Java Reflection

In this lesson, we explore Java reflection, which allows us to inspect and/or modify runtime attributes of classes, interfaces, fields and methods. This particularly comes in handy when we don't know their names at compile time.

Additionally, we can instantiate new objects, invoke methods and get or set field values using reflection.

Simple Example

To use Java reflection, we don't need to include any special jars, any special configuration or Maven dependencies. The JDK ships with a group of classes that are bundled in the *java.lang.reflect* package specifically for this purpose.

So, all we need to do is to make the following import into our code:

```
import java.lang.reflect.*;
```

To get access to the class, method and field information of an instance, we call the *getClass* method, which returns the runtime class representation of the object. The returned *class* object provides methods for accessing information about a class.

Let's create a simple *Person* class with only *name* and *age* fields and no methods at all.

Here is the Person class:

```
public class Person {
    private String name;
    private int age;
}
```

We'll now use Java reflection to discover the names of all fields of this class.

To appreciate the power of reflection, let's construct a *Person* object and use Object as the reference type:

```
@Test
public void givenObject_whenGetsFieldNamesAtRuntime_thenCorrect() {
    Object person = new Person();
    Field[] fields = person.getClass().getDeclaredFields();

    List<String> actualFieldNames = getFieldNames(fields);

    assertTrue(Arrays.asList("name", "age").containsAll(actualFieldN
}
```

This test shows us that we are able to get an array of *Field* objects from our *person* object, even if the reference to the object is a parent type of that object.

Notice how we use a helper method to extract the actual field names.

```
private static List<String> getFieldNames(Field[] fields) {
    return Arrays.stream(fields).map(f -> f.getName()).collect(Colle
}
```

Inspecting Java Classes

Java class objects, as we mentioned earlier, give us access to the internal details of any object. We are going to examine internal details such as an object's class name, modifiers, fields, methods, implemented interfaces, etc.

Getting Ready

Let's create an abstract *Animal* class that implements the *Eating* interface. This interface defines the eating behavior of any concrete *Animal* object we create.

First, here is the *Eating* interface:

```
public interface Eating {
    String eats();
}
```

And here is the abstract *Animal* implementation of the *Eating* interface:

```
public abstract class Animal implements Eating {

    public static String CATEGORY = "domestic";
    private String name;

    protected abstract String getSound();

    // constructor, standard getters and setters omitted
}
```

Let's also create another interface called *Locomotion* that describes how an animal moves:

```
public interface Locomotion {
    String getLocomotion();
}
```

We'll now create a concrete class called *Goat* that extends *Animal* and implements *Locomotion*. Since the superclass implements *Eating*, *Goat* will have to implement that interface's methods as well:

```
public class Goat extends Animal implements Locomotion {

    @Override
    protected String getSound() {
        return "bleat";
    }

    @Override
    public String getLocomotion() {
        return "walks";
    }

    @Override
    public String eats() {
        return "grass";
    }

    // constructor omitted
}
```

From this point onward, we will use Java reflection to inspect aspects of Java objects that appear in the classes and interfaces above.

Class Names

Let's start by getting the name of an object from the *Class*:

```
@Test
public void givenObject_whenGetsClassName_thenCorrect() {
    final Object goat = new Goat("goat");
    final Class<?> clazz = goat.getClass();

    assertEquals("Goat", clazz.getSimpleName());
    assertEquals("com.nbicocchi.tutorials.reflection.Goat", clazz.g
    assertEquals("com.nbicocchi.tutorials.reflection.Goat", clazz.g
}
```

Note that the *getSimpleName* method of *Class* returns the basic name of the object as it would appear in its declaration. Then the other two methods return the fully qualified class name including the package declaration.

Let's also see how we can investigate an object of the *Goat* class if we only know its fully qualified class name:

```
@Test
public void givenClassName_whenCreatesObject_thenCorrect() throws C
    final Class<?> clazz = Class.forName("com.nbicocchi.tutorials.r

    assertEquals("Goat", clazz.getSimpleName());
    assertEquals("com.nbicocchi.tutorials.reflection.Goat", clazz.g
    assertEquals("com.nbicocchi.tutorials.reflection.Goat", clazz.g
}
```

Notice that the name we pass to the static *forName* method should include the package information. Otherwise, we will get a *ClassNotFoundException*.

Class Modifiers

We can determine the modifiers used in a class by calling the *getModifiers* method, which returns an *Integer*. Each modifier is a flag bit that is either set or cleared.

