(/)

Guide to Java Reflection

Last modified: September 8, 2019

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) +

1. Overview

In this article we will be exploring java reflection, which allows us to inspect or/and 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.

2. Project Setup

To use java reflection, we do not 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* (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html) package specifically for this purpose.

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

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

and we are good to go.

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.

3. Simple Example

To get our feet wet, we are going to take a look at a very basic example which inspects the fields of a simple java object at runtime.

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 will now use java reflection to discover the names of all fields of this class. To appreciate the power of reflection, we will 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(actualFieldNames));
}
```

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.

In the above example, we were only interested in the names of those fields, but there is much more that can be done and we will see further examples of this in the subsequent sections.

Notice how we use a helper method to extract the actual field names, it's a very basic code:


```
private static List<String> getFieldNames(Field[] fields) {
    List<String> fieldNames = new ArrayList<>();
    for (Field field : fields)
      fieldNames.add(field.getName());
    return fieldNames;
}
```

4. Java Reflection Use Cases

Before we proceed to the different features of java reflection, we will discuss some of the common uses we may find for it. Java reflection is extremely powerful and can come in very handy in a number of ways.

For instance, in many cases we have a naming convention for database tables. We may choose to add consistency by pre-fixing our table names with *tbl_*, such that a table with student data is called *tbl_student_data*.

In such cases, we may name the java object holding student data as *Student* or *StudentData.* Then using the CRUD paradigm, we have one entry point for each operation such that *Create* operations only receive an *Object* parameter.

We then use reflection to retrieve the object name and field names. At this point, we can map this data to a DB table and assign the object field values to the appropriate DB field names.

5. Inspecting Java Classes

In this section, we will explore the most fundamental component in the java reflection API. 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, their modifiers, fields, methods, implemented interfaces etc.

5.1. Getting Ready

To get a firm grip on the reflection API, as applied to java classes and have examples with variety, we will create an abstract *Animal* class which implements the *Eating* interface. This interface defines the eating behavior of any concrete *Animal* object we create.

So firstly, here is the *Eating* interface:

```
1  public interface Eating {
2      String eats();
3  }
```

and then the concrete *Animal* implementation of the *Eating* interface:

```
1  public abstract class Animal implements Eating {
2
3      public static String CATEGORY = "domestic";
4      private String name;
5
6      protected abstract String getSound();
7
8      // constructor, standard getters and setters omitted
9  }
```

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

```
1  public interface Locomotion {
2      String getLocomotion();
3  }
```

We will now create a concrete class called *Goat* which extends *Animal* and implements *Locomotion*. Since the super class 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.

5.2. Class Names

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

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

    assertEquals("Goat", clazz.getSimpleName());
    assertEquals("com.baeldung.reflection.Goat", clazz.getName());
    assertEquals("com.baeldung.reflection.Goat", clazz.getCanonicalName());
}
```

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

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

```
@Test
public void givenClassName_whenCreatesObject_thenCorrect(){
    Class<?> clazz = Class.forName("com.baeldung.reflection.Goat");

    assertEquals("Goat", clazz.getSimpleName());
    assertEquals("com.baeldung.reflection.Goat", clazz.getName());
    assertEquals("com.baeldung.reflection.Goat", clazz.getCanonicalName());
}
```

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

5.3. Class Modifiers

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

The *java.lang.reflect.Modifier* (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Modifier.html) class offers static methods that analyze the returned *Integer* for the presence or absence of a specific modifier.

Let's confirm the modifiers of some of the classes we defined above:

```
@Test
public void givenClass_whenRecognisesModifiers_thenCorrect() {
    Class<?> goatClass = Class.forName("com.baeldung.reflection.Goat");
    Class<?> animalClass = Class.forName("com.baeldung.reflection.Animal");

    int goatMods = goatClass.getModifiers();
    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.

5.4. 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* (https://docs.oracle.com/javase/8/docs/api/java/lang/Package.html) class which is returned by a call to *getPackage* method on the class object.

Lets run a test to retrieve the package name:

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

    assertEquals("com.baeldung.reflection", pkg.getName());
}
```

5.5. Super Class

We are also able to obtain the super class of any java class by using java reflection.

In many cases, especially while using library classes or java's builtin classes, we may not know beforehand the super class of an object we are using, this sub section will show how to obtain this information.

