Lecture 19

Checker Framework
Motivation

java.lang.NullPointerException
Problem: Your code has bugs

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

• It’s better to be smart than lucky
Java’s type checking is too weak

• Type checking prevents many bugs
  
  int i = “hello”;  // type mismatch
  myString.getDate(); // method not found

• Type checking doesn’t prevent all bugs
  
  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, …
Type indicates legal operations

• Type checking prevents many bugs

  ```java
  int i = "hello";
  myString.getDate();
  ```

• Goal: avoid NullPointerException

• Idea: use types to indicate legality

• Consider references (pointers) as an ADT
  – Operation: dereferencing
  ```java
  x.field, x.method()
  ```
Types for null pointer prevention

Replace Object by two new types

• **NonNullObject**
  Dereference is permitted
  ```java
  NonNullObject nn;
  nn.field
  nn.method()
  ```

• **PossiblyNullObject**
  Dereference is forbidden
  ```java
  PossiblyNullObject pn;
  pn.field     // compile-time error
  pn.method()  // compile-time error
  ```

Problems:

• Can you use PossiblyNullObject for anything?
• Must rewrite all your Java applications and libraries
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;
```
Compile-time checking

1. Write type qualifiers in code

   ```java
   @NonNull Date date1 = new Date();
   @Nullable Date date2 = null;
   ```

2. Type checker warns about violations (bugs)

   ```java
   date1.setTime(70); // OK
   date2.setTime(70); // compile-time error
   ```
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)
Types for null-pointer-prevention

• Which type hierarchy is best?

• 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: no one can do mutation
@Mutable: anyone can do mutation
@ReadOnly
– I can’t do mutation
– No guarantee about mutation from elsewhere
What bugs can you find & prevent?

- Null dereferences @NonNull
- Mutation and side-effects @Immutable
- Concurrency: locking @GuardedBy
- Security: encryption, tainting @Encrypted @Untainted
- Aliasing @Linear
- Equality tests @Interned
- Strings: localization, regular expression syntax @Localized @Regex
- Typestate (e.g., open/closed files) @State
- You can write your own checker!

The annotation you write:
Using a checker

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

% javac -processor NullnessChecker MyFile.java

MyFile.java:9: incompatible types.
    nonNullVar = nullableValue;
    ^
found   : @Nullable String
required: @NonNull String
Using a checker

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

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

MyFile.java:9: incompatible types.
  nonNullVar = nullableValue;
  ^
  found   : @Nullable String
  required: @NonNull String
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 may contain an error
Static and dynamic typing

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

- **Dynamic typing**
  - Run-time system tracks types, and throws errors
  - Examples: Racket, Perl, PHP, Python, Ruby, JS

- **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
Advanced features

Avoiding the limitations of the conservative, static type-checker
Flow sensitivity

- Control flow determines the type
  
  ```java
  if (x==null) {
    ...  // treat as nullable
  } else {
    ...  // treat as non-null
  }
  ```

- Can refine the type to a subtype
More flow sensitivity

- Which calls type-check? Which calls ought to?

```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: permit changes

Legal changes: change to a **subtype**

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

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

Illegal changes: change to a **supertype**

Violates the declaration

```java
String name;  
name = new Object();  
...  // treat name as Object
```

```java
@NonNull String name;  
name = null;  
...  // treat name as nullable
```
Local type inference

Bottom line:

Rarely write annotations on local variables

Default for nullness checker:

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

How many arguments does `Object.equals` take?

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

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

Neither one is mutated by the method

- Java 8 syntax:
  ```java
  public boolean equals(@Readonly MyClass this, @ReadOnly Object other) {…}
  ```
  Optional syntax, for annotations

- For backwards compatibility:
  ```java
  public boolean equals(/*>>>@Readonly MyClass this,*/
                      @ReadOnly Object other) {…}
  ```
Find 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() { ... }
}
Lazy initialization

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 – reason from specs, not from implementation Libraries you call must be annotated. (much of the JDK is provided)
Why doesn’t this typecheck?

// Default: @NonNull
class C {
    Map<String, Date> m;
    String getDateString(String k) {
        return m.get(k).toString();
    }
}

Non-null map from non-null String to non-null Date
Non-null String
Non-null String
Possible NullPointerException
Map keys

// Default: @NonNull
class C {
    Map<String, Date> m;
    String getDateString(@KeyFor("m") String k) {
        return m.get(k).toString();
    }
}

Map.get returns null if the key is not in the map
Map is a formal parameter

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

    void useDate(Map<String, Date> m) {
        String s = "now",
        Date d = new Date();
        m.put(s, d);
        getDate(s);
    }
}
class C {
    Date getDate(Map<String, Date> m,
                 @KeyFor("#1") 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);
    }
}

Use number, not name, for formal parameters. 😞
Start counting at 1.
How to annotate identity?

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: ∀T. T → T

Java calls this a generic method

The standard term is polymorphism
Polymorphism over nullness

@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 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;
        }
    }
}
How to run the Nullness Checker

• `ant check-nullness`
• Run ant from within Eclipse
• Eclipse plug-in

More resources:
• Checker Framework manual
  
  http://types.cs.uw.edu/checker-framework/
Why run the Nullness Checker?

• In Winter 2011:
  – *Every* student discovered null pointer bugs
  – Students wished they had been using the Nullness Checker from the beginning of the quarter
Nullness annotation summary

@Nullable
@NonNull (rarely used)
   @LazyNonNull
Preconditions: @NonNullOnEntry
Postconditions:
   @Pure
   @AssertNonNullAfter
   @AssertNonNullIfTrue
   @AssertNonNullIfFalse
Initialization: @Raw (rarely used)
Maps: @KeyFor
Polymorphism: @PolyNull (rarely used)
Key ideas

- Many “run-time errors” can actually be prevented at compile time
- A type system is a simple way of doing so
- A stronger type system more expressive
  - This can be good or bad
- More practice understanding subtyping