Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

Annotations

Object Oriented Programming

Tiziano Leidi

15.10.2021

Preamble: unit testing framework (part 2)

We want to extend our unit testing framework to remove the dependency on the method name.

In the previous approach, test methods are public methods with the name starting with “test”.

As an alternative, we would like to have a solution that could be used for every method, independently from the method name.

Which approach do you suggest to use?

Expected behaviour (new version)

We want that the framework extracts all the public methods with a @ch.supsi.oop.unit.Test annotation (*convention over configuration*).

Class to test

```
public class MyMath {
    public static <T extends Number> Integer sum(T a, T b) {
        return a.intValue() + b.intValue();
    }
}
```

Performed tests

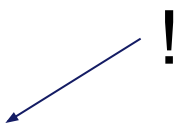
```
public class MathTestAnnotations {
    @ch.supsi.oop.unit.Test("Test the sum method")
    public void test21() {
        Integer sum = Math.sum(40.6, 50.2);
        UnitTest.assertSame(90, sum);
    }

    public void test22() {
        Integer sum = Math.sum(40.6, 50.2);
        UnitTest.assertSame(100, sum);
    }

     //...
```

Performed tests

```
    // ...

    @ch.supsi.oop.unit.Test("Test 2")
    public void test23() {
        Integer sum = Math.sum(1, 4);
        UnitTest.assertSame(5, sum);
        UnitTest.assertSame(4, sum);
    }

    @ch.supsi.oop.unit.Test("Test 3")
    public void test24() {
        throw new RuntimeException("Connection refused");
    }
}
```

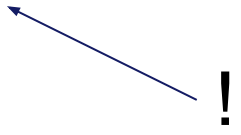
UnitTestExecutor: internalExecute method

```
private static void internalExecute(String cls) throws ClassNotFoundException,
 IllegalAccessException, IllegalArgumentException, InvocationTargetException,
  InstantiationException {
    Class<?> testClass = Class.forName(cls);
    Object testObject = testClass.newInstance();

    List<TestResult> results = new ArrayList<TestResult>();
    for (Method method : testClass.getDeclaredMethods()) {
        boolean success = true;
        boolean failure = false;
        boolean error = false;
        String message = "";

        Test annotation = method.getAnnotation(Test.class);
        if (annotation != null) {
            try {
                method.invoke(testObject);
            } catch (Exception e) {

            // ...
```



UnitTestExecutor: internalExecute method

```
                // ...

                message = a.getCause().getMessage();
                success = false;
                if (e.getCause() instanceof AssertionError)
                    failure = true;
                else
                    error = true;
            }

            TestResult testResult = new TestResult(method.getName(),
                        (annotation != null) ? annotation.value() : "",
                         success, failure, error, message);
            results.add(testResult);
        }
    }
}
```



Annotations

- An annotation is a form of syntactic **metadata** that can be added to Java source code. Classes, methods, variables, parameters and Java packages may be annotated.
- Java annotations can be read from Java class files generated by the Java compiler. This allows annotations to be retained by the Java virtual machine at run-time and read via reflection.

Annotations

Annotations are are used to provide:

- Information for the compiler (and the IDE), for example for error detection (e.g. @Override, @FunctionalInterface) or to suppress warnings (e.g. @SuppressWarnings)
- Compile-time and deployment-time processing
- Runtime processing

Annotations have no direct effect on the annotated code.

Annotations are only markers for meta programming.

Annotations

- In their simplest form, annotations look like:

```
@MyAnnotation
```

… without parentheses, if there are no parameters.

- If there are parameters, the syntax is instead:

```
@MyAnnotation(“hello”)
```

Example: built in annotations

```
public class Vehicle {
    public void go() {
        //...
    }
}

//...

public class Car extends Vehicle
{
    @Override
    public void go(int speed) {
        //...
    }
}
```

What happens here?

