# Annotations in the Java Programming Language

Péter Jeszenszky
Faculty of Informatics, University of Debrecen
jeszenszky.peter@inf.unideb.hu

Last modified: January 14, 2020

## What is an Annotation?

- An annotation is a marker which associates information with a program construct, but has no effect at run time.

## History (1)

- They were introduced in J2SE 5.0, which was released in 2004.
  - See:
    - *New Features and Enhancements J2SE* 5.0 https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html
    - *JSR 175: A Metadata Facility for the Java Programming Language (Final Release)*. 30 September 2004. https://jcp.org/en/jsr/detail?id=175
- Released in 2006, Java SE 6 added further features (i.e., the `javax.annotation.processing` package).
  - *JSR 269: Pluggable Annotation Processing API (Final Release)*. 11 December 2006. https://jcp.org/en/jsr/detail?id=269

## History (2)

- Java SE 8, which was released in 2014, brought new features (type annotations, repeatable annotations, new predefined annotation types).
  - See: *What's New in JDK 8* https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

## History (3)

- Java SE 9:
  - *JEP 277: Enhanced Deprecation*
    http://openjdk.java.net/jeps/277

## History (4)

- Java SE 11:
  - Annotations can be applied to local variables and formal parameters of lambda expressions.
  - See: *JEP 323: Local-Variable Syntax for Lambda Parameters* https://openjdk.java.net/jeps/323

# Possible Uses

- **Information for the compiler**: for example, to suppress warnings or detect errors.
  - See, for example, the `@Deprecated` and `@Override` annotations.
  - *The Checker Framework* https://checkerframework.org/
- **Code generation**: annotations can be used for code generation.
  - *Java Architecture for XML Binding* (JAXB) https://github.com/eclipse-ee4j/jaxb-ri
  - *Project Lombok* https://projectlombok.org/
- **Runtime processing**: some annotations are available to be examined at runtime.
  - *JUnit* https://junit.org/
  - *Bean Validation*: is part of Java EE (see the `javax.validation` package and its sub-packages)
    - *JSR 380: Bean Validation 2.0 (Final Release)*. 3 August 2017. https://jcp.org/en/jsr/detail?id=380
    - Reference implementation: *Hibernate Validator* http://hibernate.org/validator/

## Equivalents in Other Languages

- **.NET**: attributes
  - *.NET Framework Development Guide – Extending Metadata Using Attributes* https://docs.microsoft.com/en-us/dotnet/standard/attributes/index
- **Python**: variable and function annotations (since version 3.0)
  - *PEP 526 – Syntax for Variable Annotations* https://www.python.org/dev/peps/pep-0526/
  - *PEP 3107 – Function Annotations* https://www.python.org/dev/peps/pep-3107/

## Specification

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith. *The Java Language Specification – Java SE 11 Edition*. 21 August 2018. https://docs.oracle.com/javase/specs/jls/se11/html/
  - See the following sections:
    - 9.6. *Annotation Types* https://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.6
    - 9.7. *Annotations* https://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.7

# Annotation Syntax (1)

- An annotation consists of:
  - The name of an annotation type.
  - Optionally, a list of comma-separated element-value pairs.
    - The list must be enclosed in parentheses.
- The annotation type corresponding to the name determines the element-value pairs available.
  - Elements with a default value can be omitted.
- The order of the element-value pairs is not important.
  - It is customary, though not required, that element-value pairs in an annotation are presented in the same order as the corresponding elements in the annotation type declaration.

# Kinds of Annotations

- **Normal annotations:**
  - `@XmlElement(name = "birthday",`
    `namespace = "http://xmlns.com/foaf/0.1/",`
    `required = true)`
- **Single element annotation**:
  - `@SuppressWarnings(value = "unchecked")`, `@SuppressWarnings("unchecked")`
  - `@Target(value = {ElementType.FIELD,`
    `ElementType.METHOD})`
    `@Target({ElementType.FIELD, ElementType.METHOD})`
- **Marker annotation**: when there are no element-value pairs, the `()` characters can be omitted.
  - `@NotNull`, `@NotNull()`

# Annotation Syntax (2)

- If the element type is an array type, then value must be provided with an array initializer expression.
  - Except that when the value is a single element array, in this case the curly braces can be omitted.
- For example, the following two annotations are equivalent:
  - `@Target({ElementType.METHOD})`
  - `@Target(ElementType.METHOD)`

## Where Annotations May Appear

- On declarations:
  - Including declarations of annotation types, constructors, fields, `enum` constants, local variables, methods, modules, packages, formal parameters, classes, interfaces and `enum`s, type parameters (Java SE 8)
- On type uses (Java SE 8)

