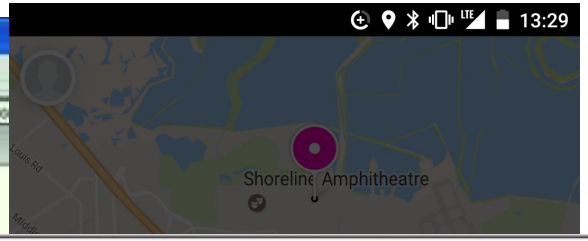
Custom type-checking for programming, teaching, and research

Michael D. Ernst
University of Washington
& Amazon Web Services

http://CheckerFramework.org/

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

```
        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.lang.NullPointerException

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();
```

Solution: Pluggable type-checking

Write stronger types (specifications)

- verification reveals errors
- static analysis: no effect on execution

Implementation: the Checker Framework

- used at Amazon, Google, Uber, startups, …
- compatible: uses standard Java syntax

Why use pluggable type-checking?

- For programming
- For teaching
- For research

Benefits for programmers

- **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)

Benefits for students

Teach students about specifications

Excite students about correctness & verification

Impose requirements on student code

Expose students to research

Results: higher grades; spent no more time

Researchers can build new analyses

Quickly prototype your analysis ideas

Framework handles full Java

    generics, type inference, inner classes, …

Enables large-scale experiments

Increases impact

(Building new analyses is the focus of this talk.)

Prevent null pointer exceptions

Java 8 introduced 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.

Enforce rules 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 dataflow or *type system* properties.

Define a type system

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

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

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

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

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

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

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

Define a type system (or any analysis)

1. **Type hierarchy** (subtyping)
2. **Type rules** (what operations are illegal)
3. **Type introduction** (type of new values)
4. **Dataflow** (run-time tests)

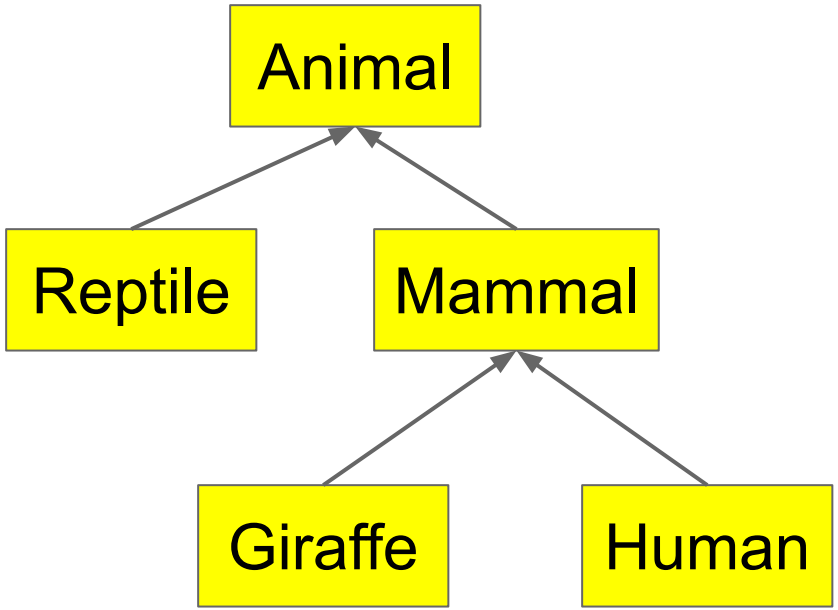
Main rule for the Optional type system:

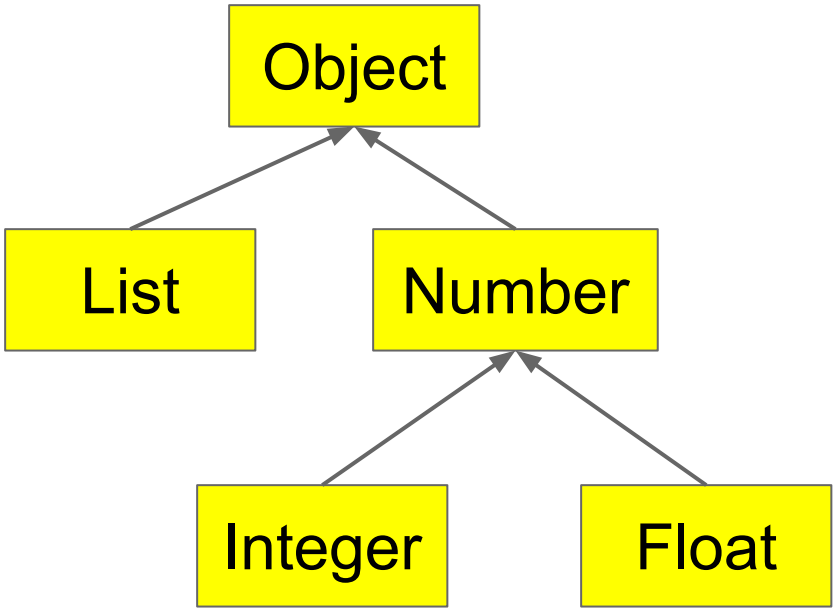
- “Never use Optional.get() unless you can prove that the Optional is present.”

