Annotations in Java

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

Kocsis, Gergely (English version)
University of Debrecen, Faculty of Informatics
kocsis.gergely@inf.unideb.hu

Last modified: 13.02.2018

Definition

- Metadata related to a program-construction, that has no direct effect on the execution of the program

History (1)

- Annotations were first announced in J2SE 5 in 2004
  - See:
    - *New Features and Enhancements J2SE* 5.0 http://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
- Java SE 6 in 2006 provides more addditional features (`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 in 2014 adds even more new features (type annotations, repeatable annotations, new pre-defined annotation types)
  - See: *What's New in JDK 8* http://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

Possible uses

- **Providing information for the compiler**:
  - e.g. suppress some given warnings or show some given errors
  - The Checker Framework http://checkerframework.org/
- **Code generation**: code can be generated based on annotations
  - e.g. JAXB can handle XML files based on annotations
- **Runtime processing**: some annotations can be reached during runtime
  - JUnit Unit Test framework http://junit.org/
  - Bean Validation: a part of Java EE 6 (see `javax.validation` package and subpackages)
    - *JSR 349: Bean Validation 1.1 (Final Release)* (24 May 2013) https://jcp.org/en/jsr/detail?id=349
    - Reference implementation: *Hibernate Validator* http://hibernate.org/validator/

Equivalent tools of other languages

- **.NET**: attributes
  - *.NET Framework Development Guide – Extending Metadata Using Attributes* https://msdn.microsoft.com/en-us/library/5x6cd29c(v=vs.110).aspx
- **Python**: variable and function annotations (since v3.0)
  - *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 9 Edition*. 7 August 2017. https://docs.oracle.com/javase/specs/jls/se9/html/
  - See the following sections:
    - 9.6. *Annotation Types* https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6
    - 9.7. *Annotations* https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7

Structure of annotations

- It builds up from the following:
  - Name of an annotation type
  - An optional list that consists of element-value pairs separated by commas
    - The list is given between parentheses → ( )
- The annotation type matching to the name determines the possible element-value pairs
  - If an element-value pair has a default value, it is not necessary be given
- The order of element-value pairs is not restricted

Forms of Annotations

- **Simple annotation:**
  - `@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**: if no element-value pairs are given the () characters are optional
  - `@NotNull`, `@NotNull()`

Structure of annotations (2)

- If an element is an array type, the value is to be given by the initialization expression.
  - Not counting the case of one-element arrays. In this case {} can be left out.
- E.g. these two annotations are equivalent:
  - `@Target({ElementType.METHOD})`
  - `@Target(ElementType.METHOD)`

Where can annotations be used?

- For declarations:
  - Annotation type, constructor, class variable, `enum` constant, local variable, method, package, formal parameter, class, interface, declaration of `enum` and type parameter (Java SE 8)
- For the use of Types (Java SE 8)

Predefined annotations

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

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

@Deprecated

- Elements marked with this annotations are better to be avoided usually because it is not safe or because there exists a better alternative
  - It is advised to also use `@Deprecated` Javadoc label to document the obsolescence of the element
- Compilers warn the use of elements marked with this annotation
- A list of deprecated elements in Java SE 8: http://docs.oracle.com/javase/8/docs/api/deprecated-list.html

@Deprecated

```
// Character.java (JDK 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 introduces two new optional elements:
  - `since`: to note from which version the element is deprecated (default value: `""`)
  - `forRemoval`: to note that the element will be removed in the future (default value: `false`)
- See: http://download.java.net/java/jdk9/docs/api/java/lang/Deprecated.html

@Deprecated (4)

- Example:

```
// Applet.java (JDK 9):
package java.applet;

import java.awt.*;

@Deprecated(since = "9")
public class Applet extends Panel {
    ...
```

@Deprecated (5)

- Java SE 9:
  - A static analysis tool that scans deprecated JDK API elements. It can e used in command line. (`jdeprscan`).
    - Example:
      - `jdeprscan commons-io-2.6.jar`
      - `jdeprscan lib/*.jar`
    - See: https://docs.oracle.com/javase/9/tools/jdeprscan.htm
  - Importing deprecated types or statically importing deprecated members do not result in warning.
    - See: *JEP 211: Elide Deprecation Warnings on Import Statements* http://openjdk.java.net/jeps/211

@SuppressWarnings

- It marks for the compiler that warnings in this element (and also in sub-elements) are to be suppressed.

@SuppressWarnings

