Pre-/post-conditions
- Warning suppression
Brainstorming new type checkers

What runtime exceptions to prevent?
What properties of data should always hold?
What operations are legal and illegal?

Type-system checkable properties:
- Dependency on values
- Not on program structure, timing, ...
Example: Nullness Checker

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?
Example: Nullness Checker

What runtime exceptions to prevent?
- NullPointerException

What properties of data should always hold?

What operations are legal and illegal?
Example: Nullness Checker

What runtime exceptions to prevent?

NullPointerException

What properties of data should always hold?

@NonNull references always non-null

What operations are legal and illegal?
Example: Nullness Checker

What runtime exceptions to prevent?

NullPointerException

What properties of data should always hold?

@NonNull references always non-null

What operations are legal and illegal?

Dereferences only on @NonNull references
Example: Regex Checker

What runtime exceptions to prevent?

What properties of data should always hold?

What operations are legal and illegal?
Example: Regex Checker

What runtime exceptions to prevent?

- PatternSyntaxException
- IndexOutOfBoundsException

What properties of data should always hold?

What operations are legal and illegal?
Example: Regex Checker

What runtime exceptions to prevent?

- PatternSyntaxException,
- IndexOutOfBoundsException

What properties of data should always hold?

- Whether a string is a regex and number of groups

What operations are legal and illegal?
Example: Regex Checker

What runtime exceptions to prevent?

- PatternSyntaxException,
- IndexOutOfBoundsException

What properties of data should always hold?

- Whether a string is a regex and number of groups

What operations are legal and illegal?

- Pattern.compile with non-@Regexp, etc,
New type system

What runtime exceptions to prevent?
1 NumberFormatException

What properties of data should always hold?
2 @ValidNumber vs. @Unknown

What operations are legal and illegal?
3 Integer.valueOf only with @ValidNumber
New type system

What runtime exceptions to prevent?
   1 Class/MethodNotFoundException

What properties of data should always hold?
   2 @ClassName, @MethodName

What operations are legal and illegal?
   3 Class.forName only on @ClassName
New type system

What runtime exceptions to prevent?
  1. UnsupportedOperation from Lists

What properties of data should always hold?
  2. @Mutable vs @UnknownMutable

What operations are legal and illegal?
  3. .add .remove only on @Mutable Lists
New type system

What runtime exceptions to prevent?
1  Invalid quoting of strings

What properties of data should always hold?
2  @JSTainted @HTMLTainted etc

What operations are legal and illegal?
3  println, displayHTML
New type system

What runtime exceptions to prevent?
  1. Unknown serializer exception

What properties of data should always hold?
  2. `@JsonSerializable` vs. `@UnknownSerializable`

What operations are legal and illegal?
  3. `.toJson` only on `@JsonSerializable`
Building a checker is easy

Example: Ensure encrypted communication

```java
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

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

The complete checker:

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

Let’s build it!
Testing infrastructure

jtreg-based testing as in OpenJDK

Lightweight tests with in-line expected errors:

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

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

1. Qualifier hierarchy
   - subtyping, assignments

```java
@SubtypeOf(UnknownRegex.class)
public @interface Regex {
    @UnknownRegex
}
```
Defining a type system

2. Type introduction rules
   - types for expressions

```java
Data d = new Data();

@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

```java
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

```java
if (ElementUtils.matchesElement(method, IS_REGEX_METHOD_NAME, String.class, int.class)) {
    ...
}
```
Dataflow Framework

Goal: Compute properties about expressions
- More accurate types than the user wrote
- Foundation for other static analyses
  - e.g. by Google error-prone and Uber NullAway

Dataflow Framework user provides
- What are we tracking?
- What do operations do?
- What are intermediate results?

Dataflow Framework does all the work!
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>
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.
Checker Framework Community

Open source project:
https://github.com/typetools/checker-framework

- Monthly release cycle
- >12,800 commits, 75 authors
- Welcoming & responsive community
Checker Framework plans

More type systems:
- Immutability
- Index-out-of-bounds
- Optional<T> type
- Determinism
- Signed vs. unsigned numbers

Type inference
Combined static & dynamic enforcement
More at JavaOne 2017

Using Type Annotations to Improve Your Code

BoF3048, tonight, 18:30 to 19:15
Moscone West - Room 2014
Pluggable type-checking improves code

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/