Implement your own Type System, today!

The Checker Framework

`http://CheckerFramework.org/`

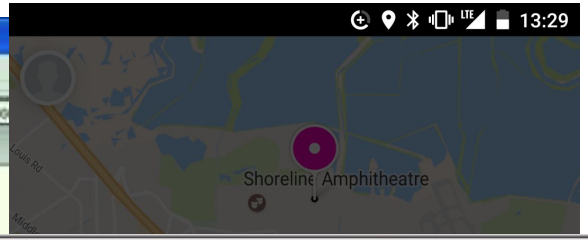
Twitter: @CheckerFrmwrk

Live demo: `http://CheckerFramework.org/live/`

Werner Dietl, University of Waterloo

UNIVERSITY OF WATERLOO

Motivation



TREND MICRO InterScan™ Web Security Virtual Appliance

Log Off | --------Help---

Search

- System Status
- Dashboard
- + Application Control
- − HTTP
  - + HTTPS Decryption
  - + Advanced Threat Protection
  - + HTTP Inspection
  - + Data Loss Prevention
  - + Applets and ActiveX
  - − URL Filtering
    - Policies
    - Settings
- Reports
- + Updates
- Notifications
- + Administration

HTTP Status 500 - java.lang.NullPointerException

type Exception report

message java.lang.NullPointerException

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
org.apache.jasper.JasperException: java.lang.NullPointerException
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:432)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:313)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:260)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
```

ter.java:73)
77)

java.lang.NullPointerException

java.lang.NullPointerException

```
        org.apache.jsp.urlf_005fsection_005fpolicy_005frule_jsp._jspService(urlf_005fsection_005fpolicy_005frule_jsp.java:742)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:388)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:313)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:260)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
        com.trend.iwss.servlets.filters.CSRFGuardFilter.doFilter(CSRFGuardFilter.java:73)
        com.trend.iwss.servlets.filters.AuthFilter.doFilter(AuthFilter.java:377)
```

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();
```


Prevent null pointer exceptions

Java 8 introduces the `Optional<T>` type

- Wrapper; content may be *present* or *absent*
- Constructor: `of(T value)`
- Methods: `boolean isPresent()`, `T get()`

```
Optional<String> maidenName;
```


`Optional` reminds you to check

Without `Optional`:

```
String mName;
mName.equals(...);

if (mName != null) {
    mName.equals(...);
}
```

possible NullPointerException

With `Optional`:

```
Optional<String> omName;
omName.get().equals(...);

if (omName.isPresent()) {
    omName.get().equals(...);
}
```

possible NoSuchElementException

possible NullPointerException

Complex rules for using `Optional` correctly!

How not to use Optional

Other gu
Stephen
Dalorzo,
Brian Go
Olszewsl
Oleg She

Stuart Marks's rules:

1. Never, ever, use null for an Optional variable or return value.
2. Never use Optional.get() unless you can prove that the Optional is present.
3. Prefer alternative APIs over Optional.isPresent() and Optional.get().
4. It's generally nal for the specific purpose of chaining met
5. If an Optiona chain, or has an intermediate result of Opt ex.
6. Avoid using Optional in fields, method parameters, and collections.
7. Don't use an Optional to wrap any collection type (List, Set, Map). Instead, use an empty collection to represent the absence of values.

Let's enforce the rules with a tool.

Which rules to enforce with a tool

Stuart Marks's rules:

1. **Never**, ever, use null for an Optional variable or return value.
2. **Never** use Optional.get() unless you can prove that the Optional is present.
3. *Prefer* alternative APIs over Optional.isPresent() and Optional.get().
4. It's *generally a bad idea* to create an Optional for the specific purpose of chaining methods from it to get a value.
5. If an Optional chain has a nested Optional chain, or has an intermediate result of Optional, it's *probably too complex*.
6. *Avoid* using Optional in fields, method parameters, and collections.
7. **Don't** use an Optional to wrap any collection type (List, Set, Map). Instead, use an empty collection to represent the absence of values.

Which rules to enforce with a tool

Stuart Marks's rules:

1. **Never**, ever, use null for an Optional variable or return value.
2. **Never** use Optional.get() unless you can prove that the Optional is present.
3. *Prefer* alternative APIs over Optional.isPresent() and Optional.get().
4. It's *generally* … specific purpose of chaining met…
5. If an Optiona… as an intermediate result of Opt…
6. *Avoid* using Optional in fields, method parameters, and collections.
7. **Don't** use an Optional to wrap any collection type (List, Set, Map). Instead, use an empty collection to represent the absence of values.

These are *type system* properties.