Example: built in annotations

```
public class Vehicle {
    public void go() {
        //...
    }
}

//...

public class Car extends Vehicle
{
    @Override
    public void go(int speed) {
        //...
    }
}
```

Compilation error: method does not override or implement a method from a supertype

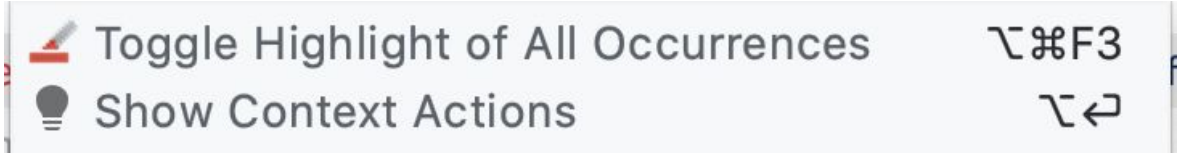
Example: built in annotations

```
@SuppressWarnings({"rawtypes", "unchecked"})
void aMethod() {
   // warning: raw use of parameterized class 'HashMap'
   HashMap map = new HashMap();

   // warning: unchecked call to 'put(K, V)' as a member
               of raw type 'java.util.HashMap'
   map.put("abc", "def");
}

@Deprecated
public void oldMethod() {
   // this is deprecated
}
```

@SuppressWarnings

- Valid arguments depend on the IDE and compiler.
- In Intellij IDEA:
  - invoke the “Show Context Actions”
  - suppress the warning(s) related to code inspection (e.g. current method)

  to automatically introduce the @SuppressWarning annotation.



How to develop your own annotation

An annotation can be declared with a special interface, by using the *@interface* keyword:

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {
    String dueDate();
    String[] task() default "";
}
```

@Target

- @*Target* is used to provide the context in which the annotation can be applied. Available values are provided by the enumeration:

```
java.lang.annotation.ElementType
```

- It is possible to have annotations on annotations, by using the `ElementType.ANNOTATION_TYPE.`
- When @*Target* is omitted, the declared type can be used on **any program element**.
- **Important:** more than one element type can be specified for the same @*Target*.

@Target

```java
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {
    String dueDate();
    String[] task() default "";
}
```

@Target - declaration annotations

```
ElementType.TYPE class, interface, or enum declaration

ElementType.FIELD

ElementType.METHOD

ElementType.PARAMETER

ElementType.CONSTRUCTOR

ElementType.LOCAL_VARIABLE

ElementType.ANNOTATION_TYPE

ElementType.PACKAGE

ElementType.MODULE                  // since Java 9
```

@Target - type annotations

```
ElementType.TYPE_PARAMETER      // since Java 8

ElementType.TYPE_USE            // since Java 8
```

Type annotations

Before the Java 8 release, annotations could only be applied to declarations. As of the Java 8 release, annotations can also be applied to any use of a type: instance creation expressions (new), casts, implements clauses, and throws clauses.

```
@NonNull String str;

myString = (@NonNull String) str;

new @Interned MyObject();
```

Type annotations

Declaration annotation says something, for example about a method or use of a field. Instead, type annotation provide information about the value: for example, with @Positive the field is not allowed to have a value below zero.

Declaration annotations are written on their own line. Type annotations are written directly before the type, on the same line.

Type annotations

- The ElementType.TYPE_USE target indicates that the annotation can be written on any use of a type.
- The ElementType.TYPE_PARAMETER target indicates that the annotation can be applied at type variables (e.g. T in MyClass<T>).

@Retention

```java
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {
    String dueDate();
    String[] task() default "";
}
```

@Retention

@Retention specifies how long the annotation survives:

- RetentionPolicy.SOURCE: annotations are used and discarded by the compiler.
- RetentionPolicy.CLASS **(default policy):** annotations are preserved in the .class file but ignored by the JVM, discarded during class load. Can be used by special libraries, which operate directly on the byte-code.
- RetentionPolicy.RUNTIME: annotations are preserved in the .class and can be accessed at runtime using the Reflection API.

Example - @SuppressWarnings

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
   String[] value();
}
```

RetentionPolicy Example

@Override has RetentionPolicy.SOURCE

- It informs **the IDE and compiler** that the element is meant to override another element declared in a superclass.
- The IDE and compiler use this annotation to check if there are any errors.

