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

Preventing errors before they happen: Lightweight verification via pluggable type-checking

Michael D. Ernst

University of Washington (Seattle, WA, USA)

University of Buenos Aires

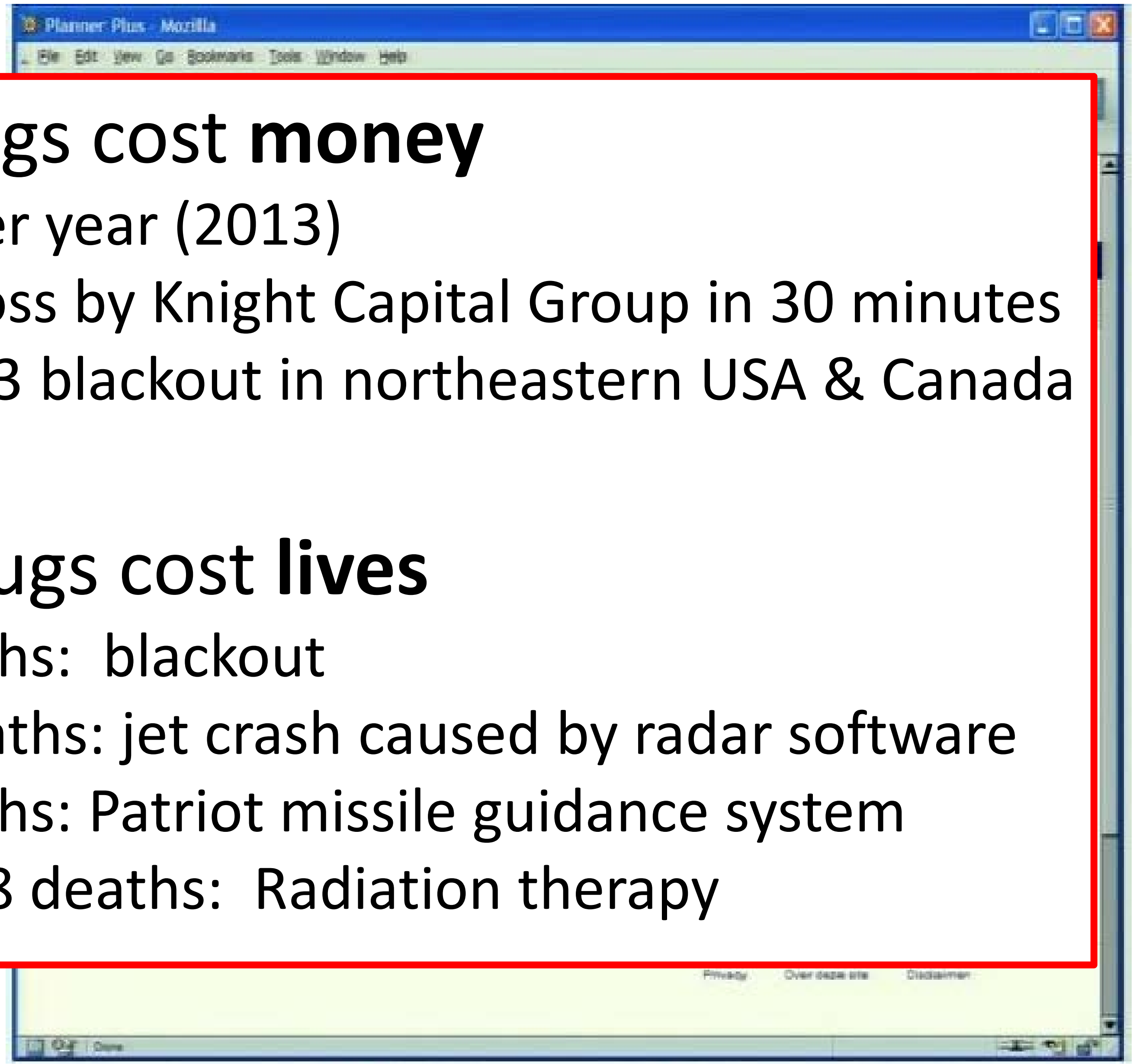
Joint work with Werner Dietl and many others

http://CheckerFramework.org/

Schedule

- Part 1 (11:00 – 12:30)
  - pluggable type-checking: what and why
  - demo of the Checker Framework
  - relevance to your programming problems
- Part 2 (14:00 – 15:30)
  - how to create your own type system
  - hands-on practice in using pluggable types