Define a type system

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

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

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

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

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

$$
\text{GT-Upd}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{N}_0 \qquad \mathtt{N}_0 = \mathtt{u}_0\ \mathtt{C}_0\langle\_\rangle \\ \mathtt{T}_1 = fType(\mathtt{C}_0, \mathtt{f}) \\ \Gamma \vdash \mathtt{e}_2 : \mathtt{N}_0 \triangleright \mathtt{T}_1 \\ \mathtt{u}_0 \neq \mathtt{any} \qquad rp(\mathtt{u}_0, \mathtt{T}_1)\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e}_2 : \mathtt{N}_0 \triangleright \mathtt{T}_1}
$$

$$
\text{DYN}\ \frac{\begin{array}{l}\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ \mathtt{h} \vdash \iota_1 : dyn({}^{\mathtt{s}}\mathtt{N}, \mathtt{h}, {}^{\mathtt{r}}\iota_1) \\ \mathtt{h} \vdash \iota_2 : dyn({}^{\mathtt{s}}\mathtt{T}, \iota_1, \mathtt{h}(\iota_1)\downarrow_1) \\ {}^{\mathtt{s}}\mathtt{N} = \mathtt{u}_N\ \mathtt{C}_N\langle\_\rangle \\ \mathtt{u}_N = \mathtt{this}_u \Rightarrow {}^{\mathtt{r}}\Gamma(\mathtt{this}) \\ free({}^{\mathtt{s}}\mathtt{T}) \subseteq dom(\mathtt{C}_N) \\ \left.\begin{array}{l}\\ \\ \\ \\ \\ \end{array}\right\} \Longrightarrow \mathtt{h} \vdash \iota_2 : dyn({}^{\mathtt{s}}\mathtt{N} \triangleright {}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \\ {}^{\mathtt{r}}\mathtt{T} = \iota'\ \_\langle\_\rangle \qquad \iota \vdash {}^{\mathtt{r}}\mathtt{T}\ {}^{\mathtt{r}}{<:}\ \iota'\ \mathtt{C}\langle\overline{{}^{\mathtt{r}}\mathtt{T}}\rangle \qquad \iota \vdash {}^{\mathtt{r}}\mathtt{T}\ {}^{\mathtt{r}}{<:}\ \iota'\ \mathtt{C}\langle\overline{{}^{\mathtt{r}}\mathtt{T}_a}\rangle \Rightarrow \iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ {}^{\mathtt{r}}{<:}\ \overline{{}^{\mathtt{r}}\mathtt{T}_a} \\ \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad\qquad \mathrm{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}\end{array}}{dyn({}^{\mathtt{s}}\mathtt{T}, \iota, {}^{\mathtt{r}}\mathtt{T}, (\overline{\mathtt{X}'\ {}^{\mathtt{r}}\mathtt{T}'}; \_)) = {}^{\mathtt{s}}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{\mathtt{r}}\mathtt{T}/\mathtt{X}}, \overline{{}^{\mathtt{r}}\mathtt{T}'/\mathtt{X}'}]}
$$

Define a type system

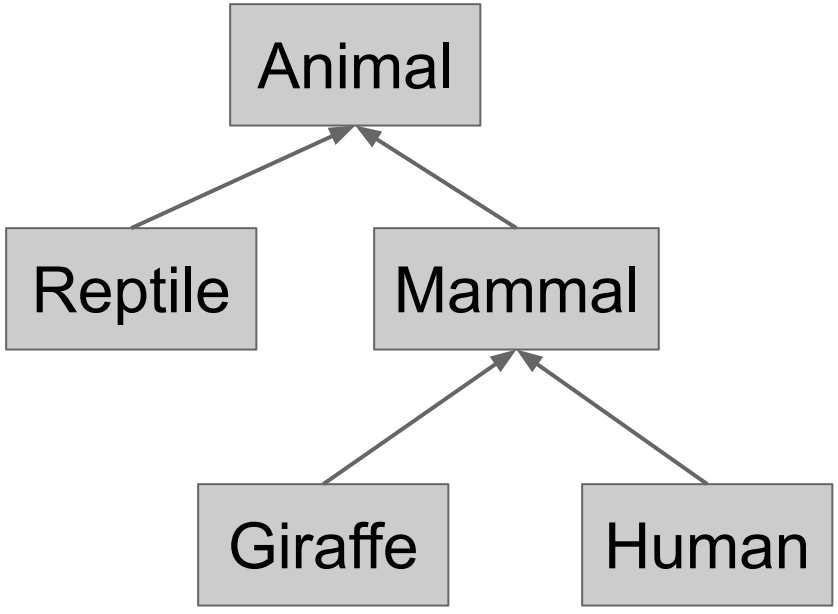
1. **Type hierarchy** (subtyping)
2. **Type rules** (what operations are illegal)
3. **Type introduction** (what types for literals, …)
4. **Dataflow** (run-time tests)