The *java.lang.reflect.Modifier* class offers static methods that analyze the returned *Integer* for the presence or absence of a specific modifier.

```
@Test
public void givenClass_whenRecognisesModifiers_thenCorrect() throws
    final Class<?> goatClass = Class.forName("com.nbicocchi.tutoria
    final Class<?> animalClass = Class.forName("com.nbicocchi.tutor
    final int goatMods = goatClass.getModifiers();
    final int animalMods = animalClass.getModifiers();

    assertTrue(Modifier.isPublic(goatMods));
    assertTrue(Modifier.isAbstract(animalMods));
    assertTrue(Modifier.isPublic(animalMods));
}
```

We are able to inspect modifiers of any class located in a library jar that we are importing into our project.

In most cases, we may need to use the *forName* approach rather than the full-blown instantiation since that would be an expensive process in the case of memory-heavy classes.

Package Information

By using Java reflection, we are also able to get information about the package of any class or object. This data is bundled inside the *Package* class, which is returned by a call to *getPackage* method on the class object.

```
@Test
public void givenClass_whenGetsPackageInfo_thenCorrect() {
    final Goat goat = new Goat("goat");
    final Class<?> goatClass = goat.getClass();
    final Package pkg = goatClass.getPackage();

    assertEquals("com.nbicocchi.tutorials.reflection", pkg.getName(
}
```

Superclass

We are also able to obtain the superclass of any Java class by using Java reflection. In many cases, especially while using library classes or Java's built-in classes, we may not know beforehand the superclass of an object we are using.

Let's go ahead and determine the superclass of *Goat*. Additionally, we also show that *java.lang.String* class is a subclass of *java.lang.Object* class:

```
@Test
public void givenClass_whenGetsSuperClass_thenCorrect() {
    final Goat goat = new Goat("goat");
    final String str = "any string";

    assertEquals("Animal", goat.getClass().getSuperclass().getSimple
    assertEquals("Object", str.getClass().getSuperclass().getSimpleN
}
```

Implemented Interfaces

Using Java reflection, we are also able to **get the list of interfaces implemented by a given class.**

Let's retrieve the class types of the interfaces implemented by the *Goat* class and the *Animal* abstract class:

```
@Test
public void givenClass_whenGetsImplementedInterfaces_thenCorrect(){
    final Class<?> goatClass = Class.forName("com.nbicocchi.tutorial
    final Class<?> animalClass = Class.forName("com.nbicocchi.tutor

    Class<?>[] goatInterfaces = goatClass.getInterfaces();
    Class<?>[] animalInterfaces = animalClass.getInterfaces();

    assertEquals(1, goatInterfaces.length);
    assertEquals(1, animalInterfaces.length);
    assertEquals("Locomotion", goatInterfaces[0].getSimpleName());
    assertEquals("Eating", animalInterfaces[0].getSimpleName());
}
```

Notice from the assertions that each class implements only a single interface. Inspecting the names of these interfaces, we find that *Goat* implements *Locomotion* and *Animal* implements *Eating*, just as it appears in our code.

We can see that *Goat* is a subclass of the abstract class *Animal* and implements the interface method *eats()*. Then, *Goat* also implements the *Eating* interface.

It is therefore worth noting that only those interfaces that a class explicitly declares as implemented with the *implements* keyword appear in the returned array.

So, even if a class implements interface methods because its superclass implements that interface, but the subclass does not directly declare that interface with the *implements* keyword, that interface will not appear in the array of interfaces.

Constructors, Methods and Fields

With Java reflection, we are able to inspect the constructors of any object's class as well as methods and fields.

Later, we'll be able to see deeper inspections on each of these components of a class. But for now, it's enough to just get their names and compare them with what we expect.

Let's see how to get the constructor of the *Goat* class:

```
@Test
public void givenClass_whenGetsConstructor_thenCorrect() throws Cla
    final Class<?> goatClass = Class.forName("com.nbicocchi.tutoria
    final Constructor<?>[] constructors = goatClass.getConstructors(

    assertEquals(1, constructors.length);
    assertEquals("com.nbicocchi.tutorials.reflection.Goat", constru
}
```

We can also inspect the fields of the *Animal* class:

```
@Test
public void givenClass_whenGetsFields_thenCorrect(){
    final Class<?> animalClass = Class.forName("com.nbicocchi.tutor
    Field[] fields = animalClass.getDeclaredFields();

    List<String> actualFields = getFieldNames(fields);

    assertEquals(2, actualFields.size());
    assertTrue(actualFields.containsAll(Arrays.asList("name", "CATE
}
```

And we can similarly inspect the methods of the *Animal* class:

```
@Test
public void givenClass_whenGetsMethods_thenCorrect(){
    final Class<?> animalClass = Class.forName("com.nbicocchi.tutor
    Method[] methods = animalClass.getDeclaredMethods();
    List<String> actualMethods = getMethodNames(methods);

    assertEquals(4, actualMethods.size());
    assertTrue(actualMethods.containsAll(Arrays.asList("getName",
      "setName", "getSound")));
}
```

Inspecting Constructors

With Java reflection, we can **inspect constructors** of any class and even **create class objects at runtime.** This is made possible by the *java.lang.reflect.Constructor* class.