Motivation



Sofware bugs cost **money**

$312 billion per year (2013)

$440 million loss by Knight Capital Group in 30 minutes

$6 billion: 2003 blackout in northeastern USA & Canada

Software bugs cost **lives**

2003: 11 deaths: blackout

1997: 225 deaths: jet crash caused by radar software

1991: 28 deaths: Patriot missile guidance system

1985-2000: >8 deaths: Radiation therapy

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();
```

$\Rightarrow$ NullPointerException

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

$\Rightarrow$ 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, …

Goal: Find errors at **compile time**
… before testing, customers, or hackers find them

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
- Verification vs. bug finding
- Conclusion

Type qualifiers

- **In Java 8**: annotations on types

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

- **Backward-compatible**: 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 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
- Verification vs. bug finding
- Conclusion

Using a checker

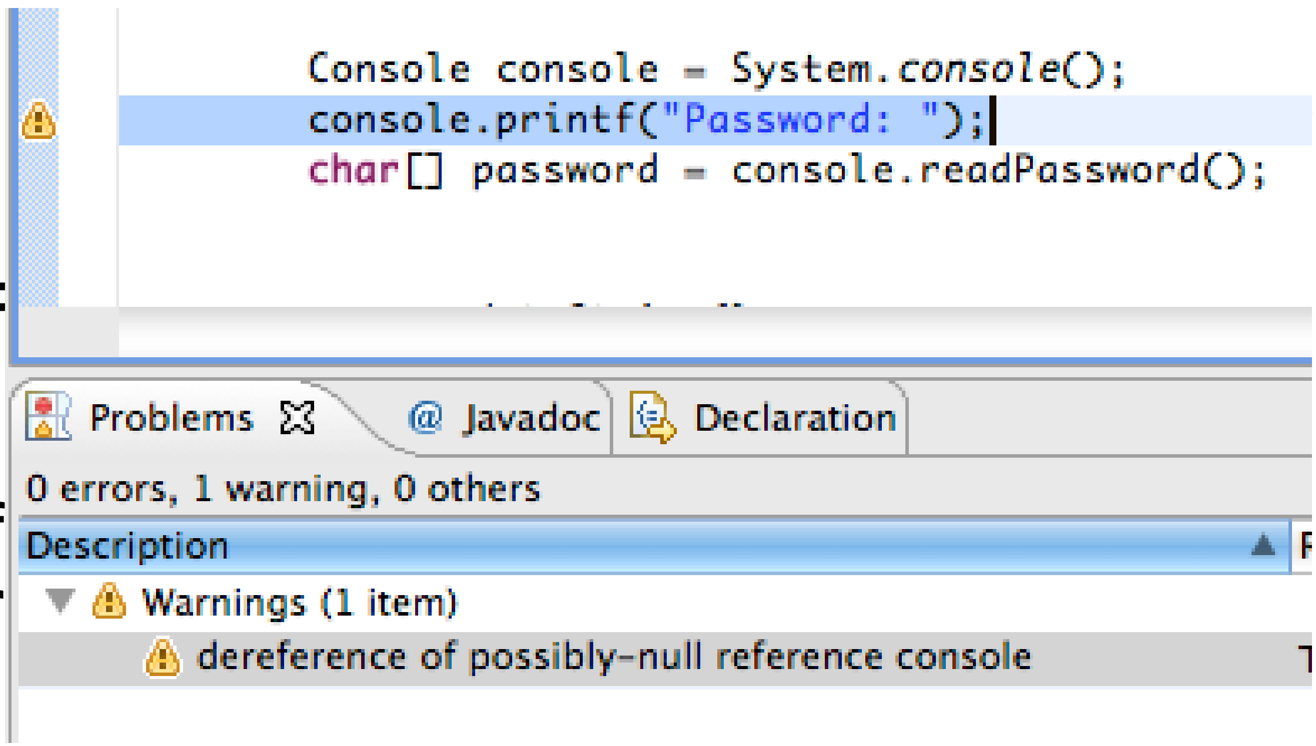
- Run in IDE or on command line
- Works as a compiler plug-in (annotation processor)
- Familiar workflow and error messages

% j File.java

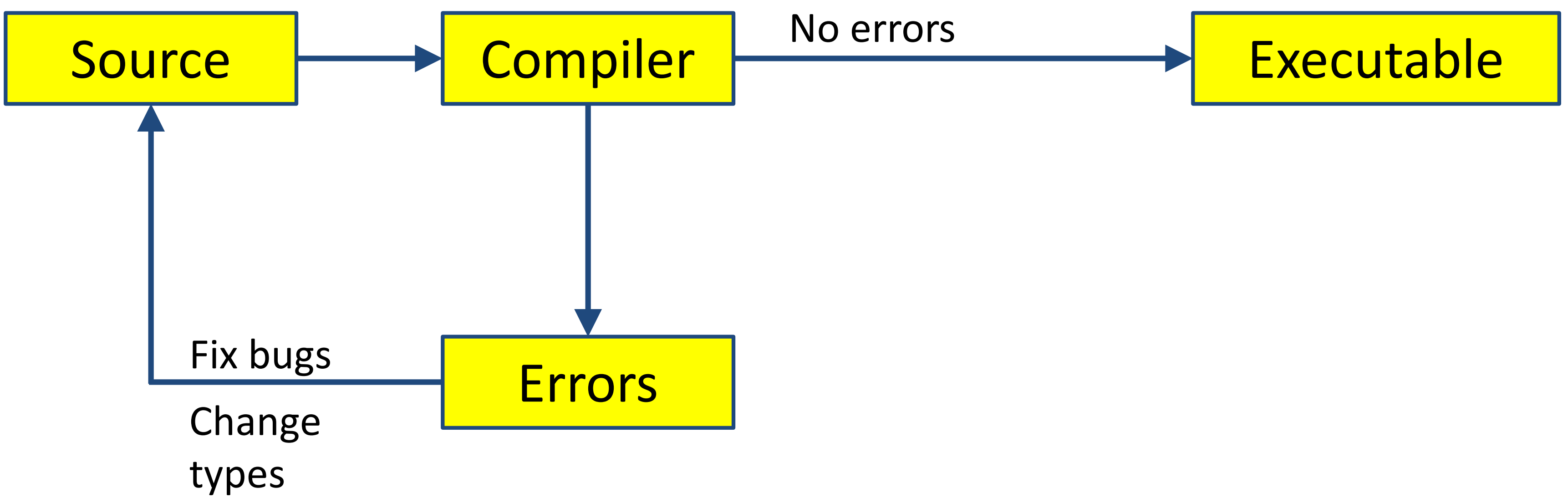
MyF

f

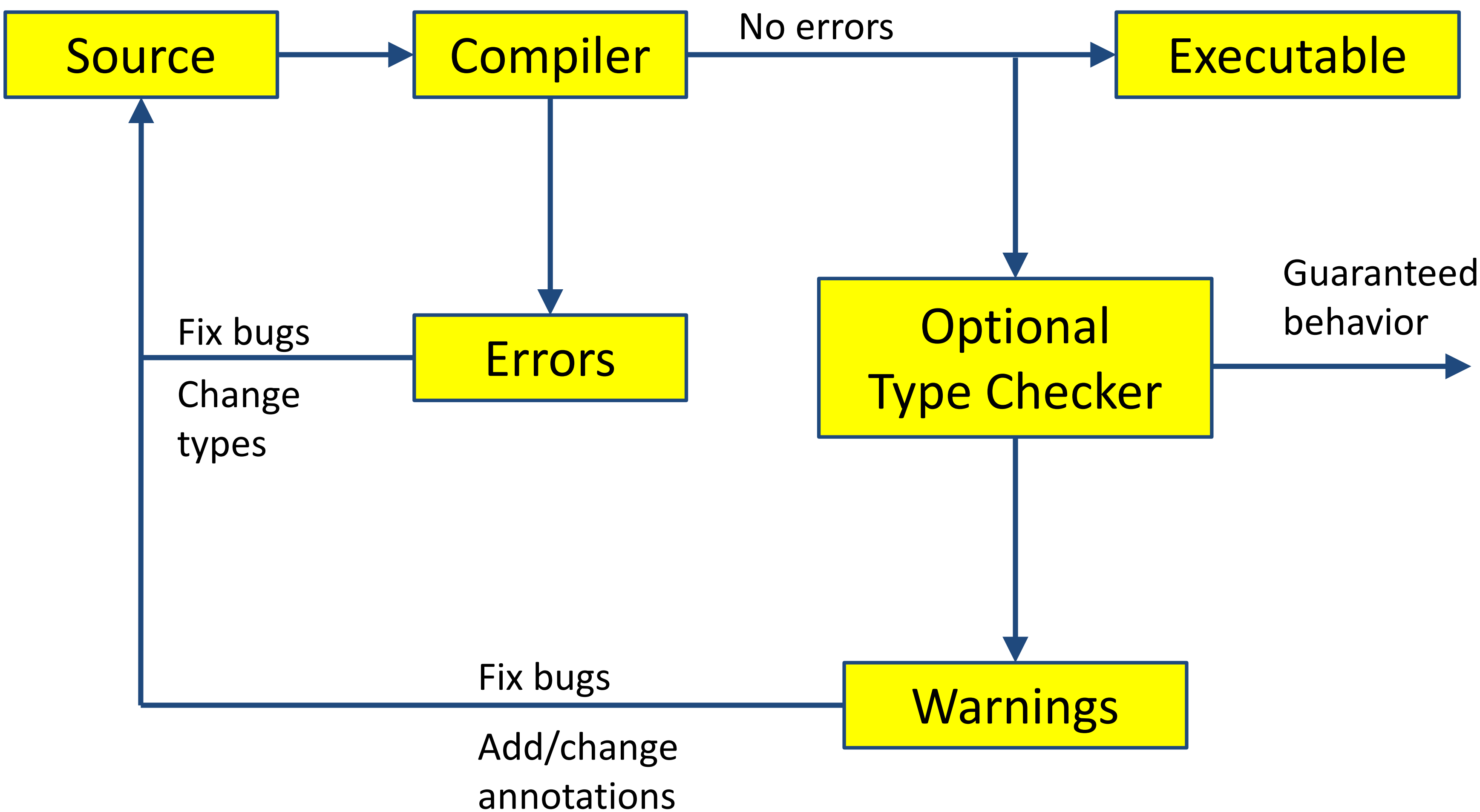