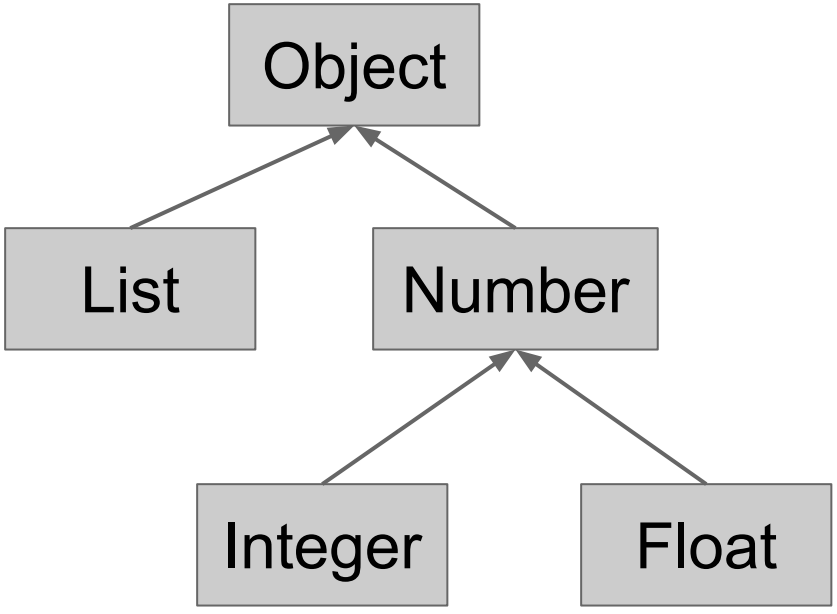
We will define two type systems:
Nullness and Optional

Define a type system

1. **Type hierarchy (subtyping)**
2. Type rules (what operations are illegal)
3. Type introduction (what types for literals, …)
4. Dataflow (run-time tests)

1. Type hierarchy

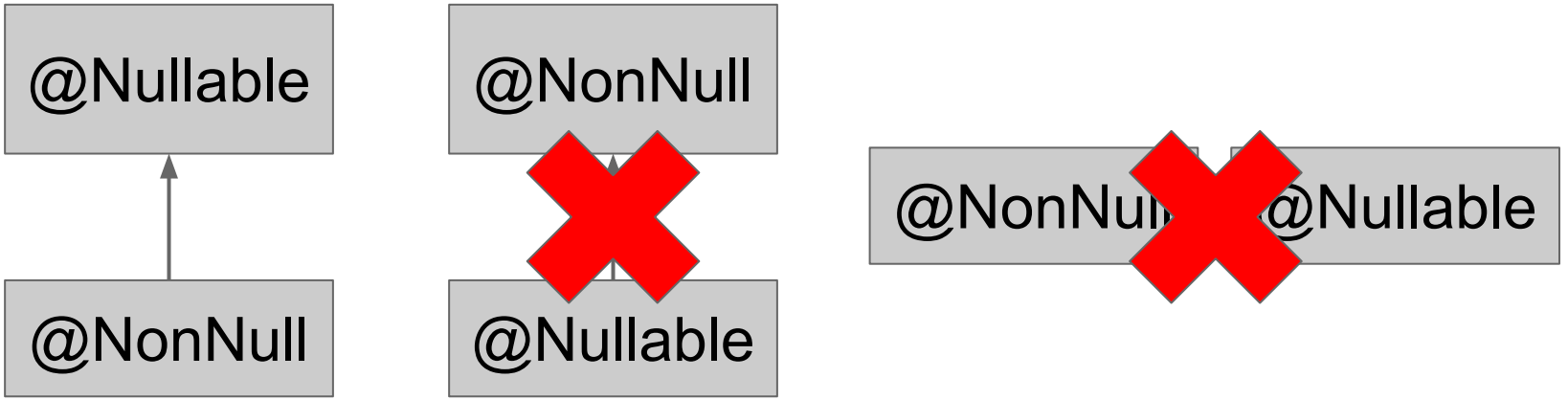




2 pieces of information:

- the types
- their relationships (lower = fewer values, more properties)

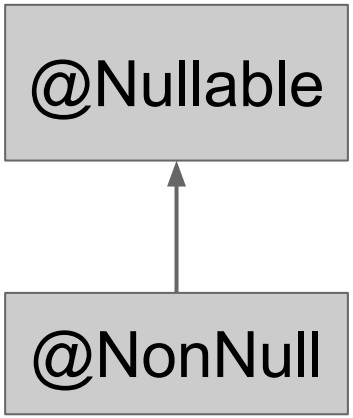
Type hierarchy for nullness

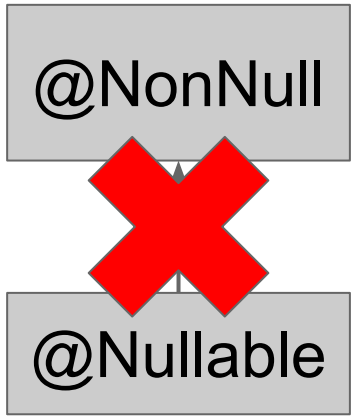


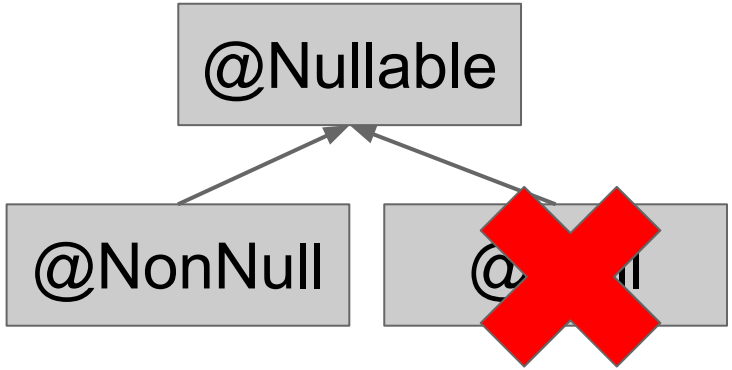
2 pieces of information:

- the types
- their relationships

Type hierarchy for nullness





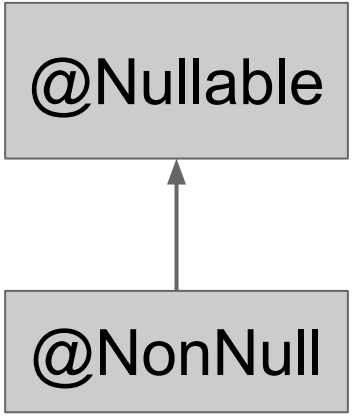


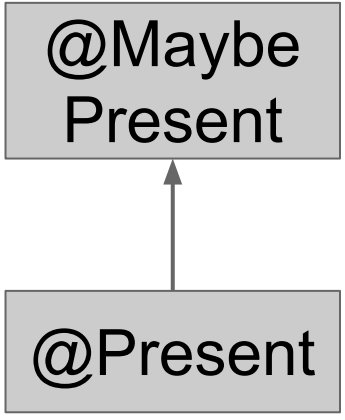
2 pieces of information:

- the types
- their relationships

Type hierarchy for Optional

“Never use Optional.get() unless you can prove that the Optional is present.”



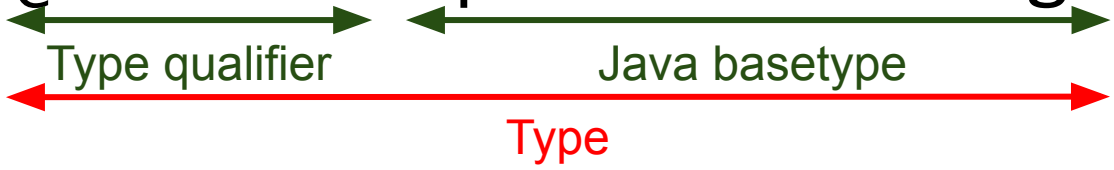


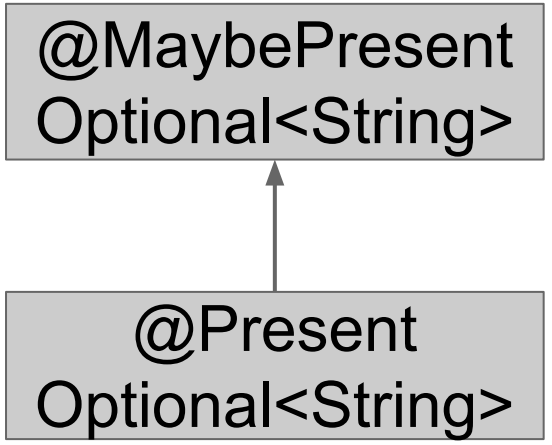
2 pieces of information:

- the types
- their relationships

Type = type qualifier + Java basetype

```
@Present Optional<String> maidenName;
```





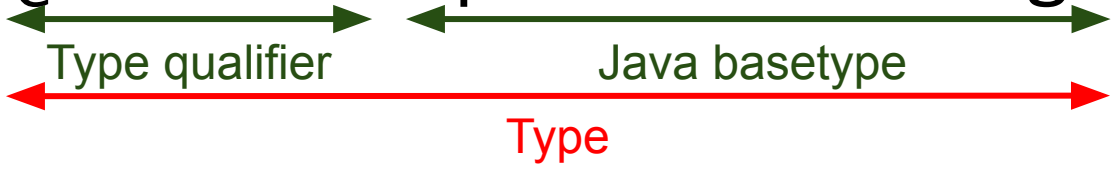
Default qualifier = `@MaybePresent`

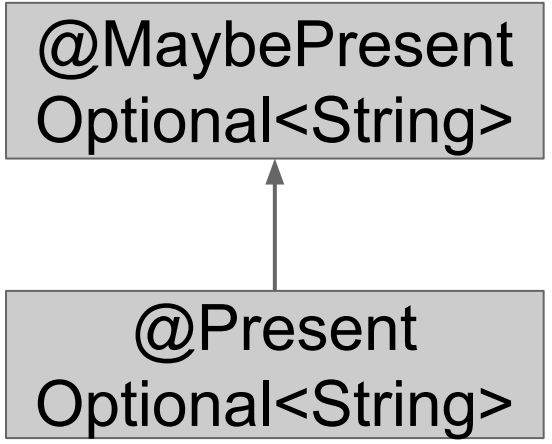
- `@MaybePresent Optional<String>`
- `Optional<String>`

equivalent

Type = type qualifier + Java basetype

```
@Present Optional<String> maidenName;
```