## Predefined Annotation Types

- In the `java.lang` package:
  - `@Deprecated`
  - `@FunctionalInterface` (Java SE 8)
  - `@Override`
  - `@SafeVarargs` (Java SE 8)
  - `@SuppressWarnings`

- In the `java.lang.annotation` package:
  - `@Documented`
  - `@Inherited`
  - `@Native` (Java SE 8)
  - `@Repeatable` (Java SE 8)
  - `@Retention`
  - `@Target`

## @Deprecated (1)

- Indicates that the use of the annotated element should be avoided, because it is dangerous or because a better alternative exists.
  - It is strongly recommended that the reason for deprecating a program element be explained in the documentation, using the `@deprecated` Javadoc tag.
- Compilers issue warnings when a deprecated program element is used.
- The deprecated elements of Java SE 11: https://docs.oracle.com/en/java/javase/11/docs/api/deprecated-list.html

# @Deprecated (2)

```
// Character.java (OpenJDK 8):
package java.lang;

public final class Character implements java.io.Serializable,
        Comparable<Character> {
    ...

    /**
     * Determines if the specified character is permissible as the first
     * character in a Java identifier.
     * ...
     *
     * @param   ch the character to be tested.
     * @return  {@code true} if the character may start a Java
     *          identifier; {@code false} otherwise.
     * ...
     * @deprecated Replaced by isJavaIdentifierStart(char).
     */
    @Deprecated
    public static boolean isJavaLetter(char ch) {
        return isJavaIdentifierStart(ch);
    }
    ...
```

## @Deprecated (3)

- Java SE 9 introduced the following two optional elements:
  - `since`: specifies the version in which the annotated element became deprecated (default: `""`)
  - `forRemoval`: indicates whether the annotated element is subject to removal in a future version (default: `false`)
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Deprecated.html

## @Deprecated (4)

- Example:

```
// Runtime.java (OpenJDK 11):
package java.lang;
...
public class Runtime {
    ...

    /**
     * Not implemented, does nothing.
     *
     * @deprecated
     * This method was intended to control instruction tracing.
     * It has been superseded by JVM-specific tracing mechanisms.
     * This method is subject to removal in a future version of Java SE.
     *
     * @param on ignored
     */
    @Deprecated(since="9", forRemoval=true)
    public void traceInstructions(boolean on) {}
    ...
```

## @Deprecated (5)

- Java SE 9:
  - A command-line static source code analyzer tool for detecting uses of deprecated API elements (`jdeprscan`).
    - Examples:
      - `jdeprscan commons-io-2.6.jar`
      - `jdeprscan lib/*.jar`
    - See: https://docs.oracle.com/javase/9/tools/jdeprscan.htm
  - Compilers do not issue a warning anymore when a deprecated type or member is imported.
    - See: *JEP 211: Elide Deprecation Warnings on Import Statements* http://openjdk.java.net/jeps/211

## @SuppressWarnings (1)

- Indicates that the named compiler warnings should be suppressed in the annotated element (and in all program elements contained in the annotated element).
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/SuppressWarnings.html

# @SuppressWarnings (2)

- Example:

```
@SuppressWarnings("unchecked")
public ArrayList<String> getMusketeers() {
    ArrayList musketeers = new ArrayList();
    musketeers.add("D'Artagnan");
    musketeers.add("Athos");
    musketeers.add("Aramis");
    musketeers.add("Porthos");
    return musketeers;
}
```

```
import java.util.Date;
...
@SuppressWarnings("deprecation")
public static Date getDDay() {
    return new Date(1944 - 1900, 6 - 1, 6);
}
```

## @Override (1)

- Indicates that a method declaration is intended to override a method declaration in a supertype.
- Although it is not required to use this annotation when overriding a method, it helps to prevent errors.
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Override.html

# @Override (2)

```
// Integer.java (OpenJDK 11):
package java.lang;

public final class Integer extends Number implements
        Comparable<Integer> {
    ...

    /**
     * Returns a hash code for this {@code Integer}.
     *
     * @return a hash code value for this object, equal to the
     *         primitive {@code int} value represented by this
     *         {@code Integer} object.
     */
    @Override
    public int hashCode() {
        return Integer.hashCode(value);
    }
    ...
```

## @FunctionalInterface (1)

- Indicates that an interface type declaration is intended to be a functional interface.
  - A functional interface has exactly one abstract method.
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/FunctionalInterface.html

