Null Pointer Exceptions

Problems, Current Approaches, and Ongoing Efforts

`https://eisop.github.io/`

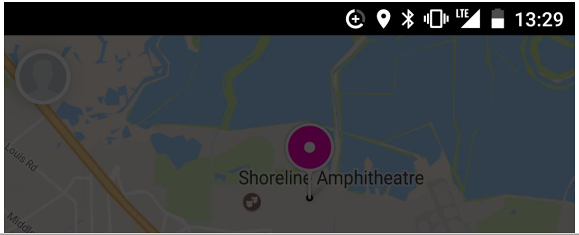
Live demo: `http://eisop.uwaterloo.ca/live/`

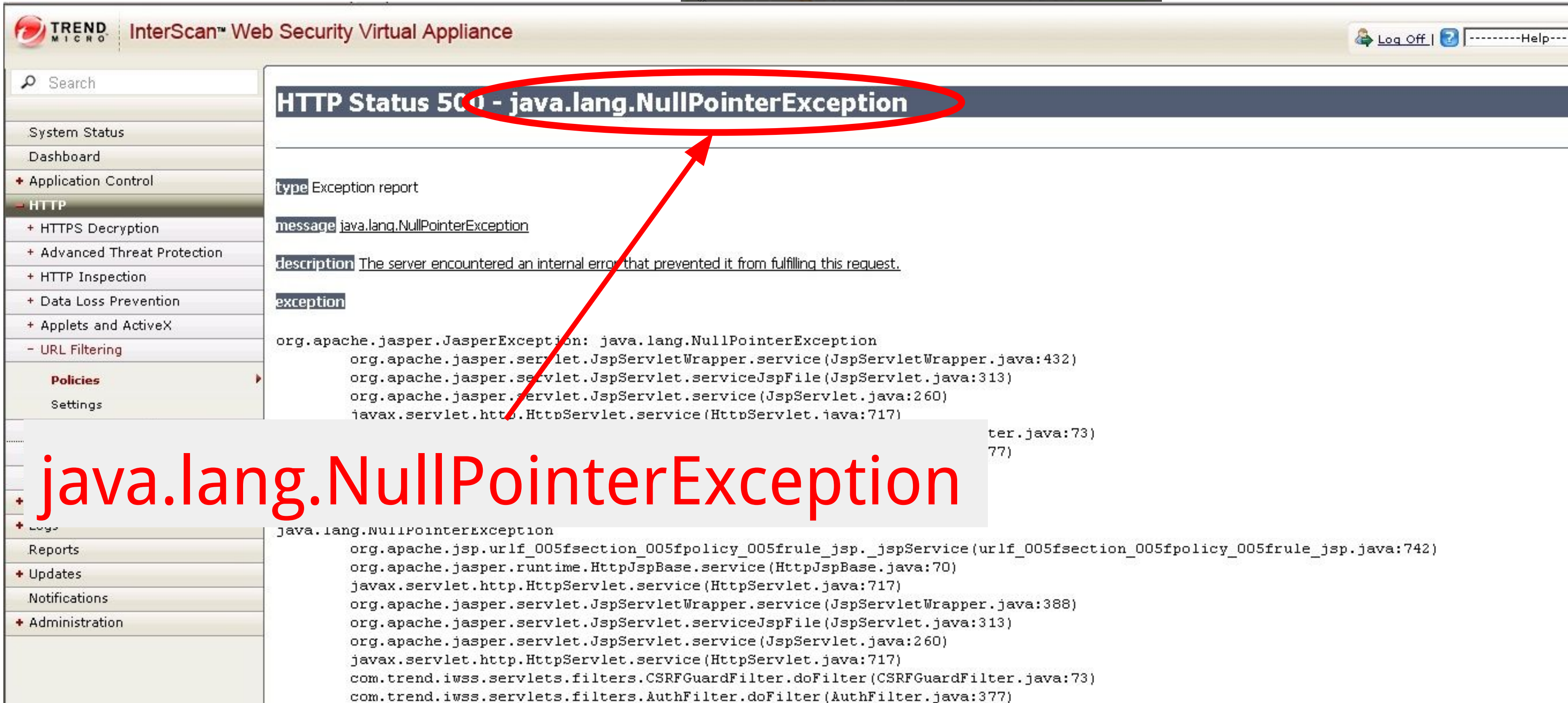
Werner Dietl, University of Waterloo

`https://ece.uwaterloo.ca/~wdietl/`

UNIVERSITY OF WATERLOO

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 Nullness Checker
- How to use advanced features
- Alternatives and ongoing efforts

This talk focuses on nullness.

