# Java 8: Type Annotations

by Daniel Stankiewicz

## about.me

(mostly) Java Developer since 2005
currently @AssecoPolandSA
inspired by Java 8
privately husband, dad, guitarist

## Java 8 New Features

and Type Annotations amongst them

## Annotations on Java Types (JSR-308)

A brief look at Open JDK Type Annotations project:

JSR 308, Annotations on Java Types lays the foundations for stronger typing in Java by extending the language to allow annotations on essentially any use of a type.

## First Naive Test

```java
public List<@TypeUse String> typeUse() {
    @TypeUse String s = "abc";
    List<@TypeUse String> list = new ArrayList<>();
    list.add(s);
    list.add("def");
    boolean isTypeUse = "def" instanceof @TypeUse String;
    System.out.println("@TypeUse String: " + isTypeUse);
    return list;
}
```

Wow! No compilation errors!!!

## A Deeper Dive into Specification

or a piece of more organized knowledge

## A Deeper Dive into Specification

### Declaration vs type context

- Before Java 8 annotations could be used only in declaration contexts
- Annotation could be accessed by reflection, except for `ElementType.LOCAL_VARIABLE`
- Java 8 extends declaration context to the use of type parameter declarations.
- It also introduces type context for annotations - types used in declarations and expressions.

## A Deeper Dive into Specification

## Declaration context @Target

- ElementType.PACKAGE
- ElementType.TYPE
- ElementType.ANNOTATION_TYPE
- ElementType.FIELD
- ElementType.CONSTRUCTOR
- ElementType.METHOD
- ElementType.PARAMETER
- ElementType.LOCAL_VARIABLE

## A Deeper Dive into Specification

## Declaration context @Target

- ElementType.PACKAGE
- ElementType.TYPE
- ElementType.ANNOTATION_TYPE
- ElementType.FIELD
- ElementType.CONSTRUCTOR
- ElementType.METHOD
- ElementType.PARAMETER
- ElementType.LOCAL_VARIABLE
- ElementType.TYPE_PARAMETER

## A Deeper Dive into Specification

### Declaration context @Target

- ElementType.PACKAGE
- ElementType.TYPE
- ElementType.ANNOTATION_TYPE
- ElementType.FIELD
- ElementType.CONSTRUCTOR
- ElementType.METHOD
- ElementType.PARAMETER
- ElementType.LOCAL_VARIABLE
- ElementType.TYPE_PARAMETER

### Type context @Target

- ElementType.TYPE_USE

## Declaration context @Target

- ElementType.PACKAGE
- ElementType.TYPE
- ElementType.ANNOTATION_TYPE
- ElementType.FIELD
- ElementType.CONSTRUCTOR
- ElementType.METHOD
- ElementType.PARAMETER
- ElementType.LOCAL_VARIABLE
- ElementType.TYPE_PARAMETER

## Type context @Target

- ElementType.TYPE_USE

`ElementType.TYPE_USE` includes elements of declaration context

## A Deeper Dive into Specification

## ElementType.TYPE_PARAMETER usages

Type parameter of generic class (or interface)

```
public class GenericClass<@TypeParameter V extends Number>
```

Type parameter of generic constructor

```
public <@TypeParameter A> GenericClass(V value, A another)
```

Type parameter of generic method

```
public <@TypeParameter K> Map<K, V> getSingletonMap(K key)
```

## A Deeper Dive into Specification

## ElementType.TYPE_PARAMETER reflection

Since Java 8 `java.lang.reflect.TypeVariable` interface extends `java.lang.reflect.AnnotatedElement` interface:

```
TypeVariable<Class<GenericClass>>[] typeParameters =
    GenericClass.class.getTypeParameters();
TypeParameter annotation =
        typeParameters[0].getAnnotation(TypeParameter.class);
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Declaration context

```
@TypeUse("class")
public class DeclarationContext
                <@TypeUse("type parameter") V extends Number> {

    public <@TypeUse("type parameter") K> Map<K, V> getMap(K key) {}

    @TypeUse("annotation") public @interface AnnotationExample {}
    @TypeUse("interface") public interface InterfaceExample {}
    @TypeUse("enum") public enum EnumExample {}
}
```

Usages of `ElementType.TYPE_USE` in declaration contexts can be also accessed by reflection.

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in declarations

`extends` or `implements` clause of a class declaration

```
public class TypeContext implements @TypeUse Serializable
```

`extends` clause of an interface declaration

```
public interface InterfaceExample extends @TypeUse Runnable
```

Return type of a method

```
public @TypeUse String getValue()
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in declarations