## @FunctionalInterface (2)

- Example:

```
// FileFilter.java (OpenJDK 11):
package java.io;

@FunctionalInterface
public interface FileFilter {

    boolean accept(File pathname);

}
```

# @SafeVarargs (1)

- Suppresses certain warnings about variable arity methods.
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/SafeVarargs.html

## @SafeVarargs (2)

- Example:

```
// Collections.java (OpenJDK 11):
package java.util;

public class Collections {
    ...
    @SafeVarargs
    public static <T> boolean addAll(Collection<? super T> c,
            T... elements) {
        boolean result = false;
        for (T element : elements)
            result |= c.add(element);
        return result;
    }
    ...
```

# @Native (1)

- Indicates that a field defining a constant value may be referenced from native code.
  - Can be used, for example, to generate C++ header files.
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Native.html

## @Native (2)

- Example:

```
// Integer.java (OpenJDK 11):
package java.lang;

public final class Integer extends Number implements
        Comparable<Integer> {

    /**
     * A constant holding the minimum value an {@code int} can
     * have, -2<sup>31</sup>.
     */
    @Native public static final int MIN_VALUE = 0x80000000;
    ...
```

## Meta-annotations (1)

- Annotations that apply to other annotations are known as meta-annotations.
- The following meta-annotation types are provided by the `java.lang.annotation` package:
  - `@Documented`
  - `@Inherited`
  - `@Repeatable`
  - `@Retention`
  - `@Target`

## Meta-annotations (2)

- **`@Documented`**:
  - Indicates that the use of the marked annotation must be included in the API documentation (by default, annotations are not included in the documentation generated by the `javadoc` tool).
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Documented.html
- **`@Inherited`**:
  - Indicates that an annotation type is automatically inherited (by default, there is no inheritance).
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Inherited.html

## Meta-annotations (3)

- **`@Repeatable`**:
  - Introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use (see later).
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Repeatable.html
- **`@Retention`**:
  - Specifies how the marked annotation is stored, the following options are available:
    - **`RetentionPolicy.SOURCE`**: the marked annotation is ignored by the compiler.
    - **`RetentionPolicy.CLASS`**: the compiler records the marked annotation in the bytecode, but it is not available at runtime.
    - **`RetentionPolicy.RUNTIME`**: the compiler records the marked annotation in the bytecode, and it is available at runtime.
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Retention.html

# Meta-annotations (4)

- **`@Target`**:
  - Determines the contexts in which an annotation type is applicable, the following options are available:
    - Annotation type declaration (`ElementType.ANNOTATION_TYPE`)
    - Constructor declaration (`ElementType.CONSTRUCTOR`)
    - Field declaration (includes `enum` constants) (`ElementType.FIELD`)
    - Local variable declaration (`ElementType.LOCAL_VARIABLE`)
    - Method declaration (`ElementType.METHOD`)
    - Module declaration (`ElementType.MODULE`)
    - Package declaration (`ElementType.PACKAGE`)
    - Formal parameter declaration (`ElementType.PARAMETER`)
    - Class, interface (including annotation type), or `enum` declaration (`ElementType.TYPE`)
    - Type parameter declaration (`ElementType.TYPE_PARAMETER`)
    - Use of a type (`ElementType.TYPE_USE`)
  - See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Target.html

## Declaring Annotation Types (1)

- A new annotation type is declared using the following annotation type declaration:
  - *modifiers* `@interface` *name* `{` *declarations* `}`
- The above declaration defines a special interface.
  - Not all of the rules that apply to normal interface declarations apply to annotation type declarations.
    - For example, an annotation type declaration cannot be generic, and no `extends` clause is permitted.
    - The direct superinterface of every annotation type is `java.lang.annotation.Annotation` that is itself a normal interface.

# Declaring Annotation Types (2)

- The body of an annotation type declaration may contain any of the following:
  - Class declarations
  - Interface declarations (including annotation type declarations)
  - Constant declarations, such as:
    - `int MIN = 0;`
    - `int MAX = 10;`
  - Special method declarations

# Declaring Annotation Types (3)

- Each method declaration in the body of an annotation type declaration declares an element.
  - Such a method declaration cannot have formal parameters, type parameters, or a `throws` clause.
  - The return type of the method defines the element type, it must be one of the following, or a compile-time error occurs:
    - A primitive type
    - `String`
    - `Class`/`Class<`$T_1$`,...,`$T_n$`>`
    - An `enum` type
    - An annotation type
    - An array type whose component type is one of the preceding types
  - The keyword `default` can be used to specify a default value for an element.
  - By convention, the name of the sole element in a single-element annotation type is `value`.