A talk yesterday focussed on writing your own type system.

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();
```

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();
```

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();
```

[Nullness Checker demo](#) [Fixed](#)

Prevent null pointer exceptions

Goal: the program only dereferences 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!

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);
```

Solution: Pluggable Type Checking

1. Design a type system to solve a specific problem
2. Write type qualifiers in code (or, use type inference)

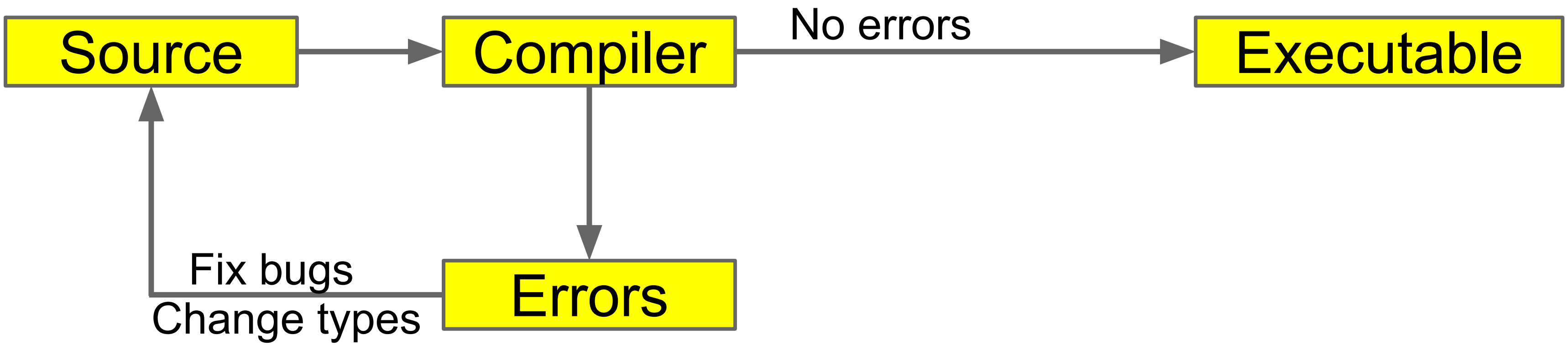
```
void foo (@Nullable Date date) {
  date.setSeconds(0); // compile-time error
```