Define a type system

1. **Type hierarchy (subtyping)**
2. Type rules (what operations are illegal)
3. Type introduction (type of new values)
4. Dataflow (run-time tests)

1. Type hierarchy



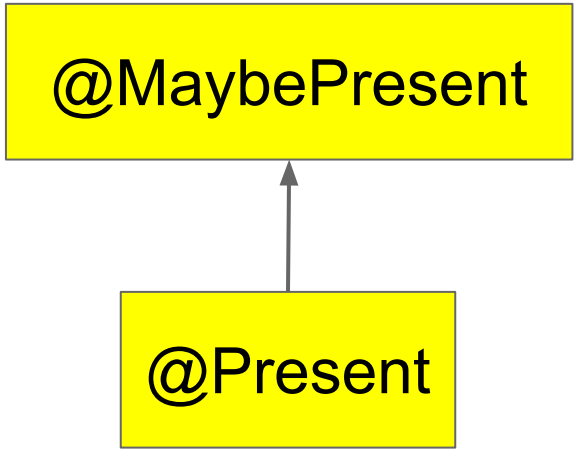


2 pieces of information:

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

Type hierarchy for Optional

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

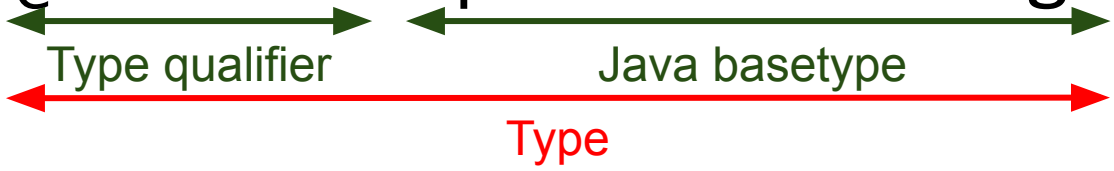


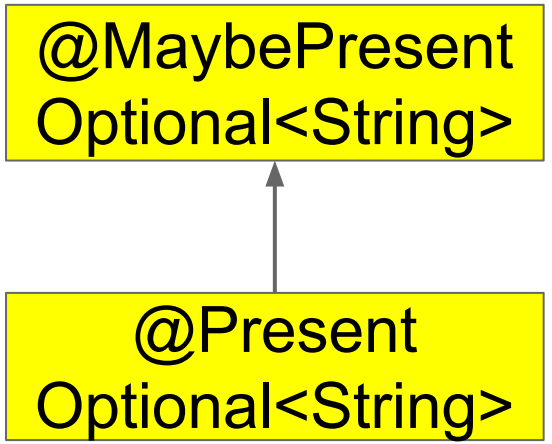
2 pieces of information:

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

Type = type qualifier + Java basetype

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





Default qualifier = `@MaybePresent`

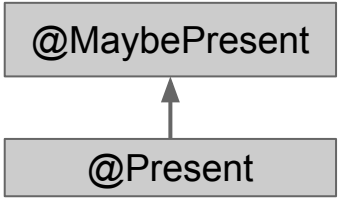
- `@MaybePresent Optional<String>`
- `Optional<String>`

equivalent

Define a type system

1. Type hierarchy (subtyping)
2. **Type rules (what operations are illegal)**
3. Type introduction (types of new values)
4. Dataflow (run-time tests)

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`.

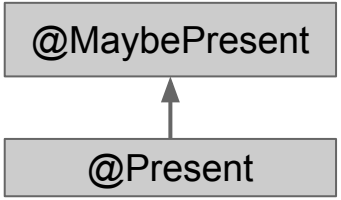
```
class Optional<T> {
 T get(Optional<T> this) { … }
}
```

example call:

```
myOptional.get()
```

`get()` takes 1 argument, the receiver

Type rules for Optional



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

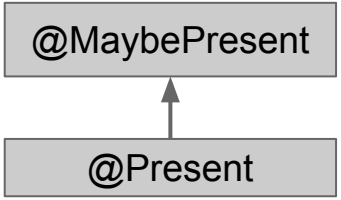
Only call `Optional.get()` on a receiver of type `@Present Optional`.

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:

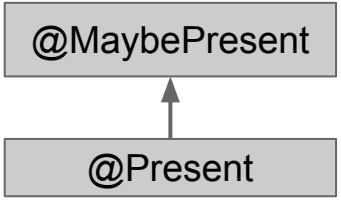
```
myOptional.get()
```

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

Define a type system

1. Type hierarchy (subtyping)
2. Type rules (what operations are illegal)
3. **Type introduction (type of new values)**
4. Dataflow (run-time tests)

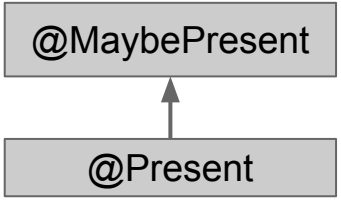
Type introduction for Optional



“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



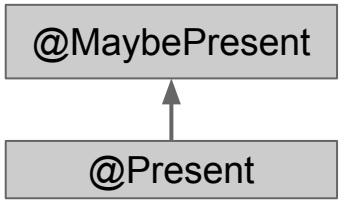
“Never use Optional.get() unless you can prove that the Optional is present.”

```
@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 (type of new values)
4. **Dataflow (run-time tests)**

Flow-sensitive type refinement



After an operation, give an expression a more specific type

```
@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

You can use the Optional Checker

Distributed with the Checker Framework

Checks 6 of the 7 rules for using `Optional`

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/`