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

SUPSI

Reflection

Object Oriented Programming

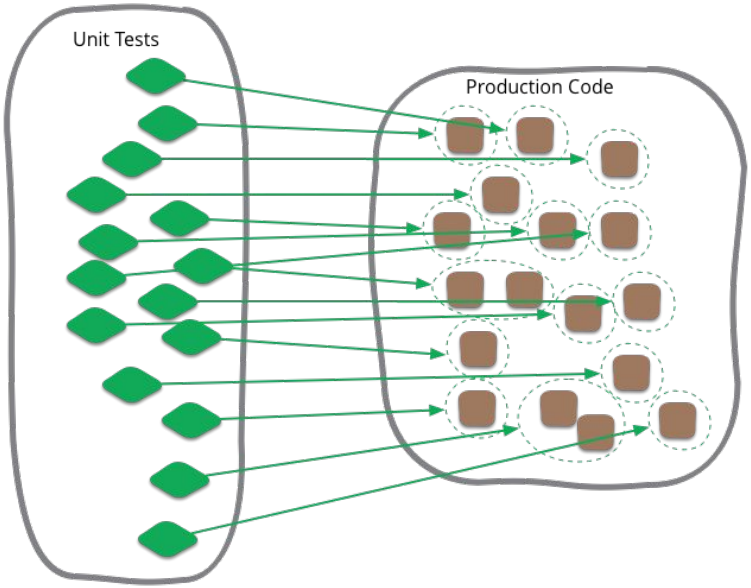
Tiziano Leidi

10/10/2021

Preamble: unit testing framework

We want to perform unit tests on our source code to know if the methods return the expected results (by using equals).

A unit test is a low-level test, focusing on a small part of the software system (Martin Fowler)



We want to write our tests in this form:

```
package ch.supsi.oop.unit.test;
import ch.supsi.oop.unit.UnitTest;

public class MathTest {
    public void testAbs() {
        int abs = java.lang.Math.abs(-128);
        UnitTest.assertEquals(128, abs);
        abs = java.lang.Math.abs(2);
        UnitTest.assertEquals(2, abs);
    }

    private int max(int a, int b) {
        return a;
    }

    //...
```

```
    //...

    public void testMax() {
        int max = max(1, 4);
        UnitTest.assertEquals(4, max);
    }

    public void testFail() {
        throw new RuntimeException("Something went wrong");
    }
}
```

Implementation of the assertion

```
package ch.supsi.oop.unit;

public class UnitTest {
   public static void assertEquals(Object expected, Object actual) {
       if ((expected == null) && (actual == null)) {
           return;
       }
       if ((expected != null) && expected.equals(actual)) {
           return;
       }
       throwError("The object is not equal [expected: <"
          + expected + ">, actual: <" + actual + ">]");
   }

   //...

   private static void throwError(String message) {
       throw new AssertionError(message);
   }
}
```

Expected behaviour

We need that the framework extracts all public methods starting with “test” (*convention over configuration*), then execute the method and check if the assertions are valid.

If there is a failure (wrong assertion) or an error (an exception is triggered), a message has to be written.

Important: a test should pass when **all the assertions** are valid (otherwise it must fail).

Execution

To start the execution, we need a UnitTestExecutor class with a static method *execute*, which gets the name of the class containing the test methods and perform all the specified tests:

```
UnitTestExecutor.execute("ch.supsi.oop.unit.test.MathTest");
```

```java
package ch.supsi.oop.unit;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import ch.supsi.oop.unit.TestResult;

public class UnitTestExecutor {
    public static void execute(String... classes) throws Exception {
        for (String curClass : classes) {
            System.out.println("Unit tests for " + curClass);
            System.out.println();
            internalExecute(curClass);
            System.out.println();
        }
    }


//...
```

```
    //...

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

        List<TestResult> results = new ArrayList<TestResult>();

        for (Method method : testClass.getDeclaredMethods()) {
            if (method.getName().startsWith("test")) {
                boolean success = true;
                boolean failure = false;
                boolean error = false;
                String message = "";

                try {
                    method.invoke(testObject);
                } catch (Exception e) {
                    message = e.getCause().getMessage();
                    success = false;
                    if (e.getCause() instanceof AssertionError)
                        failure = true;
                    else
                        error = true;
                }

                TestResult testResult = new TestResult(method.getName(), method.getName(),
                success, failure, error, message);
                results.add(testResult);
            }
        }
    }
```

Reflection

- Reflection allows a program to introspect upon itself and manipulate internal properties of the program.
- It also allows to invoke methods or instantiate objects (even if the class and its members are unknown at compile-time).
- It’s not required that the source code is available, the information stored by the JVM for classes is used.

Metaprogramming

- Reflection is mainly used to implement testing and debugging tools, as well as automatic code generation tools.