3. Type checker warns about violations (bugs)

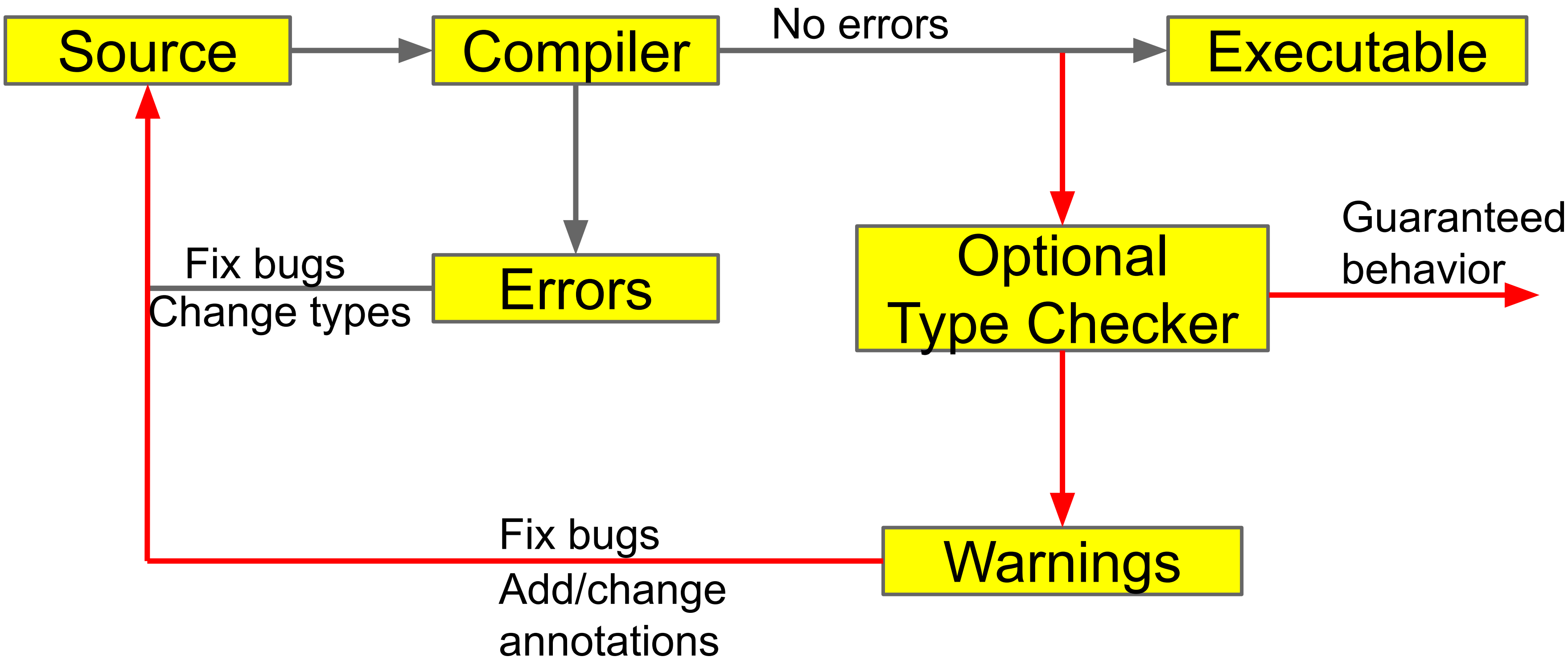
```
% javac -processor NullnessChecker MyFile.java

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

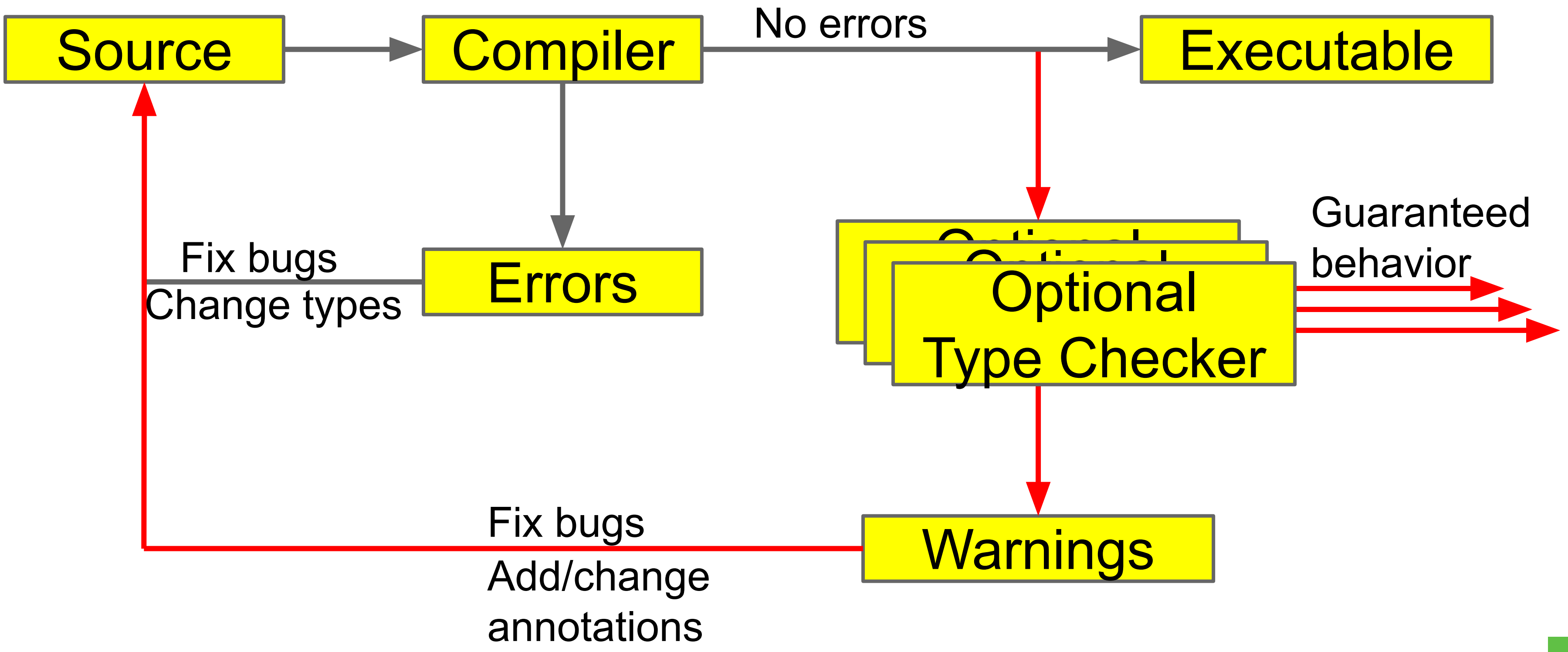
Type Checking



Optional Type Checking



Optional Type Checking



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 …
```