r



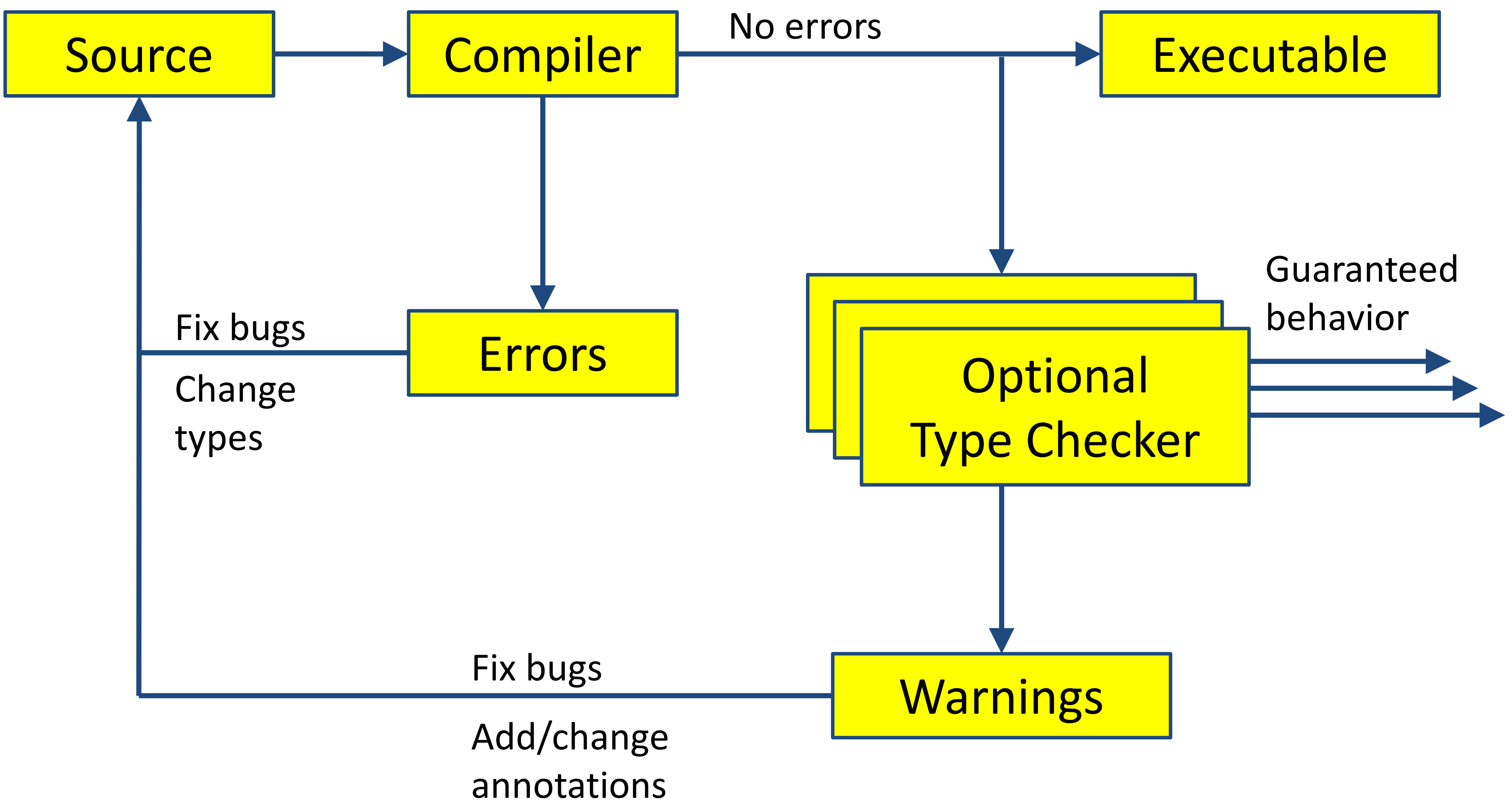
Type Checking



Optional Type Checking



Optional Type Checking



Nullness and mutation demo

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

Checkers are effective

Practical: in daily use at Google, on Wall Street, etc.
Scalable: > 6 MLOC checked at UW

Selected case study results:

- Signature strings: 28 errors in OpenJDK, ASM, AFU
- Nullness: >200 errors in Google Collections, javac, Daikon
- Interning: >200 problems in Xerces, Lucene
- Format strings: 104 errors, only 107 annotations required
- Regular expressions: 56 errors in Apache, etc.; 200 annos
- Fake enumerations: problems in Swing, JabRef
- Compiler messages: 8 wrong keys in Checker Framework

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)
- False warnings are suppressed via an annotation or assertion

Checkers are featureful

- Full type systems: inheritance, overriding, generics (type polymorphism), etc.
- Type qualifier polymorphism
- Flow-sensitive type qualifier inference
  - no need to write annotations within method bodies
- Qualifier defaults
- Pre-/post-conditions, side effect annotations
- Warning suppression

Checkers are usable

- Integrated with toolchain
  - javac, Eclipse, Ant, Maven
- Annotations are not too verbose
  - `@NonNull`: 1 per 75 lines
    - with program-wide defaults, 1 per 2000 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
