Using Type Annotations to Improve Code Quality

Birds-of-a-Feather Session

Werner Dietl, University of Waterloo
Geertjan Wielenga, NetBeans
Konstantin Bulenkov, JetBrains

Since Java 5: annotations

Only for declaration locations:

```
@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 read-only array of non-empty arrays 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 read-only array of non-empty arrays of English strings:

```
@English String @ReadOnly [] @NonEmpty [] a;
```

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, …

Array annotations

A read-only array of non-empty arrays of English strings:

```
@English String @ReadOnly [] @NonEmpty [] a;
```

Explicit method receivers

```
class MyClass {
  int foo(@TParam String p) {…}
  int foo(@TRecv MyClass this,
          @TParam String p) {…}
```

No impact on method binding and overloading

Constructor return & receiver types

Every constructor has a return type

```
class MyClass {
  @TReturn MyClass(@TParam String p) {...}
```

Inner class constructors also have a receiver

```
class Outer {
  class Inner {
    @TReturn Inner(@TRecv Outer Outer.this,
                   @TParam String p) {...}
```

The Checker Framework: Preventing Errors Before They Happen

http://CheckerFramework.org/

Werner Dietl
University of Waterloo
https://ece.uwaterloo.ca/~wdietl/

UNIVERSITY OF WATERLOO

Joint work with Michael D. Ernst and many others.

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

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

Collections.emptyList().add(“one”);

dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking c ors

NullPointerException

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

Collections.emptyList().add(“one”);

dbStatement.executeQuery(userData);
```

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

```
System
Collections.emptyList().add(“one”);
dbStatement.executeQuery(userData);
```

UnsupportedOperationException

Java's type system is too weak

Type checking prevents many errors

```
int i = “hello”; // error
```

Type checking doesn't prevent enough errors

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

Collections

dbStatement.executeQuery(userData);
```

SQL Injection Attacks!

Null pointer exception

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

Where is the error?

Null pointer exception

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

Where is the error?

Can’t decide without specification!

Solution 1: Restrict use

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

Solution 2: Restrict implementation

```
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
  - Reduce number of run-time checks

Possible negatives:

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

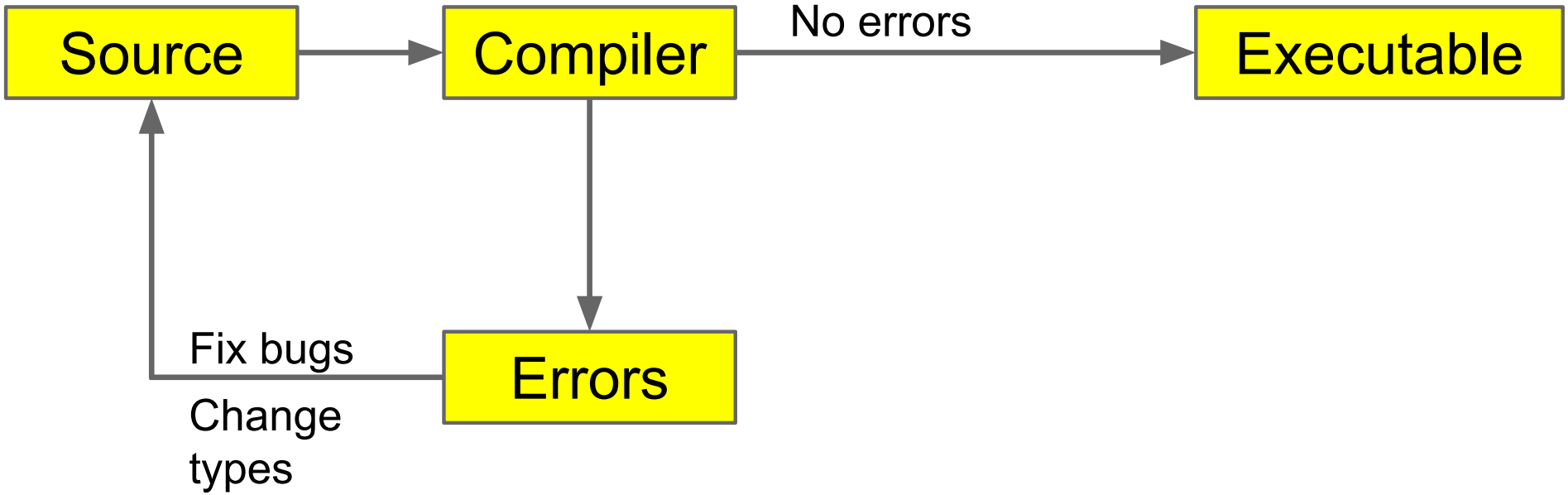
The Checker Framework

A framework for pluggable type checkers
"Plugs" into the OpenJDK compiler

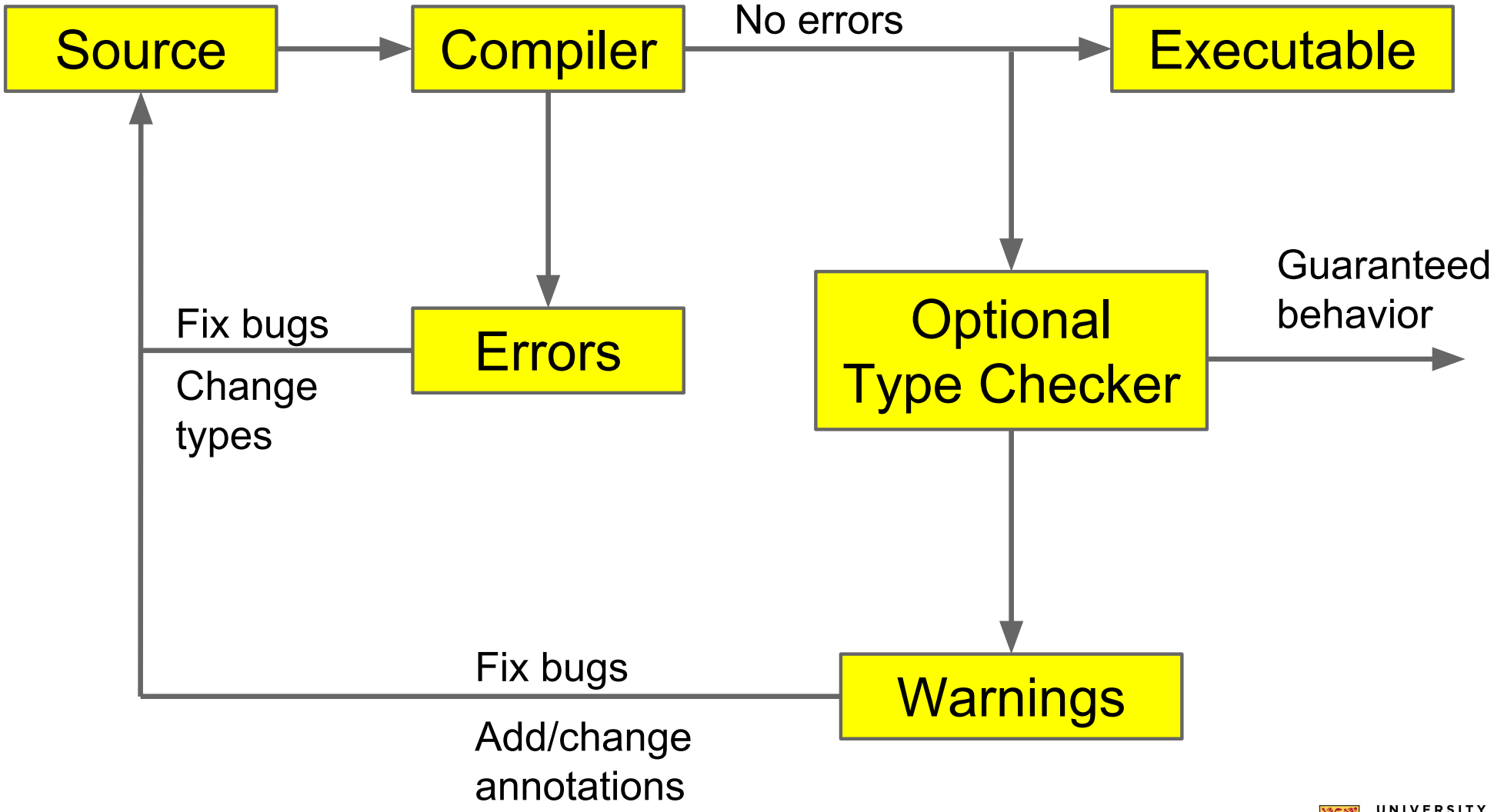
```
javac -processor MyChecker ...
```

Eclipse plug-in, Ant and Maven integration

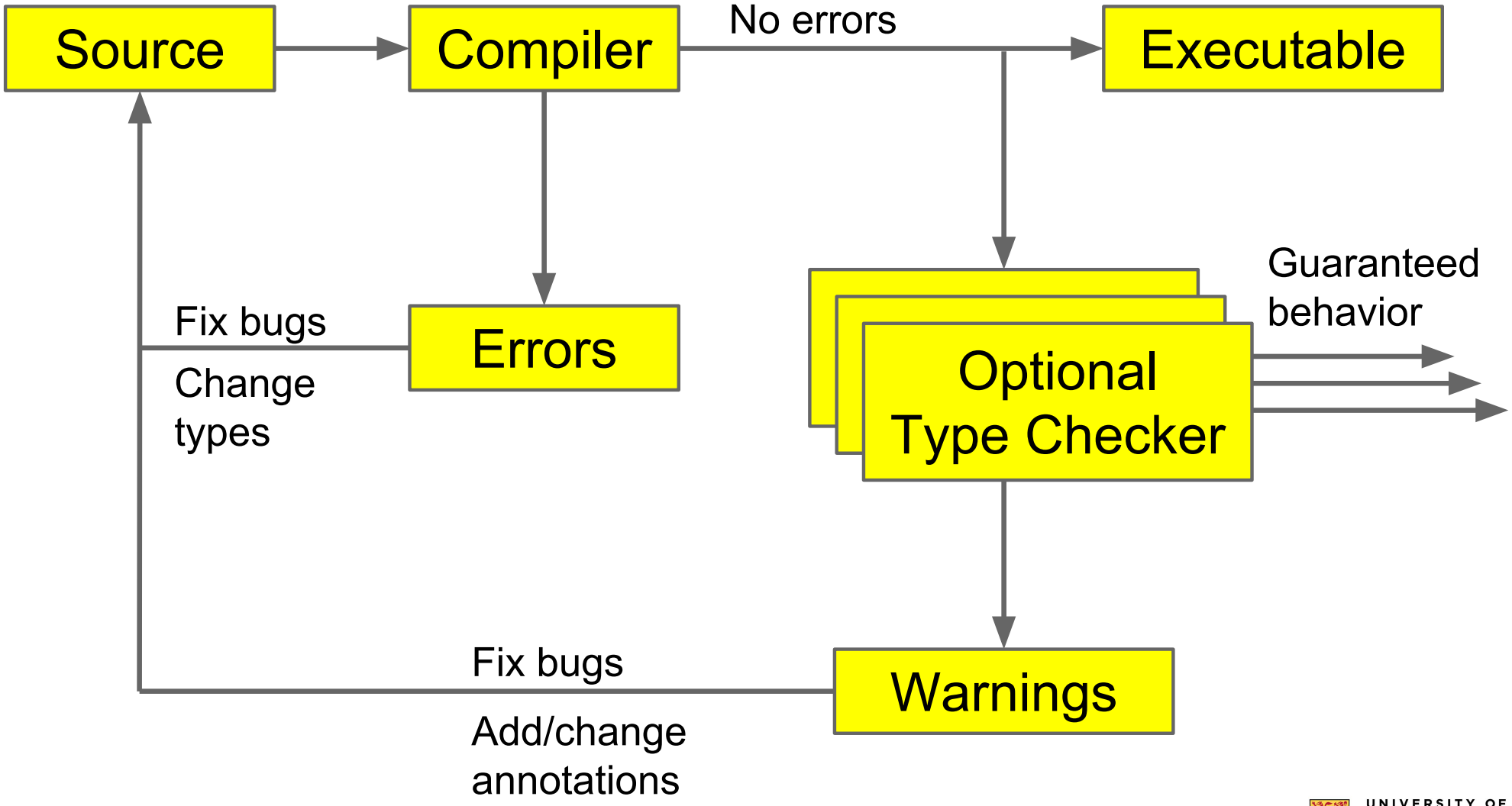
Type Checking



Optional Type Checking



Optional Type Checking



Checker Framework experience

Type checkers reveal important latent defects

Ran on >6 million LOC of real-world open-source code

Found hundreds of user-visible failures

Improved design and documentation

Annotation overhead is low

Mean 2.6 annotations per kLOC [Dietl et al. ICSE'11]

Conclusions

Java 8 syntax for type annotations

Checker Framework for creating type checkers