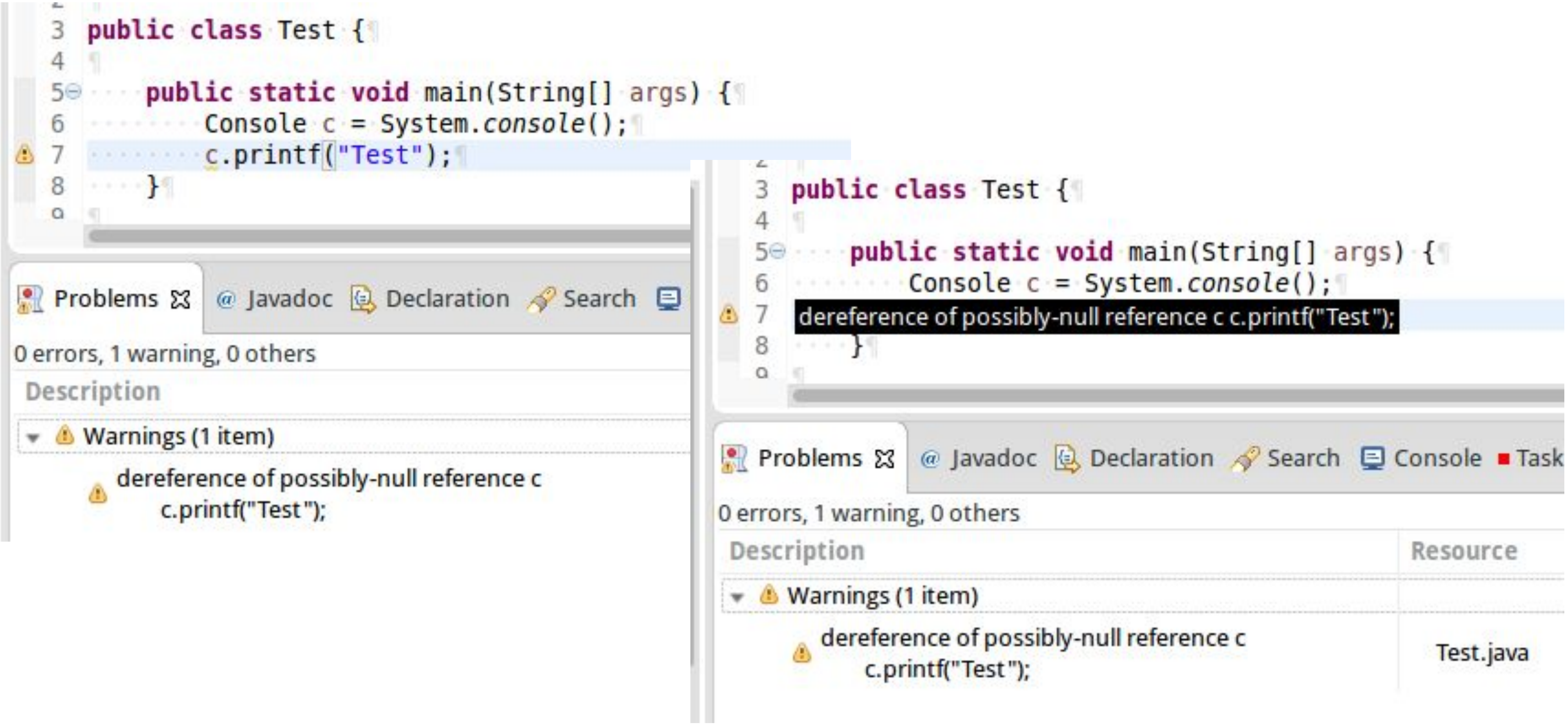
Standard error format allows tool integration

