

Using Type Annotations to Improve Your Code

The background of the slide features two grey birds with red beaks perched on a dark branch. The birds are facing each other, and the background is a soft, light greenish-yellow gradient.

Birds-of-a-Feather Session

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

Open for questions

Survey:

Did you attend the developer session yesterday?

Did you attend the tutorial this morning?

Which of these best describes you?

- Specific question / concern / feedback
- Specific problem / use case / tool
- Curious, want to learn more

Please raise questions / issues



Verification using annotations

Annotations are a specification

They enable verification

- Example: Checker Framework

Is verification on your radar?

What are you looking for in a verification tool?

What prevents you from using a verification tool?



Schedule

Java syntax for type annotations

Pluggable types: a use of type annotations

Questions and discussion



Since Java 5: declaration annotations

Only for **declaration** locations:

@Deprecated

class

```
class Foo {
```

field

```
    @Getter @Setter private String query;
```

```
    @SuppressWarnings("unchecked")
```

```
    void foo() { ... }
```

method

```
}
```



Since Java 8: type annotations

Annotations on all occurrences of types:

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



How Java treats type annotations

Does not affect bytecode, stored in classfile

Handled by javac, javap, javadoc, ...

Writing type annotations has **no effect** unless you run an annotation processor



Write annotations before the element

Write declaration annotations before the decl.

Write type annotations before the type

```
@Override
```

```
public @NonNull String toString() {...}
```

Don't split them up:

```
@NonNull
```

```
public String toString() {...}
```



Array annotations

```
String [] [] a;
```

An **array** of **arrays** of **strings**



Array annotations

String

[]

[] a;

An

array of
arrays of
strings



Array annotations

String

[]

[] a;

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



Array annotations

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

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

Rule: write the annotation before the type



Explicit method receivers

```
class MyClass {  
    public String toString() {}  
    public boolean equals(Object other) {}  
  
}
```



Explicit method receivers

```
class MyClass {  
    public String toString() {}  
    public boolean equals(Object other) {}  
  
}
```

```
myval.toString();  
myval.equals(otherVal);
```



Explicit method receivers

```
class MyClass {  
    public String toString(MyClass this) {}  
    public boolean equals(MyClass this,  
                          Object other) {}  
}
```

```
myval.toString();  
myval.equals(otherVal);
```

No impact on method
binding and overloading



Explicit method receivers

```
class MyClass {  
    public String toString(@ReadOnly MyClass this) {}  
    public boolean equals(@ReadOnly MyClass this,  
                          @ReadOnly Object other) {}  
}
```

```
myval.toString();  
myval.equals(otherVal);
```

Rationale: need a syntactic location for type annotations



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) {...}
```



Why were type annotations added to Java?



Annotations are a specification

- More concise than English text or Javadoc
- Machine-readable
- Machine-checkable

- Improved documentation
- Improved correctness



Pluggable Type Systems

- Use Type Annotations to express properties
- Prevent errors at compile time

 <http://CheckerFramework.org/>

Twitter: @CheckerFrmwrk

Facebook/Google+: CheckerFramework



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

```
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");
```



Solution: Pluggable Type Checking

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