`throws` clause of a method or constructor

```
public void throwException() throws @TypeUse Exception
```

`extends` clause of a type parameter declaration

```
public <N extends @TypeUse Number> Integer getInteger(N number)
```

Field declaration including an enum constant

```
private @TypeUse String value;
private enum Status {@TypeUse OK, @TypeUse ERROR}
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in declarations

Parameter of a method, constructor, or lambda expression

```
public List<Car> filterCars(@TypeUse List<Car> cars, int year) {
    return cars.stream()
            .filter((@TypeUse Car c) -> c.getYear() == year)
            .collect(Collectors.toList());
}
```

Receiver parameter of a method (explicit `this` parameter)

```
public void show(@TypeUse TypeContext this, boolean other)
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in declarations

Local variable declaration

```
@TypeUse Date date;
```

Exception parameter of a `catch` clause

```
catch (@TypeUse NumberFormatException e)
```

## ElementType.TYPE_USE usages

### Type context in declarations != Declaration context

Some of the above examples may look like use of annotations of declaration context, but don't be tricked, they are not!

They cannot be accessed by reflection:

```
private @TypeUse String value;

TypeContext.class.getDeclaredField("value")
        .getDeclaredAnnotation(TypeUse.class)
```

returns `null`.

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in expressions

Explicit type parameter of a constructor or method invocation or `new` operator

```
<@TypeUse Long>this(0l);
new <@TypeUse String> GenericClass<@TypeUse Long>(123l, "abc")
        .<@TypeUse Integer> getSingletonMap(123);
```

Class instance creation (including anonymous)

```
new @TypeUse Thread(new @TypeUse Runnable() {
    @Override
    public void run() {}
});
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in expressions

Type of element in array creation

```
Integer[] array = new Integer @TypeUse[] {};
```

Cast operator

```
String s = (@TypeUse String) "abc";
```

`instanceof` operator

```
boolean isInstance = s instanceof @TypeUse String;
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in expressions

Method reference expression

```
@TypeUse GenericClass::new
List<@TypeUse String>::size
Arrays::<@TypeUse Integer> sort
```

Element type of an array

```
@TypeUse int[] a;    // annotates the primitive type int
int @TypeUse[] b;    // annotates the array type int[]
int @TypeUse[][] c;  // annotates the array type int[][]
int [] @TypeUse[] d; // annotates the array type int[] as component
                     // of array int[][]
```

## A Deeper Dive into Specification

## ElementType.TYPE_USE usages

### Type context in expressions

Type argument of a parametrized type

```
List<@TypeUse Integer> integers;
Set<? extends @TypeUse Number> set;
Collection<? super @TypeUse Number> collection = new ArrayList<>();
```

## Quasi ElementType.TYPE_USE usages

Warning! The following are not type uses and such annotations are not allowed here

```
// Annotation uses
@/* quasi @TypeUse */Deprecated String s;

// Class literal
Class<GenericClass> clazz = /* quasi @TypeUse */GenericClass.class;

// Import syntax
import java.io./* quasi @TypeUse */Serializable;

// Static member access
String title = /* quasi @TypeUse */ GenericClass.TITLE;

// Scoping (e.g. inner class)
String v = /* quasi @TypeUse */TypeContext.this.getValue();
```

## The Checker Framework

or who is the main culprit?

## The Checker Framework

### A piece of history

- Rationale for JSR-308 was to allow building tools for more thorough source code analysis
- The Most Innovative Java SE/EE JSR of the Year award in... 2007 :)
- The Checker Framework, an extendable set of compiler plugins that find bugs, has been already up and running much before Java 8 with a bunch of other tools
- Until Java 8, Checker Framework's own compiler had to be used to process type annotations, also written in comments: `/* @TypeUse */`
- Better late than never - type annotations in Java 8 in 2014 :)

## What do we have out of the box?

- Nullness Checker
- Initialization Checker
- Map Key Checker
- Interning Checker
- Lock Checker
- Fake Enum Checker
- Tainting Checker
- Regex Checker
- Format String Checker
- Property File Checker
- Internationalization Checker
- Signature String Checker
- Units Checker
- Linear Checker
- Mutation Checkers
- Subtyping Checker

Each checker with a set of suitable annotations, there is also ability to write our own checkers.