- Inference tools add annotations to your program
- Few false positives
- First-year CS majors preferred using checkers to not

What a checker guarantees

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

Formalizations

$$
\begin{array}{llll}
\mathtt{h} & \in & \mathtt{Heap} & = \mathtt{Addr} \to \mathtt{Obj} \\
\iota & \in & \mathtt{Addr} & = \text{Set of Addresses} \cup \{\mathtt{null}_a\} \\
\mathtt{o} & \in & \mathtt{Obj} & = {}^{r}\mathtt{Type},\ \mathtt{Fields} \\
{}^{r}\mathtt{T} & \in & {}^{r}\mathtt{Type} & = \mathtt{OwnerAddr}\ \mathtt{ClassId}\texttt{<}\overline{{}^{r}\mathtt{Type}}\texttt{>} \\
\mathtt{Fs} & \in & \mathtt{Fields} & = \mathtt{FieldId} \to \mathtt{Addr} \\
\iota & \in & \mathtt{OwnerAddr} & = \mathtt{Addr} \cup \{\mathtt{any}_a\} \\
{}^{r}\Gamma & \in & {}^{r}\mathtt{Env} & = \overline{\mathtt{TVarId}\ {}^{r}\mathtt{Type}};\ \overline{\mathtt{ParId}\ \mathtt{Addr}}
\end{array}
$$

$$
\begin{array}{llll}
\mathtt{P} & \in & \mathtt{Program} & ::= \overline{\mathtt{Class}},\ \mathtt{ClassId},\ \mathtt{Expr} \\
\mathtt{Cls} & \in & \mathtt{Class} & ::= \mathtt{class\ ClassId}\texttt{<}\overline{\mathtt{TVarI}} \\
 & & & \quad \mathtt{extends\ ClassId}\texttt{<}\overline{{}^{s}\mathtt{Typ}} \\
 & & & \quad \{\ \overline{\mathtt{FieldId}\ {}^{s}\mathtt{Type};}\ \mathtt{Met} \\