- Featureful, effective, easy to use, scalable

Prevent bugs at compile time

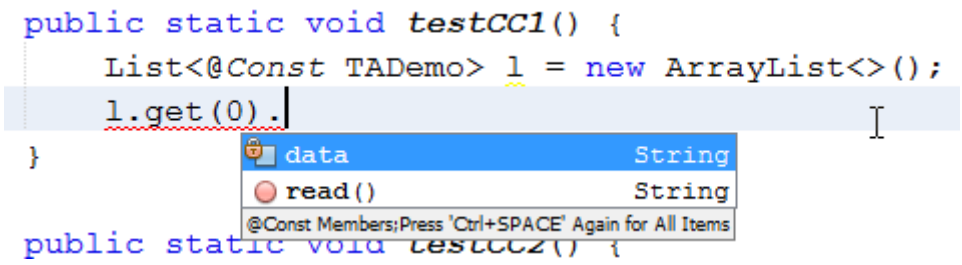
Create custom type-checkers

Learn more, or download at:

`http://CheckerFramework.org/`

Code Analysis Using Type Annotations

- Code Completion
- Find Usages
- Smart Code Completion



Java™ ORACLE

Code Analysis Using Type Annotations

- Code completion in an IDE could understand the meaning of some of the type annotations.
  - Consider there would be annotation "@Const" similar to "const" in C++

```
List<@Const MutableObject> l = ...;
l.get(0).|
```

  - Code completion could only show methods marked as "@Const" at this place, over methods not marked with @Const.
  - On the code analysis front, a hint/warning could appear if a non-@Const method would be used.

Find Usages Using Type Annotations

- Find usages could also understand some of the annotations.
  - e.g. @Const: there could be a filter allowing to split "mutations" of an object from read-only access, similar to current read/write filters for variables.
  - There could be two new filters in the Find Usages window, for non-mutators and mutators.
  - The outcome of unselecting "non-mutators" should be just the invocation of a "write" method.
  - Note that both non-mutators and mutators are also read-only accesses, so read-only access must be selected.

Smart Code Completion Using Type Annotations

- The "smart types" code completion could understand type annotations.

```
List<@NonNull String> a1 = …
List<@Nullable String> a2 = …

List<@NonNull String> result =

/* Code could show only a1 in smart code completion here, */
/* as a2's type won't match */
```