Ant, Maven, Gradle integration

```
<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>io.github.eisop</groupId>
      <artifactId>checker-qual</artifactId>
      <version>3.34.0-eisop1</version>
    </dependency>
</dependencies>
```

Eclipse, IntelliJ, NetBeans integration



Live demo: http://eisop.uwaterloo.ca/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 ▼

Check

Examples:

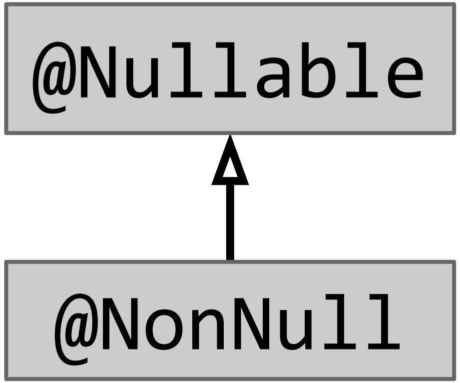
Nullness: NullnessExample | NullnessExampleWithWarnings

MapKey: MapKeyExampleWithWarnings

Interning: InterningExample | InterningExampleWithWarnings

Lock: GuardedByExampleWithWarnings | HoldingExampleWithWarnings | EnsuresLockHeldExample | Locl

Preventing null-pointer exceptions



Basic type system:

`@Nullable`	might be null
`@NonNull`	definitely not null

Default is `@NonNull` (opposite of Java's default)

- Requires fewer annotations
- Makes the dangerous case explicit

(Nearly) no annotations in method bodies!

Type rules

To prevent <u>null pointer exceptions</u>:

- `expr.field`
  `expr.getValue()`
  receiver must be non-null
- `synchronized (expr) { … }`
  monitor must be non-null
- …

Flow-sensitive type refinement

After an operation, give an expression a more specific type

```
@Nullable Object x;
if (x != null) {
  ...
}
...
```

x is @NonNull here

x is @Nullable again

```
@Nullable Object y;
y = new SomeType();
...
y = unknownValue;
...
```

y is @NonNull here

y is @Nullable again

Flow-sensitive type refinement

```
if (myField != null) {

  myField.hashCode();
}
```

No need to declare a new local variable

One check for `null` is not enough

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

3 ways to express persistence across side effects:

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

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

```
/** Interns a String, and handles null. */
@PolyNull String intern(@PolyNull String a) {
  if (a == null) {
    return null;
  }
  return a.intern();
}
```

Like defining two methods:

```
    @NonNull  String intern(@NonNull  String a) {…}
    @Nullable String intern(@Nullable String a) {…}
```

A non-null field might contain `null`

```
@NonNull String name;
MyClass() {  // constructor
  … this.name.hashCode() …
}
```

Initialization

- `@Initialized` (constructor has completed)
- `@UnderInitialization(Frame.class)`
  - Its constructor is currently executing
- `@UnknownInitialization(Frame.class)`
  - Might be initialized or under initialization

Map keys and `Map.get`

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

`Map.get` can return null!   … unless

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

`@KeyFor`   [rarely written, usually inferred]

Map key example

```
/** Computes predominators for each node in the graph. */
<T> Map<T, List<T>>
dominators(Map<T, List<@KeyFor("#1") T>> predecessors) {
  ...
  for (T node : predecessors.keySet()) {
    for (T pred : predecessors.get(node)) {   // no NPE
      ... predecessors.get(pred) ...          // no NPE
```

Suppressing warnings

Because of Checker Framework false positives

`@SuppressWarnings(“nullness”)`

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

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

More: https://eisop.github.io/cf/manual/#suppressing-warnings

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)