- Reflection is part of the meta programming techniques (like for example macros): metaprogramming is a programming technique in which programs have the ability to treat other programs as their data.

Reflection in Java

- Java provides some specific classes for reflection that can be instantiated to obtain information about the structure of other classes, instantiate objects, call methods, ...

java.lang.Class<T>

- The most important reflection class is Class<T>, because it is the entry point for all the reflection operations.
- Using its methods and additional classes in the *java.lang.reflect* package, it is possible to inspect all the fields and methods exposed by the class itself.
- Class<T> represents a class, interface, a primitive value or void.

Take a look at:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Class.html

Obtaining the Class<T> instance

There are three ways to obtain an instance of Class<T>.

For example, for a class named “MyClass”:

```
1. MyClass.class
```

If an instance of the MyClass is available:

```
2. myInstance.getClass()
```

If only the class name (with package prefix) is available:

```
3. Class.forName(“package.ClassName”)
```

How to obtain a Class<T> instance: .class

- From the class itself, using the static .class property:

```
Class<String> myClass = String.class;
```

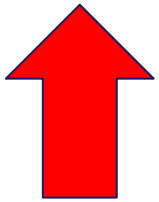
To avoid an unchecked conversion warning*

* see https://docs.oracle.com/javase/tutorial/reflect/class/classTrouble.html

How to obtain a Class<T> instance: getClass

- From an object, using the getClass() method (available on the Object class):

```
Class<? extends String> myClass = "hello".getClass();
```



Why?

Example: class comparison

```
public class GettingTheClass {
    public static void main(String[] args) {
        Person person0 = new Person();
        Person person1 = new Student();
        Class<? extends Person> class0 = person0.getClass();
        Class<? extends Person> class1 = person1.getClass();
        System.out.println(class0.equals(class1));
    }
}
```

... prints false on the console

Example: polymorphism

```
Set<String> set = new HashSet<String>();
Class<? extends Set> setClass = set.getClass();
System.out.println("class of the set is " + setClass);
```

... prints java.util.HashSet on the console

Example: array of bytes

```
byte[] bytes = new byte[1024];
Class<?> bytesClass = bytes.getClass();
System.out.println(bytesClass);
```

... prints the encoding corresponding to an array of bytes: [B

Example: enums

```
public enum Seasons {
    FALL, WINTER, SPRING, SUMMER
}

//...

public class Enums {
    public static void main(String[] args) {
        System.out.println(Seasons.SUMMER.getClass());
    }
}
```

… prints the corresponding enumeration: Season

Why?

How to obtain a Class<T> instance: .forName

- Using the static forName method, with the complete class name as a String (throws ClassNotFoundException):

```
Class<?> clazz = Class.forName("java.lang.String");
```

Use the fully qualified name here

- Useful when the class name is not known in advance.

Examples

The following calls are valid:

```
Class.forName("[D");                        // array of double

Class.forName("[[Ljava.lang.String;"); // a 2D array of String
```

Type name encodings

element type	encoding
boolean	Z
byte	B
char	C
class or interface	L*classname*;
double	D
float	F
int	I
long	J
short	S

Example: primitive types

```
int i = 42;
i.getClass(); // won't work
int.class;    // will return the Class of the primitive type

// special case: primitive type wrappers have a TYPE field
int.class.equals(Integer.TYPE); // true
```

Compatibility of results

```
public class TheClass {
    public static void main(String[] args) {
        try {
            Class<String> class0 = String.class;
            Class<? extends String> class1 = "hello".getClass();
            Class<?> class2 = Class.forName("java.lang.String");
            System.out.println(class0.equals(class1)); // true
            System.out.println(class0.equals(class2)); // true
        } catch (ClassNotFoundException classNotFoundException) {
            classNotFoundException.printStackTrace();
        }
    }
}
```

Support for inheritance in Class<T>

- to obtain the extended class, Class<T> provides the getSuperClass() method.
- to obtain the implemented interfaces, the getInterfaces() method can be used.

- due to compile type erasure, getSuperClass() and getInterfaces() don’t provide generic parameters.
- For this reason getGenericSuperClass() and getGenericInterfaces(), with additional support for generic types, are also provided.

Example

```
public class ClassDemo {
    public static void main(String args[]) {

        // returns the superclass
        Type type = IntegerClass.class.getGenericSuperclass();
        System.out.println(type);

        ParameterizedType p = (ParameterizedType) type;
        System.out.println(p.getActualTypeArguments()[0]);
    }
}

class IntegerClass extends ArrayList<Integer> {
}
```

Fields, methods and constructors

- The package java.lang.reflect provides classes with functionality to obtain information of the members contained in a class:

  `java.lang.reflect.Field`

  `java.lang.reflect.Method`

  `java.lang.reflect.Constructor`

java.lang.reflect.Field

- Represents a class field
- Allows to **read and modify** the field value
- Allows to read and modify even **private fields,** using the setAccessible method before the get/set call

```
field.setAccessible(true);
```

- To read the value of the field for the specified object:

```
Object get(Object obj);
```

- To writes the value of the field for the specified object:

```
void set(Object obj, Object value);
```

- Returns the Class object of the declared type:

```
Class<?> getType();
```

How to obtain a Field

Class<T> provides methods that ...