{}^{s}\mathtt{T} & \in & {}^{s}\mathtt{Type} & ::= {}^{s}\mathtt{NType} \mid \mathtt{TVarId} \\
{}^{s}\mathtt{N} & \in & {}^{s}\mathtt{NType} & ::= \mathtt{OM}\ \mathtt{ClassId}\texttt{<}\overline{{}^{s}\mathtt{Type}}\texttt{>} \\
\mathtt{u} & \in & \mathtt{OM} & ::= \\
\mathtt{mt} & \in & \mathtt{Meth} & ::= \\
 & & \mathtt{MethSig} & ::= \\
\mathtt{w} & \in & \mathtt{Purity} & ::= \\
\mathtt{e} & \in & \mathtt{Expr} & ::= \\
 & & & \quad \mathtt{Expr.MethId}\texttt{<}{}^{s}\mathtt{Type}\texttt{>}(\mathtt{Expr}) \mid \\
 & & & \quad \mathtt{new}\ {}^{s}\mathtt{Type} \mid ({}^{s}\mathtt{Type})\ \mathtt{Expr} \\
{}^{s}\Gamma & \in & {}^{s}\mathtt{Env} & ::= \overline{\mathtt{TVarId}\ {}^{s}\mathtt{NType}};\ \overline{\mathtt{ParId}\ {}^{s}\mathtt{Type}}
\end{array}
$$

$$
\text{OS-Upd}\ \frac{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \mathtt{h}_0, {}^{r}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \quad \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 \rightsquigarrow \mathtt{h}',}
$$

$$
\text{OS-Read}\ \frac{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \iota = \mathtt{h}'(\iota_0)\downarrow_2(\mathtt{f})}{\mathtt{h}, {}^{r}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota}
$$

$$