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

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

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

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

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

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

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

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

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

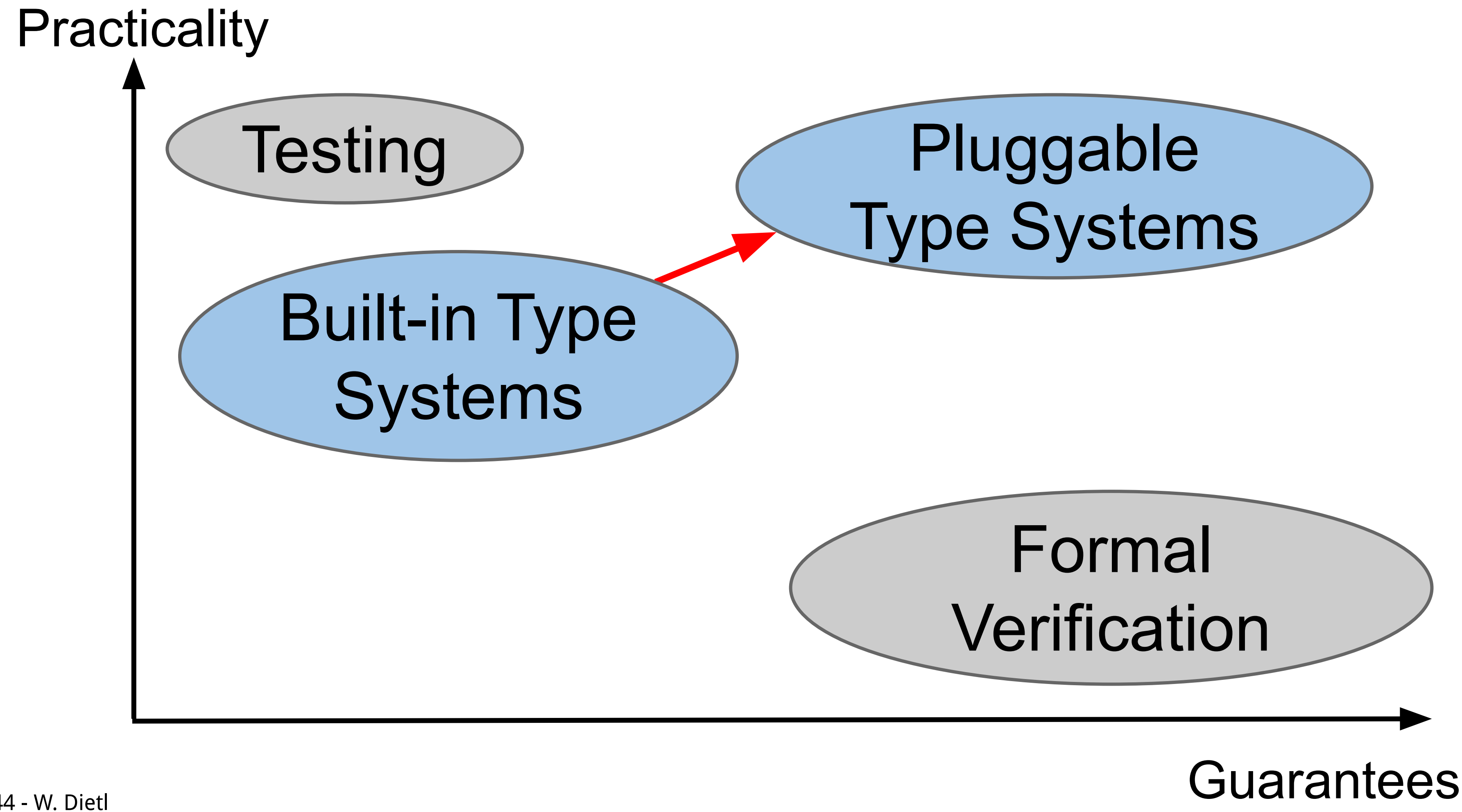
Verification

- **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.


Practicality
Testing
Pluggable Type Systems
Built-in Type Systems
Formal Verification
Guarantees

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.

NullAway from Uber

https://github.com/uber/nullaway

- Fast
- Simplified generics - always non-null
- No contracts, but library models
- Uses the CF Dataflow Framework

Eradicate by Facebook/Meta

https://fbinfer.com/docs/checker-eradicate/

https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/

Kotlin Nullness

https://kotlinlang.org/docs/null-safety.html

- Nullness as part of the language
- Nicer syntax
- Contract annotations
- Platform types
- No Initialization Checker

JSpecify

JSpecify

Standard Java Annotations for Static Analysis

JSpecify effort

http://jspecify.org/

- Collaboration between Google, JetBrains, Meta, SpotBugs, Uber, and many others
- Finally replace JSR 305
- Clear semantics and inter-op
- v0.3.0 released

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>`

Pluggable type-checking improves code

Checker Framework for optional type checkers

- Featureful, effective, easy to use, scalable

Prevent null pointers at compile time

Improve your code!

`https://eisop.github.io/`

Acknowledgement: Slides developed with Michael Ernst, University of Washington.