Default qualifier = `@MaybePresent`

- `@MaybePresent Optional<String>`
- `Optional<String>`

equivalent

Define a type system

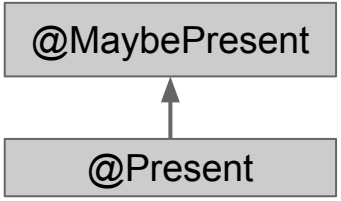
1. Type hierarchy (subtyping)
2. **Type rules (what operations are illegal)**
3. Type introduction (what types for literals, …)
4. Dataflow (run-time tests)

2. Type rules

To prevent null pointer exceptions:

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

Type rules for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

Only call `Optional.get()` on a receiver of type `@Present Optional`.

```
class Optional<T> {
  T get() { … }
}
```

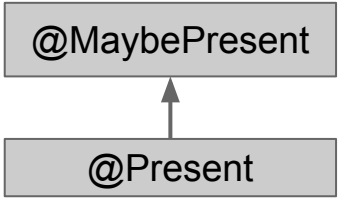
example call:

```
myOptional.get()
```

example call:

```
a.equals(b)
```


Type rules for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

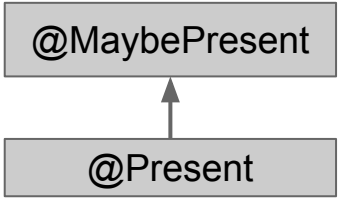
Only call `Optional.get()` on a receiver of type `@Present Optional`.

example call:

`myOptional.get()`

```
class Optional<T> {
  T get(Optional<T> this) { … }
}
```


Type rules for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

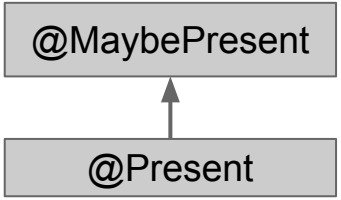
Only call `Optional.get()` on a receiver of type `@Present Optional`.

example call:

`myOptional.get()`

```
class Optional<T> {
  T get(@Present Optional<T> this) {…}
}
```


Type rules for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

Only call `Optional.get()` on a receiver of type `@Present Optional`.

example call:

`myOptional.get()`

```
class Optional<T> {
  T get(@Present Optional<T> this) {…}
  T orElseThrow(@Present O… this, …) {…}
}
```


Define a type system

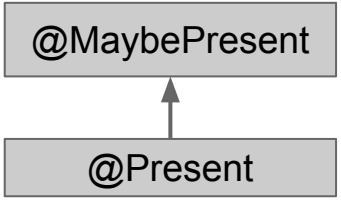
1. Type hierarchy (subtyping)
2. Type rules (what operations are illegal)
3. **Type introduction (what types for literals…)**
4. Dataflow (run-time tests)

Type introduction rules

For Nullness type system:

- `null : @Nullable`
- `"Hello World" : @NonNull`

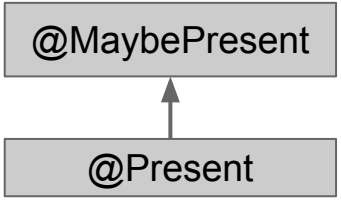
Type introduction for Optional



“Never use Optional.get() unless you can prove that the Optional is present.”

```
Optional<T> of(T value) {...}
Optional<T> ofNullable(T value){...}
```


Type introduction for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

```
@Present Optional<T> of(T value) {...}
Optional<T> ofNullable(@Nullable T value){...}
```


Define a type system

1. Type hierarchy (subtyping)
2. Type rules (what operations are illegal)
3. Type introduction (what types for literals, …)
4. **Dataflow (run-time tests)**

Flow-sensitive type refinement

After an operation, give an expression a more specific type

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

x is @NonNull here

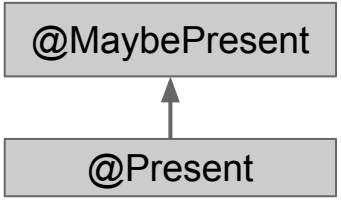
x is @Nullable again

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

y is @NonNull here

y is @Nullable again

Type refinement for Optional



"Never use Optional.get() unless you can prove that the Optional is present."

After `receiver.isPresent()` returns true, the receiver's type is `@Present`

```
@MaybePresent Optional<String> x;
if (x.isPresent()) {
  …
}
…
```

x is @Present here

x is @MaybePresent again

Now, let's implement it

Follow the instructions in the
Checker Framework Manual

https://checkerframework.org/manual/#creating-a-checker

Design the type system first

Before you start coding, first write the user manual.

What problem are you solving?

What qualifiers will you need?

What rules do you need to enforce?