In Java, as we know, no two constructors of a class share exactly the same method signature. So, we will use this uniqueness to get one constructor from many.

To appreciate the features of this class, we will create a *Bird* subclass of *Animal* with three constructors. We will not implement *Locomotion* so that we can specify that behavior using a constructor argument, to add still more variety:

```
public class Bird extends Animal {
    private boolean walks;

    public Bird() {
        super("bird");
    }

    public Bird(String name, boolean walks) {
        super(name);
        setWalks(walks);
    }

    public Bird(String name) {
        super(name);
    }

    @Override
    public String eats() {
        return "grains";
    }

    @Override
    protected String getSound() {
        return "chaps";
    }

    public boolean walks() {
        return walks;
    }

    public void setWalks(boolean walks) {
        this.walks = walks;
    }
}
```

Let's confirm by using reflection that this class has three constructors:

```
@Test
public void givenClass_whenGetsAllConstructors_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Constructor<?>[] constructors = birdClass.getConstructors

    assertEquals(3, constructors.length);
}
```

Next, we will retrieve each constructor for the *Bird* class by passing the constructor's parameter class types in declared order:

```
@Test
public void givenClass_whenGetsEachConstructorByParamTypes_thenCorre
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria

    Constructor<?> cons1 = birdClass.getConstructor();
    Constructor<?> cons2 = birdClass.getConstructor(String.class);
    Constructor<?> cons3 = birdClass.getConstructor(String.class, b
}
```

There is no need for assertion since we'll get a *NoSuchMethodException* and the test will automatically fail when a constructor with given parameter types in the given order does not exist.

In the last test, we'll see how to instantiate objects at runtime while supplying their parameters:

```
@Test
public void givenClass_whenInstantiatesObjectsAtRuntime_thenCorrect
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria

    final Constructor<?> cons1 = birdClass.getConstructor();
    final Constructor<?> cons2 = birdClass.getConstructor(String.cl
    final Constructor<?> cons3 = birdClass.getConstructor(String.cl

    final Bird bird1 = (Bird) cons1.newInstance();
    final Bird bird2 = (Bird) cons2.newInstance("Weaver bird");
    final Bird bird3 = (Bird) cons3.newInstance("dove", true);

    assertEquals("bird", bird1.getName());
    assertEquals("Weaver bird", bird2.getName());
    assertEquals("dove", bird3.getName());
    assertFalse(bird1.walks());
    assertTrue(bird3.walks());
}
```

We instantiate class objects by calling the *newInstance* method of *Constructor* class and passing the required parameters in declared order. We then cast the result to the required type.

For *bird1*, we use the default constructor that automatically sets the name to bird from our *Bird* code, and we confirm that with a test. We then instantiate *bird2* with only a name and test as well. Remember that when we don't set locomotion behavior, it defaults to false as seen in the last two assertions.

Inspecting Fields

Previously, we only inspected the names of fields. In this section, **we will show how to get and set their values at runtime.**

There are two main methods used to inspect fields of a class at runtime: *getFields()* and *getField(fieldName)*.

The *getFields()* method returns all accessible public fields of the class in question. It will return all the public fields in both the class and all superclasses.

For instance, when we call this method on the *Bird* class, we will only get the *CATEGORY* field of its superclass, *Animal*, since *Bird* itself does not declare any public fields:

```
@Test
public void givenClass_whenGetsPublicFields_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Field[] fields = birdClass.getFields();

    assertEquals(1, fields.length);
    assertEquals("CATEGORY", fields[0].getName());
}
```

This method also has a variant called *getField* that returns only one *Field* object by taking the name of the field:

```
@Test
public void givenClass_whenGetsPublicFieldByName_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    Field field = birdClass.getField("CATEGORY");

    assertEquals("CATEGORY", field.getName());
}
```

We are not able to access private fields declared in superclasses and not declared in the child class. This is why we can't access the *name* field.

However, we can inspect private fields declared in the class we are dealing with by calling the *getDeclaredFields* method:

```
@Test
public void givenClass_whenGetsDeclaredFields_thenCorrect(){
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Field[] fields = birdClass.getDeclaredFields();

    assertEquals(1, fields.length);
    assertEquals("walks", fields[0].getName());
}
```

We can also use its other variant in case we know the name of the field:

```
@Test
public void givenClass_whenGetsFieldsByName_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Field field = birdClass.getDeclaredField("walks");

    assertEquals("walks", field.getName());
}
```

If we get the name of the field wrong or type in a nonexistent field, we'll get a *NoSuchFieldException*.