- returns an array of **public** fields, including inherited fields:
  `myClass.getFields();`
- returns an array of public, protected, default and private fields declared by the class, excluding inherited fields:
  `myClass.getDeclaredFields();`
- returns a **public** field with the specified name, including a field that could be in inherited classes:
  `myClass.getField(String name);`
- returns the specified field from the current class:
  `myClass.getDeclaredField(String name);`

getField vs getDeclaredField

- `getField()`

  Provides access to all the public fields in the entire class hierarchy.

- `getDeclaredField()`

  Provides access to all the fields, independently from their accessibility, but only for the current class.

Example

```
public class TargetClass {
    private Integer theAnswer = 42;

    public TargetClass(int theAnswer) {
        this.theAnswer = theAnswer;
    }

    public Integer getTheAnswer() {
        return theAnswer;
    }

    private Integer getTheAnswerInternal() {
        return theAnswer;
    }
}
```

Example

Doesn't work: the field is private

```
import java.lang.reflect.Field;

public class FieldsTest {
    public static void main(String[] args) {
        TargetClass targetObject = new TargetClass();
        System.out.println("the Answer is " + targetObject.getTheAnswer());
        try {
            Field theField = TargetClass.class.getField("theAnswer");
            System.out.println(theField.get(targetObject));
            theField.set(targetObject, -273);
            System.out.println("Now the Answer is "
                                        + targetObject.getTheAnswer());
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Example

```
import java.lang.reflect.Field;

public class FieldsTest {
   public static void main(String[] args) {
       TargetClass targetObject = new TargetClass();
       System.out.println("the Answer is " + targetObject.getTheAnswer());
       try {
           Field theField = TargetClass.class.getDeclaredField("theAnswer");
           theField.setAccessible(true);
           System.out.println(theField.get(targetObject));
           theField.set(targetObject, -273);
           System.out.println("Now the Answer is "
                                              + targetObject.getTheAnswer());
       } catch (NoSuchFieldException | IllegalAccessException e) {
           e.printStackTrace();
       }
   }
}
```

java.lang.reflect.Method

- Represents a class method
- Provides information about the method and allows to examine its arguments and return types
- Allows to invoke the method:

```
Object invoke(Object obj, Object… args);
```

- Private methods can be invoked only after setAccessible(true) has been called on the Method object
- If the method is static, the first argument of the invoke method has to be set to null

How to obtain a Method

Class<T> provides methods that ...

- returns an array of **public** methods, including inherited:

```
myClass.getMethods();
```

- returns an array of methods declared by the current class, excluding the inherited methods:

```
myClass.getDeclaredMethods();
```

- returns a **public** method, also from inherited classes:

```
myClass.getMethod(String name, Class<?>... paramTypes);
```

- returns the specified method from the current class:

```
myClass.getDeclaredMethod(String name, Class<?>...
paramTypes);
```

Example

```
import java.lang.reflect.Method;

public class MethodsTest {
   public static void main(String[] args) {
       TargetClass targetObject = new TargetClass();
       try {
           Method privateMethod = TargetClass.class
                                  .getDeclaredMethod("getTheAnswerInternal");
           privateMethod.setAccessible(true);
           System.out.println("Result of the call "
                                      + privateMethod.invoke(targetObject));
       } catch (NoSuchMethodException e) {
           e.printStackTrace();
       } catch (IllegalAccessException e) {
           e.printStackTrace();
       } catch (InvocationTargetException e) {
           e.printStackTrace();
       }
   }
}
```

java.lang.reflect.Constructor<T>

- Represents a class constructor
- Allows to instantiate new objects of a Class<T>

How to obtain a Constructor

Class<T> provides methods that ...

- returns an array of **public** constructors, including inherited:

```
myClass.getConstructors();
```

- returns an array of constructors declared by the current class, excluding the inherited ones:

```
myClass.getDeclaredConstructors();
```

- Returns the **public** constructor with the given parameter types:

```
myClass.getConstructor(Class<?>... paramTypes);
```

- Returns the declared constructor of the current class:

```
myClass.getDeclaredConstructor(Class<?>... paramTypes);
```

Invoking a constructor

It is possible to invoke the constructor with 2 approaches:

- With the newInstance method in Class<T>, deprecated since Java 9 because it throws checked and unchecked exceptions (but is not declared to do so):