Implement type qualifiers and hierarchy

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE,
        ElementType.TYPE_PARAMETER})
@SubtypeOf({MaybePresent.class})
public @interface Present {}
```

https://github.com/typetools/checker-framework/tree/master/checker/src/main/java/org/checkerframework/checker/optional/qual

Use the Subtyping Checker

Experiment with the type qualifiers using the Subtyping Checker:

```
javac -processor ...SubtypingChecker \
  -Aquals=...Present,...MaybePresent \
  SubtypeCheck.java
```


Name your checker

Simplify usage by adding a main class:

```
public class OptionalChecker
      extends BaseTypeChecker {}
```


Annotate parts of the JDK

Add `jdk.astub` JDK specifications

```
class Optional<T extends Object> {
   static <T extends Object> @Present Optional<T> of(T value) …
   T get(@Present Optional<T> this) …
   …
}
```

https://github.com/typetools/jdk/blob/master/src/java.base/share/classes/java/util/Optional.java

Type rules and type introductions

Not needed for the Optional Checker

Implement dataflow refinement

Declarative specification possible:

```
@EnsuresQualifierIf(result = true,
   expression = "this",
   qualifier = Present.class)
public boolean isPresent() {
    return value != null;
}
```


You can use the Optional Checker

Distributed with the Checker Framework

Checks 6 of the 7 rules for using `Optional`

https://checkerframework.org/manual/#optional-checker

Solution: Pluggable Type Checking

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

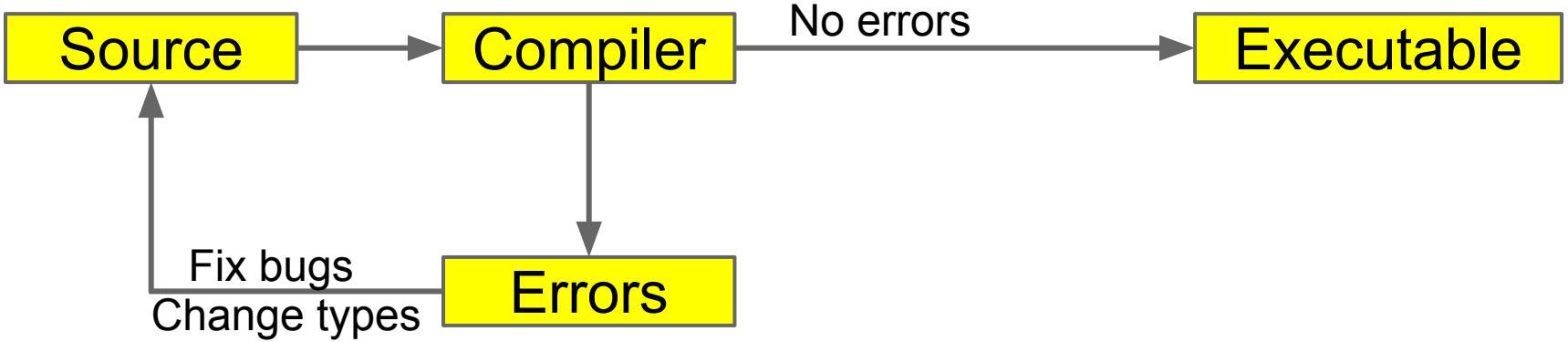
```
void foo (@Nullable Date date) {
  date.setSeconds(0); // compile-time error
```

3. Type checker warns about violations (bugs)

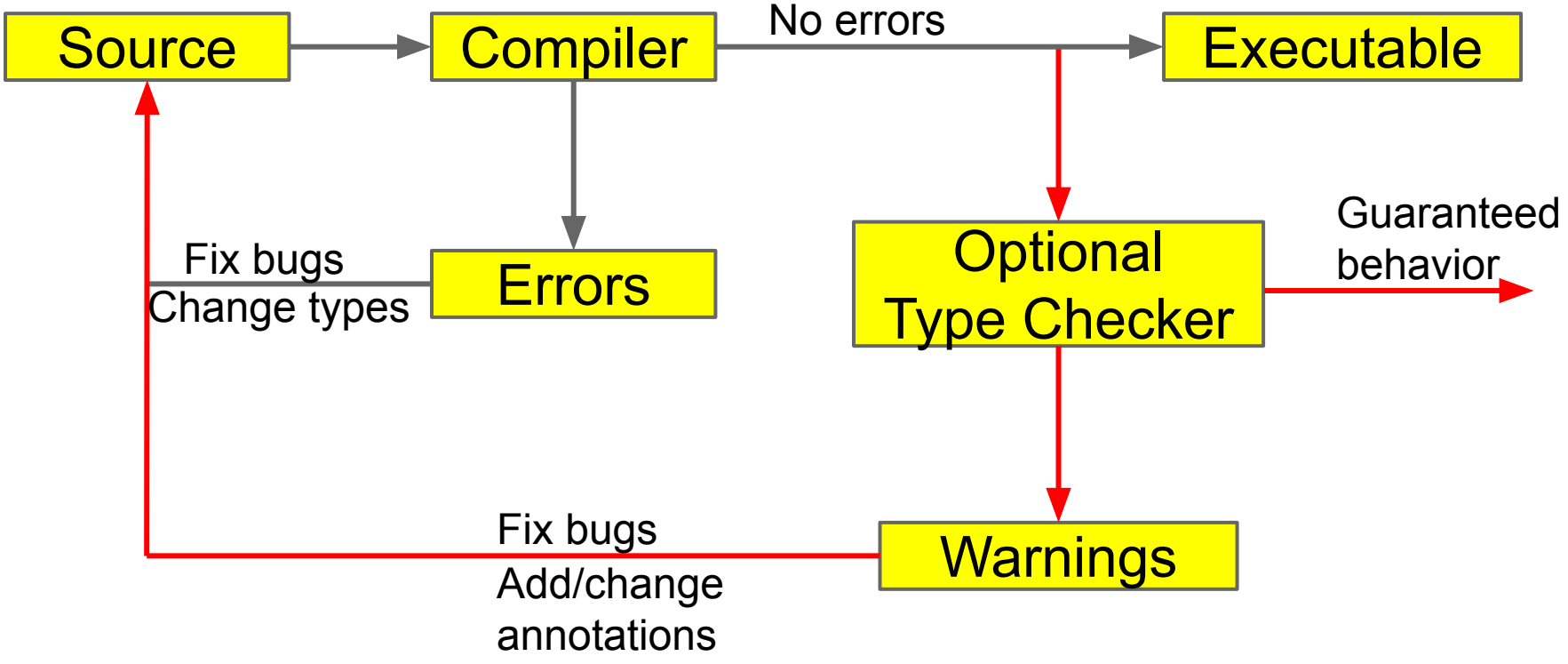
```
% javac -processor NullnessChecker MyFile.java

