Detecting and preventing null pointer errors with pluggable type-checking

CSE 331
University of Washington
Motivation

java.lang.NullPointerException
Java’s type checking is too weak

• Type checking prevents many bugs
  
  ```java
  int i = "hello";  // type error
  ```

• Type checking doesn’t prevent enough bugs

  ```java
  System.console().readLine();
  ⇒ NullPointerException

  Collections.emptyList().add(“One”);
  ⇒ UnsupportedOperationException
  ```
Some errors are silent

```java
Date date = new Date(0);
myMap.put(date, "Java epoch");
date.setYear(70);
myMap.put(date, "Linux epoch");
⇒ Corrupted map

dbStatement.executeQuery(userInput);
⇒ SQL injection attack
```

Initialization, data formatting, equality tests, ...
Problem: Your code has bugs

• Who discovers the problems?
  – If you are very lucky, testing discovers (some of) them
  – If you are unlucky, your customer discovers them
  – If you are very unlucky, hackers discover them
  – If you are smart, the compiler discovers them

• It’s better to be smart than lucky
Solution: Pluggable type systems

• Design a type system to solve a specific problem
• Write type qualifiers in code (or, use type inference)
  ```java
  @Immutable Date date = new Date(0);
  date.setTime(70); // compile-time error
  ```

• Type checker warns about violations (bugs)

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

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

• **Java 8**: annotations on types

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

• **Backward-compatible**: compile with any Java compiler

```java
List<*/@NonNull*/ String> strings;
```
Benefits of type qualifiers

- **Find bugs** in programs
- Guarantee the **absence of errors**
- **Improve documentation**
- Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
- Reduce number of assertions and run-time checks

- Possible negatives:
  - Must write the types (or use type inference)
  - False positives are possible (can be suppressed)
Nullness Checker demo

• Detect errors
• Guarantee the absence of errors
• Verify the correctness of optimizations
What bugs can you find & prevent?

- Null dereferences
- Mutation and side-effects
- Concurrency: locking
- Security: encryption, tainting
- Aliasing
- Equality tests
- Strings: localization, regular expression syntax
- Typestate (e.g., open/closed files)
- You can write your own checker!

The annotation you write:

- @NonNull
- @Immutable
- @GuardedBy
- @Encrypted
- @Untainted
- @Linear
- @Interned
- @Localized
- @Regex
- @State
Using a checker

- Run in IDE or on command line
- Works as a compiler plug-in (annotation processor)
- Uses familiar error messages

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

```
Console console = System.console();
console.printf("Password: ");
char[] password = console.readPassword();
```

![IDE screenshot showing error message]

**File.java**

**Description**: Dereference of possibly-null reference console
What is checked

• Proper use of the type hierarchy
  – assignments
  – method calls and returns
  – overriding

• Proper use of methods and operations
  – No dereferences of possibly-null values
What the checker guarantees

• Program satisfies type property
  – no bugs (of particular varieties)
  – no wrong annotations

• Caveat 1: only for code that is checked
  – Native methods
  – 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
Static and dynamic typing

• Static typing
  – Compiler guarantees that some errors cannot happen
    • The set of errors depends on the language
    • Other errors are still possible!
  – Examples: C, C++, Objective C, Java, C#, ML, Haskell

• Dynamic typing
  – The run-time system keeps track of types, and throws errors
  – Examples: Lisp, Scheme, Perl, PHP, Python, Ruby, JavaScript

• No type system
  – Example: Assembly
Why we ♥ static typing

• Documentation
• Correctness/reliability
• Refactoring
• Speed
Why we ♥ dynamic typing
(= Why we ☹ static typing)

• More concise code
  – Type inference is possible
• No false positive warnings
  Every static type system rejects some correct programs
  ```java
  @NonNull String lineSep
  = System.getProperty("line.separator");
  ```
• More flexible code
  – Add fields at run time
  – Change class of an object
• Ability to run tests at any time
  – Feedback is important for quality code
  – Programmer knows whether static or dynamic feedback is best
Nullness subtyping relationship

• Which type hierarchy is best?

@NonNull Date
@Nullable Date
@Nullable Date
@NonNull Date
@NonNull Date
@Nullable Date
@?? Date

• A subtype has fewer values
• A subtype has more operations
• A subtype is substitutable
• A subtype preserves supertype properties
**Mutability subtyping relationship**

- Which type hierarchy is best?

- **@Immutable** Date
  - no one can do mutation

- **@Mutable** Date
  - anyone can do mutation

- **@ReadOnly** Date
  - I can’t do mutation
  - No guarantee about mutation from elsewhere
Flow sensitivity

- Which calls are legal?

```java
Object name;
name = new Object();
name.toLowerCase();
name = "HELLO";
name.toLowerCase();
name = new Object();
name.toLowerCase();
```

```java
@Nullable String name;
name = null;
name.toLowerCase();
name = "HELLO";
name.toLowerCase();
name = null;
name.toLowerCase();
```
Flow sensitivity: name and legality

• Control flow determines the type
  
  ```java
  if (x==null) {
      ...
  } else {
      ...
  }
  ```

• What changes to the type are legal?
  - ```java
    String name;
    name = new Object();
    ...
  ``` // treat name as Object
  - ```java
    @NonNull String name;
    name = null;
    ...
  ``` // treat name as nullable