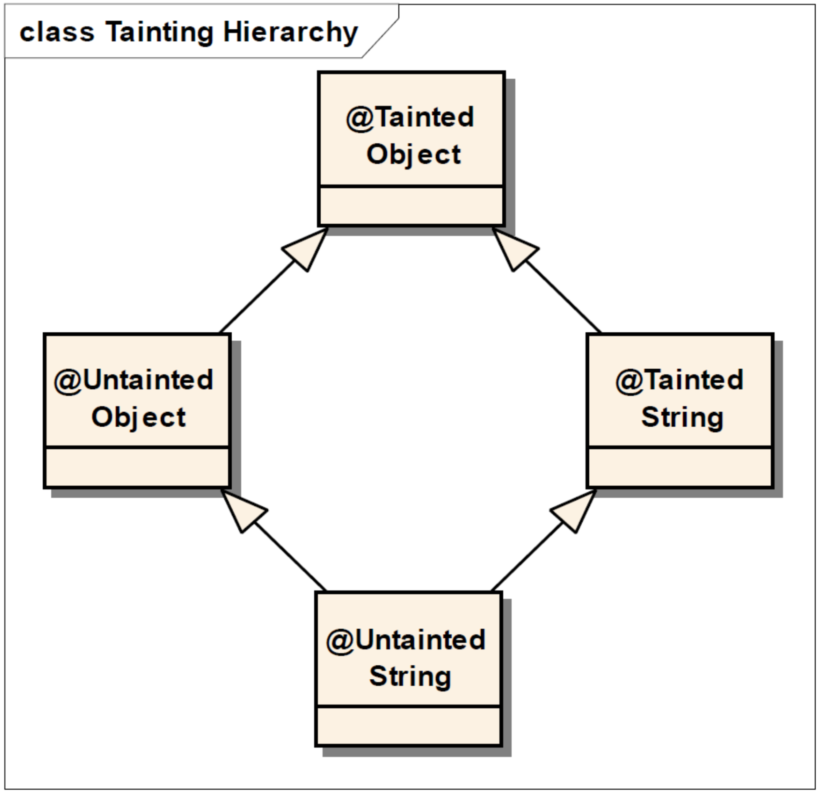
We have pre-annotated JDK classes and a way to annotate external libraries by file stubs.

## Annotations as Type Qualifiers

In the Checker Framework type annotations act as type qualifiers:

- `@Nullable Integer` - an `Integer` that may become `null`
- `@NonNull Date` - a `Date` that will never become `null`
- `@Regex String` - a `String` that is a valid regular expression
- `@Tainted String` - a `String` that may contain dangerous content
- `@Untainted String` - a `String` that has been checked and is safe to be used
- `@Interned Integer` - an `Integer` that can be safely tested for equality by reference comparison `==`
- `@ReadOnly List<String>` - a read-only list of `String` objects

## Tainting Checker Type Hierarchy



## Tainting Checker SQL Injection Prevention

Security sink in an application, it should allow only untainted values:

```
public String createQuery(@Untainted String lastName) {
    return "SELECT p.id FROM person p " +
           "WHERE p.last_name = " + lastName;
}
```

Method to sanitize values entered from the outside:

```
public @Untainted String sanitize(@Tainted String tainted) {
    for (char c : tainted.toCharArray()) {
        if (!Character.isLetterOrDigit(c)
            && !Character.isWhitespace(c)) {
            throw new IllegalArgumentException("SQL Injection!");
        }
    }
    return new @Untainted String(tainted);
}
```

## Tainting Checker SQL Injection Prevention

Proper, safe use of security sink:

```
public String getSafeQuery(@Tainted String lastName) {
    return createQuery(sanitize(lastName));
}
```

Vulnerability in code:

```
public String getVulnerableQuery(@Tainted String lastName) {
    // use of a @Tainted String without sanitizing
    return createQuery(lastName);
}
```

## Tainting Checker SQL Injection Prevention

Compiler doesn't warn, despite the vulnerability in code.

However Tainting Checker finds a security hole:

```
[INFO] --- checkerframework-maven-plugin:1.8.0:check (default)
    @ java8-type-annotations ---
[INFO] Running Checker Framework version: 1.8.0
[INFO] Running processor(s):
    org.checkerframework.checker.tainting.TaintingChecker
[INFO] Run with debug logging in order to view the compiler command line
[WARNING] ...src\main\java\pl\jug\warszawa\typeannotations\checker\
    TaintingExample.java:[31,27]
    [argument.type.incompatible] incompatible types in argument.
  found   : @Tainted String
  required: @Untainted String
```

Et voilà!

## Links

### JSR-308 Specification

http://types.cs.washington.edu/jsr308/specification/java-annotation-design.html

### Java 8 Language Specification - 9.6.4.1. `@Target`

http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6.4.1

### The Checker Framework

http://types.cs.washington.edu/checker-framework/

### Source code of this presentation

https://github.com/danielstankiewicz/type-annotations

# The End

Slideshow created with remark.