\text{GT-Upd}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \quad \mathtt{N}_0 = \mathtt{u}_0\ \mathtt{C}_0\texttt{<\_>} \quad \mathtt{T}_1 = \mathit{fType}(\mathtt{C}_0, \mathtt{f}) \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{N}_0 \triangleright \mathtt{T}_1 \quad \mathtt{u}_0 \neq \mathtt{any} \quad \mathit{rp}(\mathtt{u}_0, \mathtt{T}_1)}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 : \mathtt{N}_0 \triangleright \mathtt{T}_1}
$$

$$
\text{GT-Read}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \_\ (}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{N}_0 \triangleright \mathit{fType}(\mathtt{C}_0, \mathtt{f})}
$$

$$
\left.\begin{array}{l}
\mathtt{h} \vdash {}^{r}\Gamma : {}^{s}\Gamma \\
\mathtt{h} \vdash \iota_1 : \mathit{dyn}({}^{s}\mathtt{N}, \mathtt{h}, {}^{r}\Gamma) \\
\mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{s}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1) \\
{}^{s}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\texttt{<>} \\
\mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{r}\Gamma(\mathtt{this}) \\
\mathit{free}({}^{s}\mathtt{T}) \subseteq \mathit{dom}(\mathtt{C}_N)
\end{array}\right\} \implies \mathtt{h} \vdash \iota_2 : \mathit{dyn}({}^{s}\mathtt{N} \triangleright {}^{s}\mathtt{T}, \mathtt{h}, {}^{r}\Gamma)
$$

$$