  Not these! It’s only legal to change to a subtype.
Flow sensitivity and type inference

Default for nullness checker:

- Non-null except locals
- Locals default to nullable (top of hierarchy)
- Flow-sensitivity changes this as needed

```java
@Nullable String name;
name = "hello";
... // treat name as non-null
```

```java
@Nullable String name;
name = otherNullable;
... // treat name as nullable
```

Rarely write annotations on local variables
The receiver is just another parameter

How many arguments does `Object.equals` take?

class MyClass {
    @Override
    public boolean equals(Object other) { … } } 

Two! Their names are `this` and `other`

Neither one is mutated by the method

```java
public boolean equals(@ReadOnly Object other) @readOnly { … }
```
Find the potential null pointer error

class C {
    @Nullable Object currentObj;

    // If currentObj is non-null, // prints it and a timestamp
    void printCurrent() {
        if (currentObj != null) {
            System.out.println(this.getTimeStamp());
            System.out.println(currentObj.toString());
        }
    }

    Object getTimeStamp() {
        ... }
}
Lack of side effects

class C {
    @Nullable Object currentObj;

    // If currentObj is non-null, // prints it and a timestamp
    void printCurrent() {
        if (currentObj != null) {
            System.out.println(this.getTimeStamp());
            System.out.println(currentObj.toString);
        }
    }

    @Pure
    Object getTimeStamp() { ... }
}
class C {
    @LazyNonNull Object currentObj;

    // If currentObj is non-null, // prints it and a timestamp
    void printCurrent() {
        if (currentObj != null) {
            System.out.println(this.getTimeStamp());
            System.out.println(currentObj.toString());
        }
    }

    Object getTimeStamp() { ...
}
}
Why doesn’t this typecheck?

class C {
    @Nullable Object f;

    void m1() {
        setF();
        f.hashCode();
    }

    @AssertNonNullAfter("this.f")
    void setF() {
        this.f = new Object();
    }
}

Type-checking is modular
Libraries you call must be annotated (much of the JDK is provided)
Why doesn’t this typecheck?

class C {
    Map<String, Date> m;
    String getDateString(String key) {
        return m.get(k).toString();
    }
}
Map keys

class C {
    Map<String, Date> m,
    String getDateString(@KeyFor("m") String key) {
        return m.get(k).toString();
    }
}
Another map key example

class C {
    Date getDate(Map<String, Date> m, String key) {
        return m.get(k);
    }

    void useDate(Map<String, Date> m) {
        String s = "now",
        Date d = new Date();
        m.put(s, d);
        getDate(s);
    }
}

Naming a formal parameter

```java
class C {
    Date getDate(Map<String, Date> m, 
            @KeyFor("#0") String key) {
        return m.get(key);
    }

    void useDate(Map<String, Date> m) {
        String s = "now",
        Date d = new Date();
        m.put(s, d);
        getDate(s);
    }
}
```

Use number, not name, for formal parameters. 😞
class C {
    Map<String, Date> m;
    String getDate(String key) {
        return m.get(k).toString();
    }

    void useDate(Map<String, Date> m) {
        String s = "now",
        Date d = new Date();
        m.put(d, d);
        getDate(s);
    }
}

How should identity be annotated?

String identity(String arg) {
    return arg;
}

void client() {
    // desired result:
    identity("hello").hashCode(); // OK; no warning
    identity(null).hashCode(); // compiler warning
}
How should identity be written?

These types are too specific:

```java
String identity(String arg) {
    return arg;
}
```

We want to say:

```java
ThatSameType identity(AnyType arg) {
    return arg;
}
```

In Java, this is expressed as:

```java
<T> T identity(T arg) {
    return arg;
}
```

**identity** has many types:
- String → String
- Integer → Integer
- List<Date> → List<Date>

Java automatically chooses the best type at each call site.

We also write this as: \( \forall T. T \rightarrow T \)

Java calls this a **generic method**

The standard term is **polymorphism**

We will learn about this soon
Polymorphism over nullness

```java
@PolyNull String identity(@PolyNull String arg) {
    return arg;
}

void client() {
    identity("hello").hashCode(); // OK; no warning
    identity(null).hashCode();    // compiler warning
}

@PolyNull is a hack that is necessary for non-generic methods
It is not necessary for generic methods:

// No annotations, but type-checks just like identity().
<T> T identity2(T arg) {
    return arg;
}
```
class Point {
    // rep invariant: either rep1 or rep2 is non-null
    XAndY rep1;
    RhoAndTheta rep2;

    float magnitude() {
        if (rep1 != null) {
            return Math.sqrt(rep1.x * rep1.x
                + rep1.y * rep1.y);
        } else {
            // We know rep2 is non-null at this point.
            return rep2.rho;
        }
    }
}

Summary of nullness annotations

@Nullable
@NonNull (rarely used)
    @LazyNonNull
Preconditions: @NonNullOnEntry
Postconditions:
    @Pure
    @NonNullAfter
    @NonNullIfTrue
    @NonNullIfFalse
Initialization: @Raw (rarely used)
Maps: @KeyFor
Polymorphism: @PolyNonNull (rarely used)