```
@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, 6);
}
```

@Override

- It marks that the marked method is an overridden method of the original declared in the superclass
- It is not mandatory to add however it helps avoiding errors

@Override

```
// Integer.java (JDK 8):
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

- Marks that an interface is functional
  - Functional interfaces has exactly one explicitly declared abstract method

```
// FileFilter.java (JDK 8):
package java.io;

@FunctionalInterface
public interface FileFilter {

   boolean accept(File pathname);

}
```

@SafeVarargs

- It clears warnings when using functions with variable number of arguments

```
// Collections.java (JDK 8):
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

- It marks that the marked class variable defines a constant that can also be referred from native code.
  - Can be used e.g. to produce C++ header files

```
// Integer.java (JDK 8):
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 can be used for annotation types. They are defined in `java.lang.annotation` package:
  - `@Documented`
  - `@Inherited`
  - `@Repeatable`
  - `@Retention`
  - `@Target`

Meta-annotations (2)

- **`@Documented`**:
  - It marks that the use of this annotation also has to appear in API documentation. (By default annotations do not appear in the documentation generated by Javadoc)
- **`@Inherited`**:
  - It marks that the given annotation type is automatically inherited (Inheritance is not default).

Meta-annotations (3)

- **@Repeatable**:
  - Introduced in Java SE 8. It marks that the annotation can be used multiple times for the same declaration or type.
- **@Retention**:
  - Defines the way of storing the annotation. Options are listed below:
    - **`RetentionPolicy.SOURCE`**: The compiler neglects the annotation.
    - **`RetentionPolicy.CLASS`**: The compiler stores the annotation in the byte code, but it can not be reached in runtime.
    - **`RetentionPolicy.RUNTIME`**: The compiler stores the annotation in the byte code, and it can be reached in runtime.

Meta-annotations (4)

- **`@Target`**:
  - Defines that the for what elements the annotation can be used. Possible options are:
    - Definition of annotation type (`ElementType. ANNOTATION_TYPE`)
    - Constructor declaration (`ElementType.CONSTRUCTOR`)
    - Class variable, `enum` constant declaration (`ElementType.FIELD`)
    - Declaration of local variable (`ElementType.LOCAL_VARIABLE`)
    - Declaration of method (`ElementType.METHOD`)
    - Declaration of Module (`ElementType.MODULE`)
    - Package declaration (`ElementType.PACKAGE`)
    - Declaration of formal parameter (`ElementType.PARAMETER`)
    - Class, interface or enum declaration (`ElementType.TYPE`)
    - Declaration of type parameter (`ElementType.TYPE_PARAMETER`)
    - Use of type (`ElementType.TYPE_USE`)

Declaring annotation type (1)

- A new annotation type can be declared in the following way:
  - *modifiers* `@interface` *name* `{` *declarations* `}`
- @ (*AT*) is for Annotation Type
- This declaration describes a special interface
  - Not all of the simple interface rules hold for annotation type declarations
    - Can not be generic, and can not be parent interface
    - Super interface of all annotation types is `java.lang.annotation.Annotation`, that is a ordinary interface

Declaring annotation type (2)

- In the body the following declarations are allowed:
  - Class declaration
  - Interface declaration
  - Constant declaration, e.g.:
    - `int MIN = 0;`
    - `int MAX = 10;`
  - Special method declaration

Declaring annotation type (3)

- Method declarations in the body of the annotation type are for respective element declarations
  - Formal parameters, type parameters and the `throws` keyword are not allowed
  - The return type declares the type of the element. Options are:
    - Primitive type
    - `String`
    - `Class`/`Class<`*T*`>`
    - `enum` type
    - Annotation type
    - An array of elements of one of the above types
  - Default value can be given with the `default` keyword
  - In single element annotations the name “`value`” is conventionally used for the element