\text{DYN}\ \frac{{}^{r}\mathtt{T} = \iota'\ \_\texttt{<>} \quad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}}\texttt{>} \quad \iota \vdash {}^{r}\mathtt{T}\ {}^{r}\!\!<:\ \iota'\ \mathtt{C}\texttt{<}\overline{{}^{r}\mathtt{T}_a}\texttt{>} \Rightarrow \iota \vdash \overline{{}^{r}\mathtt{T}}\ {}^{r}\!\!<:\ \overline{{}^{r}\mathtt{T}_a} \quad \mathtt{dom}(\mathtt{C}) = \overline{\mathtt{X}} \quad \mathtt{free}({}^{s}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}}{\mathit{dyn}({}^{s}\mathtt{T}, \iota, {}^{r}\mathtt{T}, (\overline{\mathtt{X}'\ {}^{r}\mathtt{T}'}; \_)) = {}^{s}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{r}\mathtt{T}/\mathtt{X}}, \overline{{}^{r}\mathtt{T}'/\mathtt{X}'}]}
$$

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

What bugs can you detect & prevent?

The property you care about:	The annotation you write:
Null dereferences	**`@NonNull`**
Mutation and side-effects	**`@Immutable`**
Concurrency: locking	**`@GuardedBy`**
Security: encryption, tainting	**`@Encrypted`** **`@OsTrusted`**, **`@Untaint…`**
Aliasing	**`@Linear`**
Equality tests	**`@Interned`**
Strings: localization, regular expression syntax, signature representation, format string syntax	**`@Localized`** **`@Regex`** **`@FullyQualified`** **`@Format`**
Enumeractions	**`@Fenum`**
Typestate (e.g., open/closed files)	**`@State`**

Users can write their own checkers!

Outline

- Type qualifiers
- Pluggable type checkers
- **Writing your own checker**
- Verification vs. bug finding
- Conclusion

Example: Regular expressions