```
@Immutable Date date = new Date();  
date.setSeconds(0); // compile-time error
```

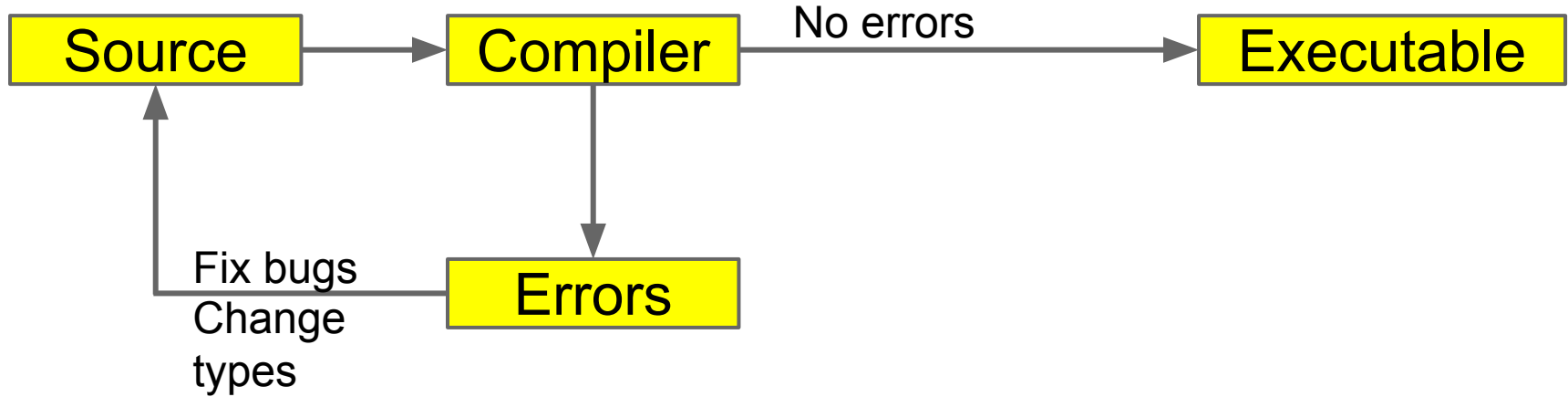
3. Type checker warns about violations (bugs)

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

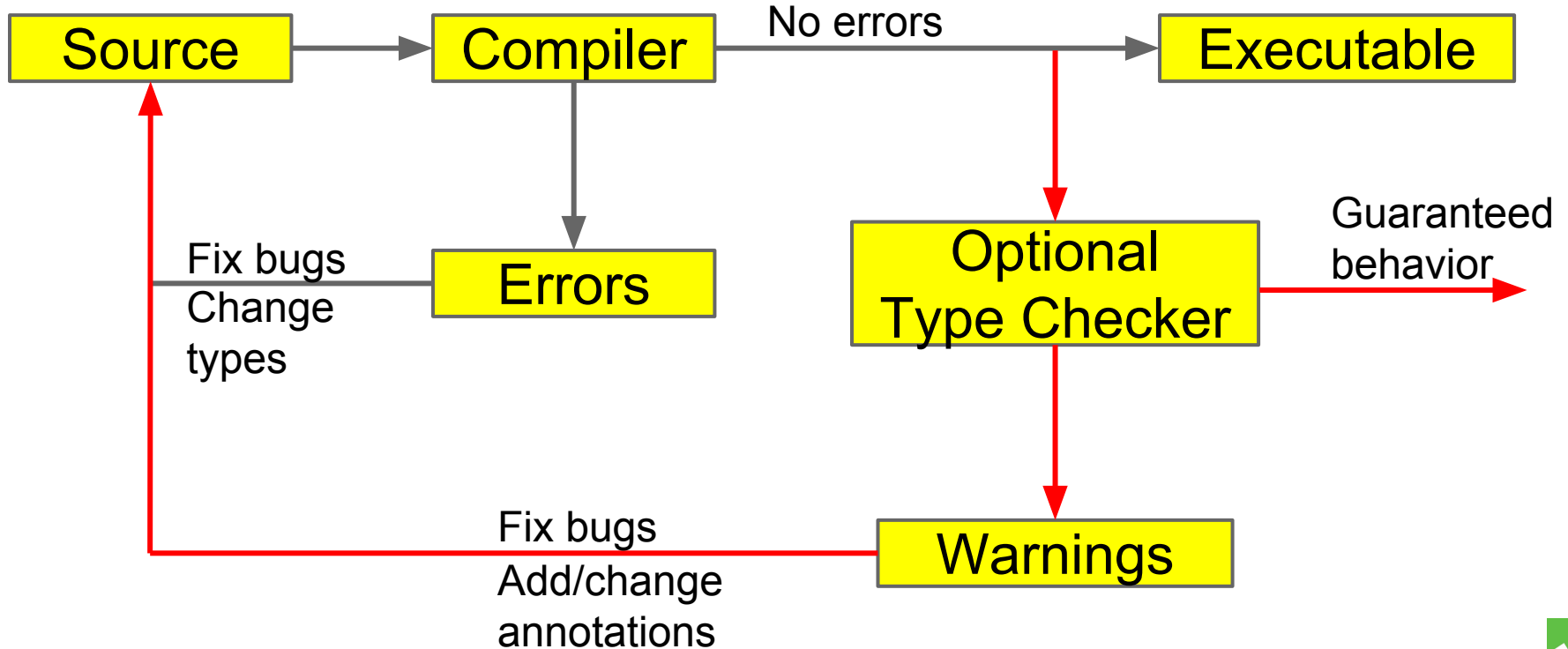
```
MyFile.java:149: dereference of possibly-null reference bb2  
    allVars = bb2.vars;  
                ^
```



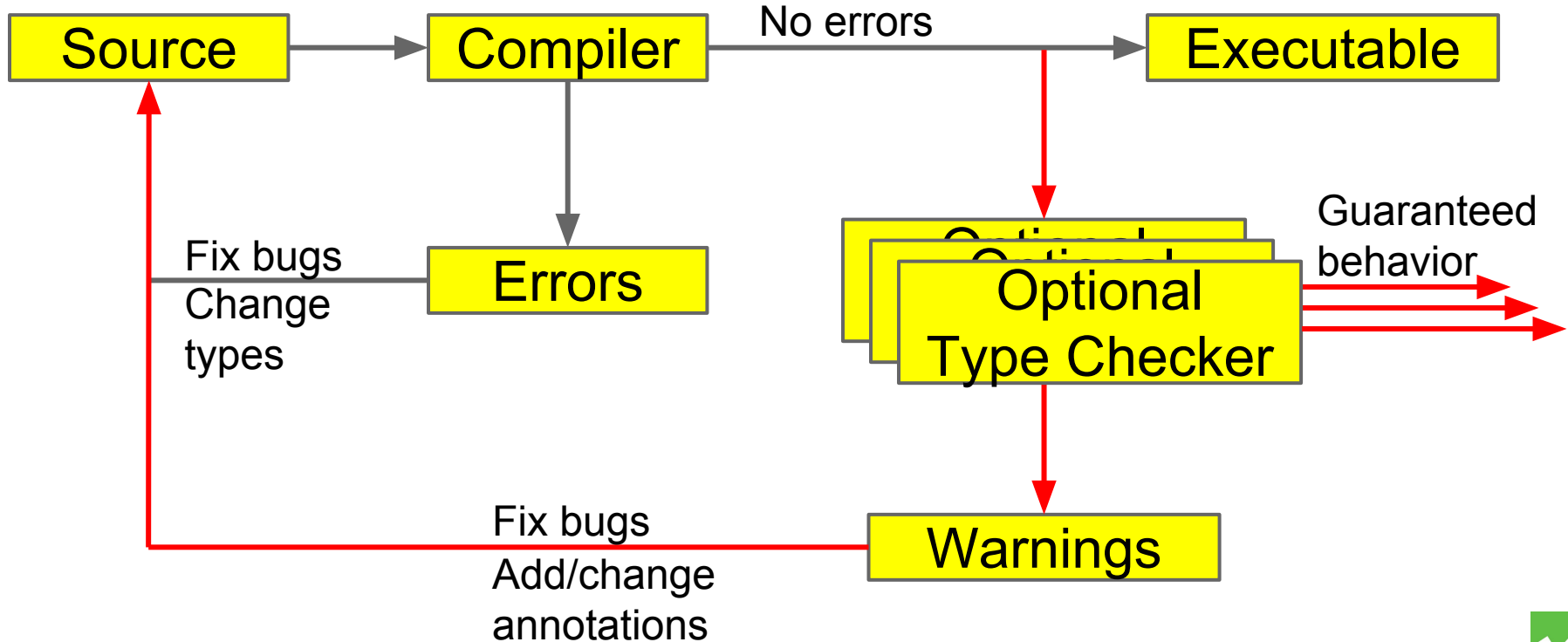
Type Checking



Optional Type Checking



Optional Type Checking



Example type systems

Null dereferences (`@NonNull`)

Equality tests (`@Interned`)

Concurrency / locking (`@GuardedBy`)

Command injection vulnerabilities (`@OsTrusted`)

Privacy (`@Source`)

Regular expression syntax (`@Regex`)

printf format strings (`@Format`)

Signature format (`@FullyQualified`)

Compiler messages (`@CompilerMessageKey`)

Fake enumerations (`@Fenum`)

You can write your own checker!



Annotating external libraries

When type-checking clients, need library spec.