Element Declarations

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {
    String dueDate();
    String[] task() default "";
}
```

Element Declarations

Element declarations look like methods, but are more a form of annotation parameter. It is possible to define default values.

The return type of an annotation element **must** be one of the followings:

- A primitive type (int, short, long, byte, char, double, float, or boolean)
- *String*
- *Class* (with an optional type parameter such as Class<? extends MyClass>)
- An enum type
- An annotation type
- An array type whose component type is one of the preceding types

Example: @Todo annotation

```
public class Example {
    @Todo(dueDate = "01/01/2021", task = "refactor this method")
    public static void main(String[] args) {
        // ...
    }
}
```

Example: @Todo annotation

We have to write a class to extract the annotation metadata using reflection. The reflection classes Class, Field, Method, etc., provide methods to obtain information about the annotations.

For example, the *Method* class provides *isAnnotationPresent()* and *getAnnotation().*

Example: @Todo annotation

```
public class Analyser {
   public static void main(String[] args) {
       Class myClass;
       try {
           myClass = Class.forName("ch.supsi.annotations.Example");
       } catch(java.lang.ClassNotFoundException classNotFoundException) {
           System.out.println("please specify an existing class name");
           return;
       }
       Method[] methods = myClass.getDeclaredMethods();
       for(Method method : methods) {
           if(method.isAnnotationPresent(Todo.class)) {
               Todo todo = method.getAnnotation(Todo.class);
               System.out.println("Annotation elements: " + todo.task() + " - " +
                   todo.dueDate());
               // Result: “Annotation elements: refactor this method - 01/01/2021”
           }
       }
   }
}
```

Meta annotations

Meta annotations are annotation on annotations:

- @Documented: the affected annotation will be included in the JavaDoc documentation.
- @Inherited: used to mark an annotation to be inherited by subclasses of the annotated class.
- @Repeatable: the annotation can be applied multiple times to the same declaration. Repeatable annotation are stored in an annotation container for compatibility reasons.

Example: @Repeatable

Trying to apply the same annotation without first declaring it to be repeatable results in a compile-time error.

The following modification is needed:

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(TodoContainer.class)
public @interface Todo {
   String dueDate();
   String task() default "";
}
```

Example: @Repeatable

```
public class Main {
    @Todo(dueDate = "01/01/2021", task = "refactor this method")
    @Todo(dueDate = "01/02/2021", task = "add jsdoc comments")
    public static void main(String[] args) {
        // ...
    }
}
```

Example: @Repeatable

We need to provide a containing annotation (TodoContainer in our example).

We don’t have to explicitly use the container annotation, because it is **automatically exploited** by the Java compiler.

The containing annotation **MUST HAVE** a value element with the correct array type.

Example: @Repeatable

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TodoContainer {
    Todo[] value();
}
```

IMPORTANT:

@Target must be a subset of the contained annotation Target (in this example {METHOD} is a subset of {METHOD}).

@Retention must not be shorter than the contained annotation retention policy.

… otherwise it won't compile ...

Example: @Repeatable

To access the information in the container, the method *isAnnotationPresent()* using TodoContainer.class as a parameter has to be used.

The contained annotations have to be extracted with *getAnnotationsByType()*, which returns an array of annotations.

Example: @Repeatable

```
public class Analyser {
   public static void main(String[] args) {
       Class myClass;
       //... same code from the previous example
       Method[] methods = myClass.getDeclaredMethods();
       for (Method method : methods) {
           if (method.isAnnotationPresent(TodoContainer.class)) {
               Todo[] todos = method.getAnnotationsByType(Todo.class);
               System.out.println("todos " + todos.length);
               for (Todo todo : todos) {
                   System.out.println("Annotation elements: " + todo.task() + " - " +
                                      todo.dueDate());
               }
               // Result:
               // Annotation elements: refactor this method - 01/01/2021
               // Annotation elements: add jsdoc comments - 01/02/2021
           }
       }
   }
}
```

Summary

- Introduction to annotations
- Built-in annotations
- Annotation target and retention policy
- Annotation elements and valid return types
- Meta annotations
- Repeatable annotations