Now we'll get the field type:

```
@Test
public void givenClassField_whenGetsType_thenCorrect() {
    final Field field = Class.forName("com.nbicocchi.tutorials.refl
    final Class<?> fieldClass = field.getType();

    assertEquals("boolean", fieldClass.getSimpleName());
}
```

Next, let's look at how to access field values and modify them.

To get the value of a field, let alone set it, we have to first set it's accessible by calling *setAccessible* method on the *Field* object and pass boolean *true* to it:

```
@Test
public void givenClassField_whenSetsAndGetsValue_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Bird bird = (Bird) birdClass.getConstructor().newInstance
    final Field field = birdClass.getDeclaredField("walks");
    field.setAccessible(true);

    assertFalse(field.getBoolean(bird));
    assertFalse(bird.walks());

    field.set(bird, true);

    assertTrue(field.getBoolean(bird));
    assertTrue(bird.walks());
}
```

In the above test, we ascertain that indeed the value of the *walks* field is false before setting it to true.

Notice how we use the *Field* object to set and get values by passing it the instance of the class we are dealing with and possibly the new value we want the field to have in that object.

One important thing to note about *Field* objects is that when it is declared as *public static*, we don't need an instance of the class containing them.

We can just pass *null* in its place and still obtain the default value of the field:

```
@Test
public void givenClassField_whenGetsAndSetsWithNull_thenCorrect(){
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Field field = birdClass.getField("CATEGORY");
    field.setAccessible(true);

    assertEquals("domestic", field.get(null));
}
```

Inspecting Methods

In a previous example, we used reflection only to inspect method names. However, Java reflection is more powerful than that. With Java reflection, we can **invoke methods at runtime** and pass them their required parameters, just like we did for constructors. Similarly, we can also invoke overloaded methods by specifying parameter types of each.

Just like fields, there are two main methods that we use for retrieving class methods. The *getMethods* method returns an array of all public methods of the class and superclasses. This means that with this method, we can get public methods of the *java.lang.Object* class such as *toString*, *hashCode* and *notifyAll*:

```
@Test
public void givenClass_whenGetsAllPublicMethods_thenCorrect(){
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Method[] methods = birdClass.getMethods();
    final List<String> methodNames = getMethodNames(methods);

    assertTrue(methodNames.containsAll(Arrays.asList("equals", "not
}
```

To get only public methods of the class we are interested in, we have to use *getDeclaredMethods* method:

```
@Test
public void givenClass_whenGetsOnlyDeclaredMethods_thenCorrect(){
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final List<String> actualMethodNames = getMethodNames(birdClass

    final List<String> expectedMethodNames = Arrays.asList("setWalk

    assertEquals(expectedMethodNames.size(), actualMethodNames.size
    assertTrue(expectedMethodNames.containsAll(actualMethodNames));
    assertTrue(actualMethodNames.containsAll(expectedMethodNames));
}
```

Each of these methods has the singular variation that returns a single *Method* object whose name we know:

```
@Test
public void givenMethodName_whenGetsMethod_thenCorrect() throws Exc
    final Bird bird = new Bird();
    final Method walksMethod = bird.getClass().getDeclaredMethod("wa
    final Method setWalksMethod = bird.getClass().getDeclaredMethod

    assertTrue(walksMethod.canAccess(bird));
    assertTrue(setWalksMethod.canAccess(bird));
}
```

Notice how we retrieve individual methods and specify what parameter types they take. Those that don't take parameter types are retrieved with an empty variable argument, leaving us with only a single argument, the method name.

Next, we will show how to invoke a method at runtime. We know by default that the *walks* attribute of the *Bird* class is *false*. We want to call its *setWalks* method and set it to *true*:

```
@Test
public void givenMethod_whenInvokes_thenCorrect() {
    final Class<?> birdClass = Class.forName("com.nbicocchi.tutoria
    final Bird bird = (Bird) birdClass.getConstructor().newInstance
    final Method setWalksMethod = birdClass.getDeclaredMethod("setW
    final Method walksMethod = birdClass.getDeclaredMethod("walks")
    final boolean walks = (boolean) walksMethod.invoke(bird);

    assertFalse(walks);
    assertFalse(bird.walks());

    setWalksMethod.invoke(bird, true);
    final boolean walks2 = (boolean) walksMethod.invoke(bird);

    assertTrue(walks2);
    assertTrue(bird.walks());
}
```

Notice how we first invoke the *walks* method and cast the return type to the appropriate data type and then check its value. We later invoke the *setWalks* method to change that value and test again.