```
class System {  
    Console console() { ... }  
}
```

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



Annotating external libraries

When type-checking clients, need library spec.

```
class System {  
    @Nullable Console console() { ... }  
}
```

Compile-time warning

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



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



Static type system

Plug-in to the compiler

Doesn't impact:

- method binding
- memory consumption
- execution

A future tool might affect run-time behavior



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



Problem: annotation effort

Programmer must write type annotations

- on program code
- on libraries

Very few: 1 per 100 lines, often much less

- depends on the type system

Solution: type inference



Type inference within a method

- Called “flow-sensitive refinement”
- A variable can have different types on different lines of code
- Low overhead
- Always used

```
x.toString(); // warning: possible NPE
if (x!=null) {
    x.toString(); // no warning
}
x.toString(); // warning: possible NPE
```

Does not affect type signatures



Whole-program type inference

- Analyze **all** the code at once
- Determine the globally optimal annotations

Approach:

- Introduce placeholder for each location
- Use the same type rules to generate constraints
- Use a solver to find a solution

Available (beta) with the Checker Framework



Practicality

Testing

Built-in Type
Systems

Pluggable
Type Systems

Formal
Verification

Guarantees



Checker Framework Community

Open source project:

<https://github.com/typetools/checker-framework>

- Monthly release cycle
- >13,800 commits, 75 authors
- Welcoming & responsive community



Checker Framework Plans

More type systems:

- Signed vs. unsigned numbers
- Immutability
- Determinism

Type inference

Combined static & dynamic enforcement



Present and Future of Type Annotations

Java 9

Type annotation implementation improvements

Java >11

What annotation-related language feature are you missing?

- Statement annotations
- More expressive annotation attributes
- Annotation inheritance



Verification using annotations

Annotations are a specification

They enable verification

- Example: Checker Framework

Is verification on your radar?

What are you looking for in a verification tool?

What is keeping you from using a verification tool?



Conclusions

Type Annotations in Java

Checker Framework for creating type checkers

- Featureful, effective, easy to use, scalable

Prevent bugs at compile time

Improve your code!

<http://CheckerFramework.org/>