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

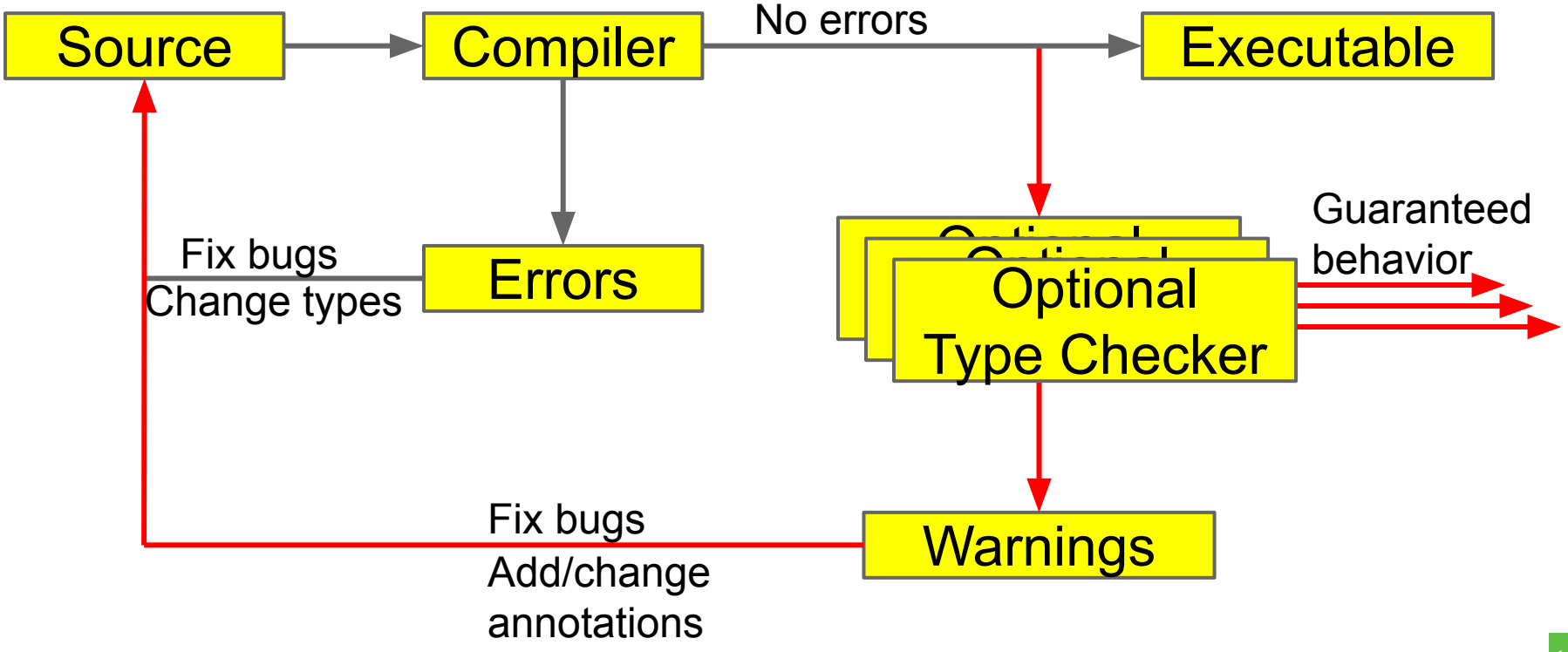

Type Checking



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 ...
```

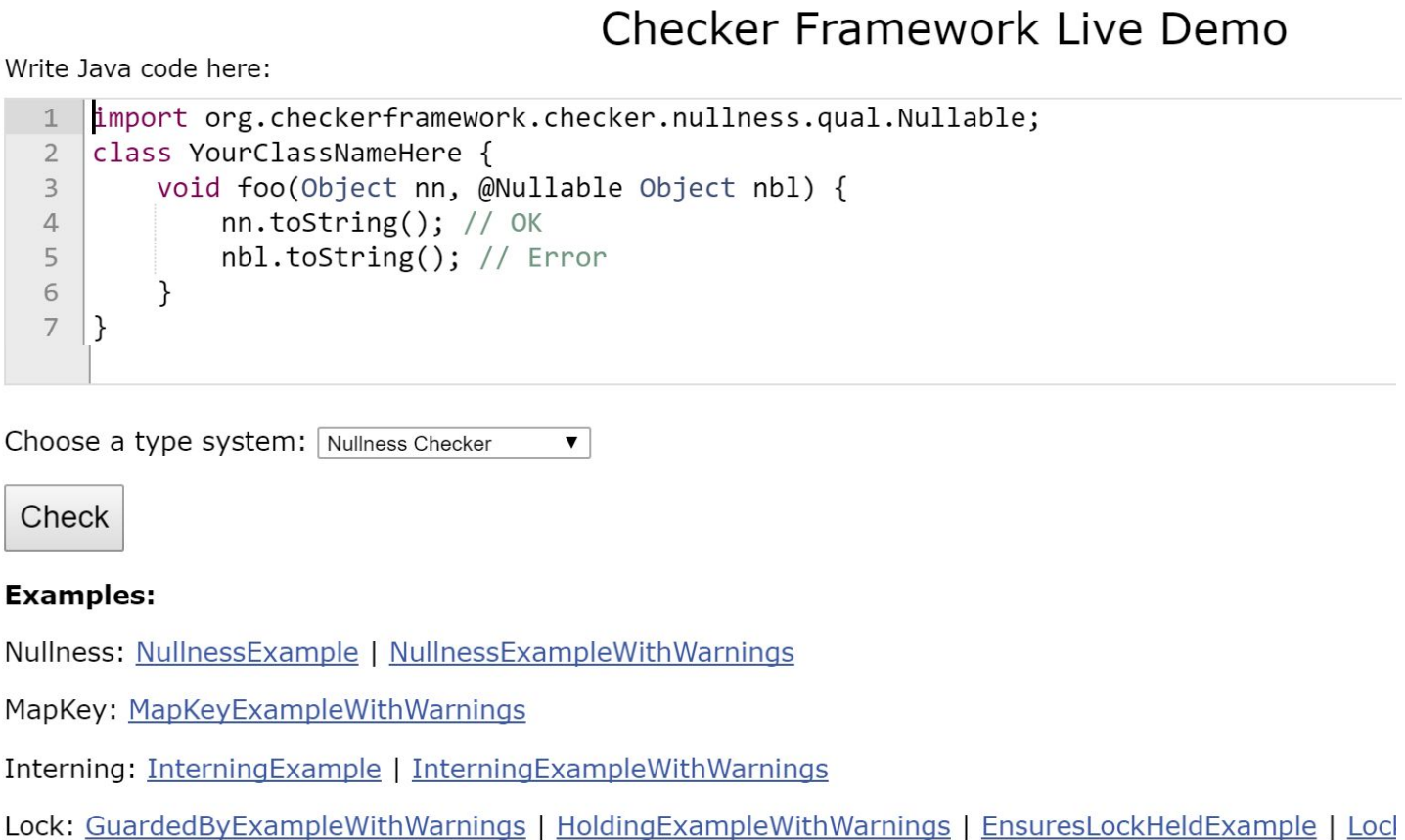
Standard error format allows tool integration

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>org.checkerframework</groupId>
      <artifactId>checker-qual</artifactId>
      <version>1.9.7</version>
    </dependency>
</dependencies>
```

Live demo: http://CheckerFramework.org/live/



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/`

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

Comparison: other nullness tools

	Null pointer errors		False warnings	Annotations written
	Found	Missed		
Checker Framework	9	0	4	35
FindBugs	0	9	1	0
Jlint	0	9	8	0
PMD	0	9	0	0
Eclipse, in 2017	0	9	8	0
Intellij (@NotNull default), in 2017	0	9	1	0
	3	6	1	925 + 8

Checking the Lookup program for file system searching (4kLOC)

Example type systems

Null dereferences (`@Nullable`)

>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
  - **`@Nullable`**: 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

Since Java 5: declaration 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 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:

```
@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, …

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

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
- >15,700 commits, 105 authors
- Welcoming & responsive community

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/`

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