Declaring and use of annotation types – 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 use of annotation types – example 2. (1)

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

Declaring and use of annotation types – example 2. (2)

```java
// 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 use of annotation types – example 3.

- The annotation can only be used on methods and constructors:

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

Declaring and use of annotation types – example 4. (1)

- The compiler stores the annotation in the byte code and makes it reachable in 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 use of annotation types – example 4. (2)

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

```
for (Method method : Foo.class.getDeclaredMethods()) {
    if (method.isAnnotationPresent(Stability.class)
        && method.getAnnotation(Stability.class).value()
            == Stability.Status.STABLE) {
            System.out.printf("%s is STABLE\n", method);
    }
}

// Output:
public void Foo.c() is STABLE
```

Declaring and use of annotation types – example 4. (3)

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

    public static Method[] getAnnotatedMethods(Class c,
            Class<? extends Annotation> a) {
        ArrayList<Method> methods = new ArrayList<Method>();
        for (Method method : c.getDeclaredMethods()) {
            if (method.isAnnotationPresent(a))
                methods.add(method);
        }
        return methods.toArray(new Method[0]);
    }
    ...
```

```
for (Method method : ReflectUtil.getAnnotatedMethods(Foo.class,
        Stability.class)) {
    System.out.println(method);
}

// public void Foo.a()
// public void Foo.b()
// public void Foo.c()
```

Declaring and use of annotation types – example 4. (4)

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

    public static Method[] getMethodsWithStability(Class c,
            Stability.Status status) {
        ArrayList<Method> methods = new ArrayList<Method>();
        for (Method method : c.getDeclaredMethods()) {
            if (method.isAnnotationPresent(Stability.class)) {
                if (method.getAnnotation(Stability.class).value()
                    == status) methods.add(method);
            } else if (status == null) methods.add(method);
        }
        return methods.toArray(new Method[0]);
    }
    ...
```

```
for (Method method : getMethodsWithStability(Foo.class,
        Stability.Status.STABLE))
    System.out.printf("%s is STABLE\n", method);

// Output:
public void Foo.c() is STABLE
```

Declaring and use of annotation types – example 5. (1)

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

Declaring and use of annotation types – example 5. (2)

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

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

    public void b() {
    }

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

}
```

Declaring and use of annotation types – example 6.

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

```
// Pattern.java:
public class Kamion {

    @Pattern(message = "Érvénytelen forgalmi rendszám",
        regex = "^F[I-Z][A-Z]-\\d{3}$")
    String rendszám;
    ...

}
```

Repeatable annotations (1)

- Multiple use of the same annotation for a given part of the same program code (Java SE 8)
  - A containing annotation type is required

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)

- The use of annotation types on types or on given parts of types (Java SE 8)

Type annotations (2)

- Declaration and use of type annotations:

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

Type annotations (3)

- Declaration and use of type annotations (continue):
  - `@NonNull String s = getString();`
  - `String s = (@NonNull String) o;`

  -
    ```
    @NonNull String processString(@NonNullString s) {
       ...
    }
    ```

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

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

Type annotations (4)

- Example: *The Checker Framework* http://types.cs.washington.edu/checker-framework/
  - Checker: a tool that warns for given errors or ensures that the error will not appear
    - Checking is done in runtime
  - It can be used in Eclipse IDE and in command line

Type annotations (5)

- Example: *The Checker Framework* (continue):
  - Use in command line:

```
$ javac -cp /path/to/checker.jar:/path/to/javac.jar \
   -Xbootclasspath/p:/path/to/jdk8.jar -processor \
   org.checkerframework.checker.nullness.NullnessChecker \
   Foo.java Bar.java
Foo.java:8: 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)

- It makes possible to process annotations in compilation time
  - Introduced in Java SE 6
  - See more: *JSR 269: Pluggable Annotation Processing* https://jcp.org/en/jsr/detail?id=269
- The `AbstractProcessor` class of the package makes it possible to process annotations

The javax.annotation.processing package (2)

- Annotation processing example:

```
// 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)

- Annotation processing example:
  - Use in command line:

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