```
// Prints the first matching group.
// For example:
//   java RegexExample ([0-9]*):([0-9]*)  23:59
// prints “Group 1 = 23”

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 1 = "
                       + mat.group(1));
  }
}
```

PatternSyntaxException

IndexOutOfBoundsException

Regular expression type system

- What runtime errors to prevent?
  PatternSyntaxException and
  IndexOutOfBoundsException.
- What operations are legal?
  Pattern.compile only on valid regex.
  Matcher.group(i) only if >i groups.
- What properties of data should hold?
  Strings: valid regex vs. invalid.
  Number of groups in a regex.

Example: Encrypted communication

```
void send(@Encrypted String msg) {…}

@Encrypted String msg1 = ...;
send(msg1);   // OK

String msg2 = ...;
send(msg2);   // Warning!
```

Encryption type system

- What runtime exceptions to prevent?
  Invalid information flow.
- What operations are legal?
  send() only on encrypted data.
- What properties of data should hold?
  Separate encrypted from plaintext strings.

Brainstorming new type checkers

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

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

Brainstorming

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

Outline

- Type qualifiers
- Pluggable type checkers
- **Writing your own checker**
- Verification vs. bug finding
- 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

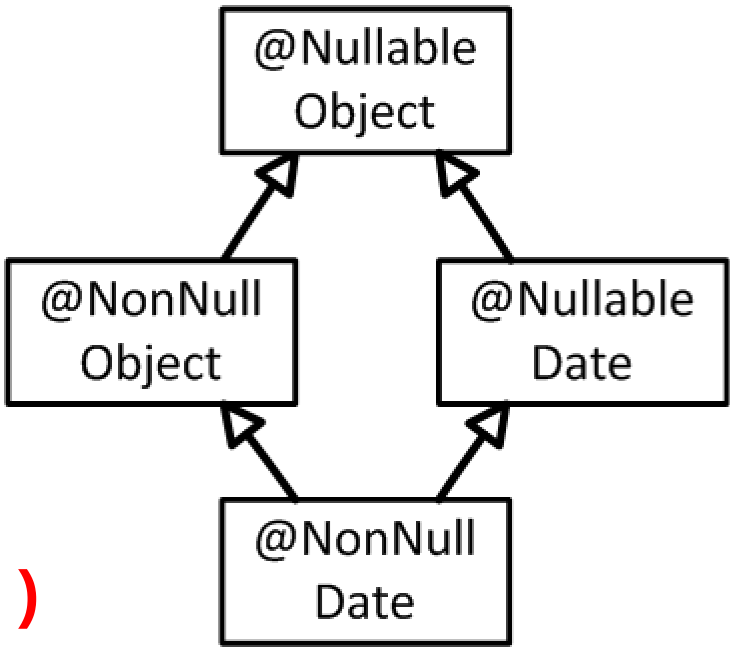
```
@TypeQualifier
public @interface NonNull { }
```

Defining a type system

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

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

What assignments are legal:



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
- **Verification vs. bug finding**
- Conclusion

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.
The approaches are converging.

Other design considerations

- Visibility of specifications and warning suppressions
  - In the source code
    - documentation aids programer understanding
  - In the tool
    - reduces code clutter
- Analysis comprehensibility
  - A transparent tool gives understandable outcomes
    - requires more upfront effort; more false positives
  - An opaque tool can use more powerful analyses
    - requires more effort to understand warnings

Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- Verification vs. bug finding
- **Hands-on practice**
- Conclusion

How to get started

1. **Write the specification**
   Search the Javadoc for occurrences of “null”
   Replace the wordy English text by **`@Nullable`**
   Can also search code, but no annos in methods
2. **Run Nullness Checker**: verify/improve spec
   For each warning:
   – Reason about whether the code is safe
   – Express that reasoning as annotations
   – Consider improving the code’s design

Tips

What to type-check:

- Only type-check properties that matter to you
  - Use subclasses (not type qualifiers) if possible
- Choose part of your code to type-check first
  - Eliminate raw types such as **`List`**; use **`List<String>`**

While you are doing type-checking:

- Write the spec first (and think of it as a spec)
- Avoid warning suppressions when possible

Your turn to improve your code!

1. Choose a project you care about
2. Improve it
   - Apply an existing checker to your code, or
   - Create a new domain-specific type checker

Or, try the tutorial:

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

Outline

- Type qualifiers
- Pluggable type checkers
- Writing your own checker
- Verification vs. bug finding
- **Conclusion**

Pluggable type-checking

- Java 8 syntax for type annotations
- 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://CheckerFramework.org/
  (or, web search for "Checker Framework")