# Declaring and Using an Annotation Type – Example 1

```java
// Evolving.java:
@Documented
public @interface Evolving {
}

// Experimental.java:
@Documented
public @interface Experimental {
}

// Stable.java:
@Documented
public @interface Stable {
}
```

```java
// Foo.java:
public class Foo {

    @Experimental
    public void a() {
    }

    @Evolving
    public void b() {
    }

    @Stable
    public void c() {
    }

    public void d() {
    }

}
```

## Declaring and Using an Annotation Type – Example 2

```java
// Stability.java:
@Documented
public @interface Stability {
    public enum Status {
        EXPERIMENTAL,
        EVOLVING,
        STABLE
    }
    Status value();
}
```

# Declaring and Using an Annotation Type – Example 2 (continued)

```
// Foo.java:
public class Foo {

    @Stability(Stability.Status.EXPERIMENTAL)
    public void a() {
    }

    @Stability(value=Stability.Status.EVOLVING)
    public void b() {
    }

    @Stability(Stability.Status.STABLE)
    public void c() {
    }

    public void d() {
    }

}
```

# Declaring and Using an Annotation Type – Example 3

- The annotation is only applicable to method and constructor declarations:

```
// Stability.java:
@Documented
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface Stability {
    public enum Status {
        EXPERIMENTAL,
        EVOLVING,
        STABLE
    }
    Status value();
}
```

# Declaring and Using an Annotation Type – Example 4

- The compiler stores the annotation in the bytecode, and thus it can be can accessed at runtime:

```
// Stability.java:
@Documented
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stability {
    public enum Status {
        EXPERIMENTAL,
        EVOLVING,
        STABLE
    }
    Status value();
}
```

# Declaring and Using an Annotation Type – Example 4 (continued)

- Methods declared in the `Foo` class that are marked with the `@Stability` annotation:

```
Arrays.stream(Foo.class.getDeclaredMethods())
    .filter(method -> method.isAnnotationPresent(Stability.class))
    .forEach(System.out::println);
// public void Foo.b()
// public void Foo.c()
// public void Foo.a()
```

# Declaring and Using an Annotation Type – Example 4

- Methods declared in the `Foo` class that are marked with the `@Stability(Stability.Status.STABLE)` annotation:

```
Arrays.stream(Foo.class.getDeclaredMethods())
    .filter(method -> method.isAnnotationPresent(Stability.class)
        && method.getAnnotation(Stability.class).value() ==
            Stability.Status.STABLE)
    .forEach(method -> System.out.println(method + " is STABLE"));
// public void Foo.c() is STABLE
```

# Declaring and Using an Annotation Type – Example 4 (continued)

```
// StabilityUtil.java:
public class StabilityUtil {

    public static Method[] getMethodsWithStability(Class c,
            Stability.Status status) {
        return Arrays.stream(c.getDeclaredMethods())
            .filter(method -> method.isAnnotationPresent(Stability.class)
                && method.getAnnotation(Stability.class).value() == status)
            .toArray(Method[]::new);
    }
    ...
}
```

```
for (Method method : getMethodsWithStability(Foo.class,
        Stability.Status.STABLE)) {
    System.out.println(method + " is STABLE");
}
// public void Foo.c() is STABLE
```

# Declaring and Using an Annotation Type – Example 5

```
// Todo.java:
@Documented
public @interface Todo {
    public enum Priority {
        LOW,
        NORMAL,
        HIGH;
    }
    Priority priority();
    String assignedTo() default "";
}
```

## Declaring and Using an Annotation Type – Example 5 (continued)

```
// Foo.java:
public class Foo {

    @Todo(priority = Todo.Priority.NORMAL)
    public void a() {
        // ...
    }

    public void b() {
        // ...
    }

    @Todo(priority = Todo.Priority.HIGH,
        assignedTo = "me")
    public void c() {
        // ...
    }

}
```

## Declaring and Using an Annotation Type – Example 6

```
// Pattern.java:
@Documented
public @interface Pattern {
    String regex();
    int flags() default 0;
    String message();
}
```

```
// Book.java:
public class Book {

    @Pattern(regex = "^\\d{13}$", message = "Invalid ISBN number")
    String isbn;
    // ...

}
```

## Repeatable Annotations (1)

- Repeatable annotation types were introduced in Java SE 8.
- Multiple annotations of a repeatable annotation type can applied to the same program construct.
  - It is a compile-time error if multiple annotations of a non-repeatable annotation type are applied to the same program construct.
- They require a containing annotation type.