So let's go ahead and determine the super class 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() {
    Goat goat = new Goat("goat");
    String str = "any string";

    Class<?> goatClass = goat.getClass();
    Class<?> goatSuperClass = goatClass.getSuperclass();

    assertEquals("Animal", goatSuperClass.getSimpleName());
    assertEquals("Object", str.getClass().getSuperclass().getSimpleName());
}
```

5.6. 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(){
    Class<?> goatClass = Class.forName("com.baeldung.reflection.Goat");
    Class<?> animalClass = Class.forName("com.baeldung.reflection.Animal");

    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.

You may have observed 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 it's super class implements that interface, but the subclass does not directly declare that interface with the *implements* keyword, then that interface will not appear in the array of interfaces.

5.7. 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.

We will later on be able to see deeper inspections on each of these components of a class but for now, it suffices 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(){
    Class<?> goatClass = Class.forName("com.baeldung.reflection.Goat");

    Constructor<?>[] constructors = goatClass.getConstructors();

    assertEquals(1, constructors.length);
    assertEquals("com.baeldung.reflection.Goat", constructors[0].getName());
}
```

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

```
@Test
public void givenClass_whenGetsFields_thenCorrect(){
    Class<?> animalClass = Class.forName("com.baeldung.java.reflection.Animal");
    Field[] fields = animalClass.getDeclaredFields();

    List<String> actualFields = getFieldNames(fields);

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

Just like we can inspect the methods of the *Animal* class:

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

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

Just like *getFieldNames*, we have added a helper method to retrieve method names from an array of *Method* objects:

```
private static List<String> getMethodNames(Method[] methods) {
    List<String> methodNames = new ArrayList<>();
    for (Method method : methods)
      methodNames.add(method.getName());
    return methodNames;
}
```

6. 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* (https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Constructor.html) class.

Earlier on, we only looked at how to get the array of *Constructor* objects, from which we were able to get the names of the constructors.

In this section, we will focus on how to retrieve specific constructors. 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);
    }

    public boolean walks() {
        return walks;
    }

    // standard setters and overridden methods
}
```

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

```
@Test
public void givenClass_whenGetsAllConstructors_thenCorrect() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    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_thenCorrect(){
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");

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

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

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

```
@Test
public void givenClass_whenInstantiatesObjectsAtRuntime_thenCorrect() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    Constructor<?> cons1 = birdClass.getConstructor();
    Constructor<?> cons2 = birdClass.getConstructor(String.class);
    Constructor<?> cons3 = birdClass.getConstructor(String.class,
      boolean.class);

    Bird bird1 = (Bird) cons1.newInstance();
    Bird bird2 = (Bird) cons2.newInstance("Weaver bird");
    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 which from our *Bird* code, automatically sets the name to bird 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 as it defaults to false, as seen in the last two assertions.

7. 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 super classes.

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

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

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

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

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

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

We are not able to access private fields declared in super classes and not declared in the child class. This is why we are not able to 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(){
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    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() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    Field field = birdClass.getDeclaredField("walks");

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

If we get the name of the field wrong or type in an in-existent field, we will get a *NoSuchFieldException*.

We get the field type as follows:

```
@Test
public void givenClassField_whenGetsType_thenCorrect() {
    Field field = Class.forName("com.baeldung.reflection.Bird")
      .getDeclaredField("walks");
    Class<?> fieldClass = field.getType();

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

Next, we will look at how to access field values and modify them. To be able to get the value of a field, let alone setting it, we must 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() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    Bird bird = (Bird) birdClass.newInstance();
    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*, then we don't need an instance of the class containing them, we can just pass *null* in it's place and still obtain the default value of the field, like so:

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

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

8. 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 super classes.

This means that with this method, we can get public methods of the *java.lang.Object* class like *toString, hashCode* and *notifyAll*:

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

    assertTrue(methodNames.containsAll(Arrays
      .asList("equals", "notifyAll", "hashCode",
        "walks", "eats", "toString")));
}
```

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

```
@Test
public void givenClass_whenGetsOnlyDeclaredMethods_thenCorrect(){
    Class<?> birdClass = Class.forName("com.baeldung.java.reflection.Bird");
    List<String> actualMethodNames
      = getMethodNames(birdClass.getDeclaredMethods());

    List<String> expectedMethodNames = Arrays
      .asList("setWalks", "walks", "getSound", "eats");

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

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

```
@Test
public void givenMethodName_whenGetsMethod_thenCorrect() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    Method walksMethod = birdClass.getDeclaredMethod("walks");
    Method setWalksMethod = birdClass.getDeclaredMethod("setWalks", boolean.class);

    assertFalse(walksMethod.isAccessible());
    assertFalse(setWalksMethod.isAccessible());

    walksMethod.setAccessible(true);
    setWalksMethod.setAccessible(true);

    assertTrue(walksMethod.isAccessible());
    assertTrue(setWalksMethod.isAccessible());
}
```

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() {
    Class<?> birdClass = Class.forName("com.baeldung.reflection.Bird");
    Bird bird = (Bird) birdClass.newInstance();
    Method setWalksMethod = birdClass.getDeclaredMethod("setWalks", boolean.class);
    Method walksMethod = birdClass.getDeclaredMethod("walks");
    boolean walks = (boolean) walksMethod.invoke(bird);

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

    setWalksMethod.invoke(bird, true);

    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 then later invoke the *setWalks* method to change that value and test again.

9. Conclusion

In this tutorial, we have covered the Java Reflection API and looked at how to use it to inspect classes, interfaces, fields and methods at runtime without prior knowledge of their internals by compile time.

The full source code and examples for this tutorial can be found in my Github (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-reflection) project.