```
myClass.newInstance(...params); // deprecated since Java 9
```

- By using the newInstance method available in java.reflect.Constructor:

```
aConstructor.newInstance(...params);
```

Invoking a constructor

- `Class.newInstance()` can only invoke the zero-argument constructor. `Constructor.newInstance()` allows to invoke any constructor, regardless of the number of parameters.
- `Class.newInstance()` throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. `Constructor.newInstance()` always wraps the thrown exception with an `InvocationTargetException.`
- Class.newInstance() requires the constructor to be visible. `Constructor.newInstance()` also allows to invoke private constructors under certain circumstances.

Example

```
import java.lang.reflect.Constructor;

public class ConstructorsTest {
   public static void main(String[] args) throws ClassNotFoundException,
    NoSuchMethodException, IllegalAccessException, InvocationTargetException,
    InstantiationException {
       Class<?> myClass = Class.forName("TargetClass");
       Constructor constructor = myClass.getDeclaredConstructor(Integer.TYPE);
       Object instance = constructor.newInstance(42);
       System.out.println("Reflection: the answer is " +
         myClass.getMethod("getTheAnswer").invoke(instance)); // 42
   }
}
```

Obtaining Class<T> from the members

- All the classes representing the contained members (Field, Method, Constructor), allows to obtain the Class<T> of the class in which the member is declared.

```
java.lang.reflect.Field.getDeclaringClass()
java.lang.reflect.Method.getDeclaringClass()
java.lang.reflect.Constructor.getDeclaringClass()
```

Obtaining Class<T> for nested classes

- For declared members that are nested classes, Class<T> provides:

```
Class.getDeclaringClass();
Class.getEnclosingClass();
```

- Depending on the nesting type, getDeclaringClass() might return null (e.g. for anonymous inner classes). For this reason getEnclosingClass() is provided.
- For local and anonymous nested classes, other methods such as getEnclosingMethod() and getEnclosingConstructor() are also provided.

getDeclaringClass returns null

```
public class MainClass {
   public static void main(String[] args) {
       MainClass test = new MainClass();
       test.test();
   }

   public void test() {
       class LocalClass {
           public LocalClass() {
               System.out.println("enclosing "
                       + LocalClass.class.getEnclosingClass()); // MainClass
               System.out.println("declaring "
                       + LocalClass.class.getDeclaringClass()); // null
           }
       }
       LocalClass localClass = new LocalClass();
       //...
   }
}
```

// when the target class is a local class ...

getDeclaringClass returns null

```
public class AnonymousDeclaring {
   public static void main(String[] args) {
       AnonymousDeclaring anonymous = new AnonymousDeclaring();
   }

   public AnonymousDeclaring() {
       new Object() {
           void runMe() {
               System.out.println("declared "
                        + this.getClass().getDeclaringClass()); // null
               System.out.println("enclosing "
                        + this.getClass().getEnclosingClass()); // class AnonymousDeclaring
           }
       }.runMe();
   }
}
```

// ... or when the target class is an anonymous inner class

Additional support in java.lang.reflect

- The package java.lang.reflect also provides:

```
java.lang.reflect.Modifier // static methods and constants to decode class and member access modifiers
java.lang.reflect.Parameter // information about method parameters. The parameter name isn't usually stored in the compiled class
java.lang.reflect.Array // static methods to dynamically create and access arrays
```

Example

```
class Calculate {
    int add(int a, int b) {
        return (a + b);
    }

    int mul(int a, int b) {
        return (b * a);
    }

    long subtract(long a, long b) {
        return (a - b);
    }
}
```

Example

```
public class ParameterExample {
    public static void main(String[] args) {
        Class<Calculate> cls = Calculate.class;
        Method[] methods = cls.getDeclaredMethods();
        for (Method method : methods) {
            Parameter[] parameters = method.getParameters();
            for (Parameter parameter : parameters) {
                System.out.print(
                    parameter.getParameterizedType() + " ");
                System.out.print(parameter.getName() + " ");
            }
        }
    }
}
```

Warning

Don’t overuse reflection in Java

- Reflection can break information hiding (you can read private fields)
- Class names could be written as string literals, so the compiler and the IDE cannot perform formal checks
- Reflection is usually slower (and more difficult to read) than regular code

- As a rule of thumb: use the reflection APIs only when there is no other way to do the task

Summary

- Preamble about unit testing
- Introduction to reflection and metaprogramming
- Introduction to Class<T>
- How to obtain the Class<T> instance
- Handling special cases like primitive types and arrays
- Support for inheritance in Class<T>
- Fields, Methods and Constructors
- Obtain the Class<T> instance from a member
- Additional support in java.lang.reflect