A Deep Dive into the Void

Advanced null Type Annotations

Java 8 ready

Stephan Herrmann
GK Software
An Old Problem

- 1965
  - Tony Hoare introduced Null references in ALGOL W
  - “simply because it was so easy to implement”
Still Leading the Charts
An Expensive Problem

• “The Billion Dollar Mistake”

Photograph by Rama, Wikimedia Commons, CC-by-sa-2.0-fr
Many Steps to a Solution

- **do nothing:**
  - boooooooooom
- **recognize patterns**
  - booooooom
- **local flow analysis**
  - booom
- **method contracts**
  - boom
- **type checking**

- Since ALGOL W 1965
- Since Eclipse 3.1 2005
- Since Eclipse Juno / Java 5 2012
- Since Eclipse Luna / Java 8 2014
Alternative Strategy

• Optional
  – a polite way of saying: this value could be missing
  – supports null-less programming
  – doesn't help against existing nulls
  – viable option for from-scratch clean-room new software
  – requires all software to be re-written
Obviously Correct

- Typechecking for null analysis
- Two kinds of types
  - those that include null
  - those that don't

- This is OK:

```java
Dog? barkAndReturn(Dog dog) {
    dog.bark();
    return null;
}
```

- These are type errors:

```java
Dog barkAndReturn(Dog? dog) {
    dog.bark();
    return null;
}
```
Typesystem

Strategy

- Richer types
  - add more information
- Richer type checking
  - new typing rules
- Stronger guarantees
  - proven absence of errors
  - no methodNotUnderstood
  - no ClassCastException

JSR 308

- Type annotations
  - @NonNull @Nullable
  - org.eclipse.jdt.annotation_2.0
- ECJ checks nullness
  - compatibility & flow
- Stronger guarantees
  - no NPE
  - no SWTException(thread)
  - …
Ternary or Boolean?

- **Buy @NonNull & @Nullable, get 3 kinds of types**
  - green, red, legacy

- **Interfacing with legacy code**
  - like using raw types in Java 5+
    - Null type safety (type annotations): The expression of type 'List<String>' needs unchecked conversion to conform to '@NonNull List<@NonNull String>'

- **Nirwana: ubiquitous @NonNullByDefault**
  - per class / interface
  - per package (use package-info.java)
  - let the compiler complain about missing @NNBD

- **And back to 2014 reality**
  - @NonNullByDefault({})
Where is: Everywhere?

DefaultLocation for `@NonNullByDefault`

```
@NonNullByDefault({ ... })
class Foo <T extends Bar> {
    Foo<String> stringFoo;
    Bar[][] bars;
    Bar getBar(String name) {
        return bar;
    }
}
```
Where is: Everywhere?

DefaultLocation for @NonNullByDefault

- Not supported
  - non-null by construction
    - new
    - throws, catch argument
    - this
    - receiver in method reference
  - would need runtime evaluation
    - instanceof & cast
  - no useful interpretation
    - prefix of static member
    - constructor
  - future
    - type declaration
Where is: Everywhere?

DefaultLocation for @NonNullByDefault

- Not supported
- No default
  - local variables
    - better covered by flow analysis
  - on type variables & wildcards
    - these obtain nullness via other channels
Where is: Everywhere?

DefaultLocation for `@NonNullByDefault`

```
@NonNullByDefault({ ... })
class Foo <T extends Bar> {
    Foo<String> stringFoo;
    Bar[][] bars;
    Bar getBar(String name) {
        return bar;
    }
}
```

Default default
Where is: Everywhere?

**DefaultLocation for @NonNullByDefault**

- PARAMETER
- RETURN_TYPE
- FIELD
- TYPE_PARAMETER
- TYPE_BOUND
- TYPE_ARGUMENT
- ARRAY_CONTENTS

```java
@NonNullByDefault
class Foo <T extends Bar> {
    Foo<String> stringFoo;
    Bar[][] bars;
    Bar getBar(String name) {
        return bar;
    }
}
```

**Default default**
Great Expectations

- **ECJ will not “make” your program “null safe”**
- **It will hurt**
  - pin-point
    - design decisions you failed to make
    - contradictory assumptions
  - type checker will need your help
  - rules & obligations
- **Application development**
  - free to move obligations around
- **Tension in API design**
  - strict for effective null checking
  - flexible for wide range of re-use

```
Object dog4 = new Dog();
dog4.bark();
```

a false positive?
Generic API

• The classic declaration
  - Unconstrained type parameter:  
    - public interface List<T> {
  ...
  }

• Client side
  - Free to choose the type argument:
    - List<@NonNull Person>
    - List<@Nullable Person>

• Implementer
  - No knowledge about “free” type variable T
  - Must assume the worst
    - need to check on dereference
    - cannot assign null to a T variable
Constraining Type Parameters

- **Admit only @NonNull types:**
  - class `MyGenericClass1<T extends @NonNull Object>`
  - legal substitutes: @NonNull Object, @NonNull String
  - illegal: @Nullable Object
  - because @NonNull X <: @Nullable X

- **Admit only @Nullable types:**
  - we can't say `<T super @Nullable Object>`
  - class `MyGenericClass2<@Nullable T>`
  - (interpreted as both upper and lower bound)
Puzzle 1

- Legal or not?

```java
@NonNullByDefault class Tricky<T> {
    void foo(T t) {}
    void test() { foo(null); }
}
```

- where does @NonNullByDefault apply
  - type parameter?
    - no
  - method parameter?
    - generally yes
    - but not to a type variable

Null type mismatch (type annotations): 'null' is not compatible to the free type variable 'T'

... Wildcards and the use of type variables are always excluded from NonNullByDefault.
Puzzle 2

• What is this?

```java
@NonNullByDefault
class Tricky<T> {
    void foo(@Nullable T t) {}
    void test() { foo(null); }
}
```

• “You said 'T' could mean anything”
  - so what is @Nullable T?
    • e.g., if T := @NonNull String?
    • e.g., @Nullable String
    - the @Nullable variant of whatever T represents
    • e.g., @Nullable String
• Double Inference
• External Annotations