# Repeatable Annotations (2)

```
// Schedule.java:
@Documented
@Target(ElementType.METHOD)
@Repeatable(Schedules.class)
public @interface Schedule {
    String month() default "*";
    String dayOfMonth() default "*";
    int hour() default 12;
    int minute() default 0;
}
```

```
// Schedules.java:
@Documented
@Target(ElementType.METHOD)
public @interface Schedules {
    Schedule[] value();
}
```

# Repeatable Annotations (3)

```
// Foo.java:
public class Foo {

    @Schedule(dayOfMonth = "last", hour = 23, minute = 59)
    public periodicActivity1() {
        // ...
    }

    @Schedule(dayOfMonth = "first", hour = 8)
    @Schedule(dayOfMonth = "last", hour = 16)
    public periodicActivity2() {
        // ...
    }

    @Schedule(month = "Apr", dayOfMonth = "29")
    @Schedule(month = "Jun", dayOfMonth = "29")
    public periodicActivity3() {
        // ...
    }

}
```

## Type Annotations (1)

- A type annotation is an annotation that applies to a type (or any part of a type).
  - It was introduced in Java SE 8.

# Type Annotations (2)

- Declaring and using a type annotation:

```
// NonNull.java:
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_USE)
public @interface NonNull {
}
```

## Type Annotations (3)

- Declaring and using a type annotation: (continued)
  - `@NonNull String s = getString();`
  - `String s = (@NonNull String) o;`
  - `@NonNull String processString(@NonNull String s) { ... }`

  -
    ```
    void processList(@NonNull List<@NonNull Object> list) {
      ...
    }
    ```
  - `<T> void processArray(@NonNull T[] arr) { ... }`

  -
    ```
    <T> void processArray(@NonNull T @NonNull [] arr) {
      ...
    }
    ```
  - `@NonNull var x = getData();`
  - `(@NonNull var x, @NonNull var y) -> x.process(y)`

## Type Annotations (4)

- Example: *The Checker Framework* (license: GPLv2) https://checkerframework.org/
  - Checker: a tool that warns about certain errors or gives a guarantee that those errors do not occur.
    - The verification happens at compile time.
  - Can be used in the Eclipse and IntelliJ IDEA IDEs, and also with the Gradle and Apache Maven build systems.
  - Requires JDK 8 or JDK 11.

# Type Annotations (5)

- Example: *The Checker Framework*: (continued)
  - Code snippet and command line usage (JDK 11):

```
List<@NonNull String> list = new ArrayList<>();
list.add(null);
```

```
$ javac -J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED \
  -processorpath /path/to/checker.jar \
  -cp /path/to/checker-qual.jar \
  -processor org.checkerframework.checker.nullness.NullnessChecker \
  Foo.java Bar.java
Foo.java:10: error: [argument.type.incompatible] incompatible types in
argument.
    list.add(null);
             ^
  found   : null
  required: @Initialized @NonNull String
1 error
```

# The javax.annotation.processing Package (1)

- Provides a means for run-time processing of annotations.
  - It was introduced in Java SE 6.
  - See: *JSR 269: Pluggable Annotation Processing* https://jcp.org/en/jsr/detail?id=269
- The `AbstractProcessor` class of the package is provided for annotation processing.
  - See: `javax.annotation.processing.AbstractProcessor` https://docs.oracle.com/en/java/javase/11/docs/api/java.compiler/javax/annotation/processing/AbstractProcessor.html

## The javax.annotation.processing Package (2)

- Example for annotation processing:

```
// StabilityProcessor.java:
@SupportedAnnotationTypes("Stability")
public class StabilityProcessor extends AbstractProcessor {

    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }

    public boolean process(Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnv) {
        for (Element element :
                roundEnv.getElementsAnnotatedWith(Stability.class)) {
            Stability stability = element.getAnnotation(Stability.class);
            final String message = String.format("%s is %s", element,
                stability.value());
            processingEnv.getMessager().printMessage(Kind.NOTE, message);
        }
        return false;
    }

}
```

## The javax.annotation.processing Package (3)

- Example for annotation processing: (continued)
  - Command line usage:

```
$ javac StabilityProcessor.java
$ javac -processor StabilityProcessor Foo.java
Note: a() is EXPERIMENTAL
Note: b() is EVOLVING
Note: c() is STABLE
```

# Further Recommended Reading

- *The Java Tutorials – Trail: Learning the Java Language – Lesson: Annotations*. https://docs.oracle.com/javase/tutorial/java/annotations/
- Joshua Bloch. *Effective Java*. Third Edition. Addison-Wesley Professional, 2017. http://www.informit.com/store/effective-java-9780134685991