Java Reflection

Contents

Preface

This guide is about reflection, the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of the program at runtime.

We are going to explain what reflection is in general and how can be used in Java. Real uses cases about different reflection uses are listed in the next chapters.

Several code snippets will be shown; at the end of this tutorial you can find a compressed file that contains all these examples (and some more).

All code has been written using Eclipse Luna 4.4 and Java update 8.25, no third party libraries are needed.

Chapter 1

Reflection

The concept of reflection in software means the ability to inspect, analyze and modify other code at runtime. For example imagine an application that takes as input some files containing source code (we do not care about what source code yet). The goal of this application is to count the number of methods that are contained in each passed class. This can be solved using reflection by analyzing the code and counting the elements which are actually methods, ignoring other kind of elements like attributes, interfaces, etc, and grouping them by classes.

Purely speaking, this example is not really reflection, because the code does not have to be analyzed at runtime and the task can be done in any other stage, but it can be also done at runtime and then we would be actually talking about reflection.

Another example would be an application that analyzes the content of given classes and executes the methods that contain a specific annotation with arguments provided in runtime: In the Java Junit framework we have for example the annotation `@Test`. This is actually what Junit does; and does it using reflection.

Chapter 2

Introduction to reflection in Java

In Java, it is possible to inspect fields, classes, methods, annotations, interfaces, etc. at runtime. You do not need to know how classes or methods are called, neither the parameters that are needed, all of that can be retrieved at runtime using reflection. It is also possible to instantiate new classes, to create new instances and to execute their methods, all of it using reflection.

Reflection is present in Java since the beginning of the times via its reflection API. The class `Class` contains all the reflection related methods that can be applied to classes and objects like the ones that allow a programmer to retrieve the class name, to retrieve the public methods of a class, etc. Other important classes are `Method`, `Field` and `Type` containing specific reflection methods that we are going to see in this tutorial.

Although reflection is very useful in many scenarios, it should not be used for everything. If some operation can be executed without using reflection, then we should not use it. Here are some reasons:

- The performance is affected by the use of reflection since all compilation optimizations cannot be applied: reflection is resolved at runtime and not at compile stages.
- Security vulnerabilities have to be taken into consideration since the use of reflection may not be possible when running in secure contexts like Applets.
- Another important disadvantage that is good to mention here is the maintenance of the code. If your code uses reflection heavily it is going to be more difficult to maintain. The classes and methods are not directly exposed in the code and may vary dynamically so it can get difficult to change the number of parameters that a method expects if the code that calls this method is invoked via reflection.
- Tools that automatically refactor or analyze the code may have trouble when a lot of reflection is present.

Chapter 3

Use cases

Despite all the limitations, reflection is a very powerful tool in Java that can be taken into consideration in several scenarios.

In general, reflection can be used to observe and modify the behavior of a program at runtime. Here is a list with the most common use cases:

- IDEs can heavily make use of reflection in order to provide solutions for auto completion features, dynamic typing, hierarchy structures, etc. For example, IDEs like Eclipse or PHP Storm provide a mechanism to retrieve dynamically the arguments expected for a given method or a list of public methods starting by "get" for a given instance. All these are done using reflection.
- Debuggers use reflection to inspect dynamically the code that is being executed.
- Test tools like Junit or Mockito use reflection in order to invoke desired methods containing specific syntax or to mock specific classes, interfaces and methods.
- Dependency injection frameworks use reflection to inject beans and properties at runtime and initialize all the context of an application.
- Code analysis tools like PMD or Findbugs use reflection in order to analyze the code against the list of code violations that are currently configured.
- External tools that make use of the code dynamically may use reflection as well.

In this tutorial we are going to see several examples of use of reflection in Java. We will see how to get all methods for a given instance, without knowing what kind of class this instance is and we are going to invoke different methods depending on their syntax.

We are not just going to show what other tutorials do, but we will go one step forward by indicating how to proceed when using reflection with generics, annotations, arrays, collections and other kind of objects. Finally we will explain the main new features coming out with Java 8 related to this topic.

Chapter 4

Reflection components and mechanisms

In order to start coding and using reflection in Java we first have to explain a couple of concepts that may be relevant.

- `Interface` in Java is a contract with the applications that may use them. Interfaces contain a list of methods that are exposed and that have to be implemented by the subclasses implementing these interfaces. Interfaces cannot be instantiated. Since Java 8 they can contain default method implementations although this is not the common use.
- `Class` is the implementation of a series of methods and the container of a series of properties. It can be instantiated.
- `Object` is an instance of a given class.
- `Method` is some code performing some actions. They have return types as outputs and input parameters.
- `Field` is a property of a class.
- `Enums` are elements containing a set of predefined constants.
- `Private` element is an element that is only visible inside a class and cannot be accessed from outside. It can be a method, a field, etc.
- `Static` elements are elements that belong to the class and not to a specific instance. Static elements can be fields used across all instances of a given class, methods that can be invoked without need to instantiate the class, etc. This is very interesting while using reflection since it is different to invoke a static method than a non static one where you need an instance of a class to execute it.
- `Annotation` is code Meta data informing about the code itself.
- `Collection` is a group of elements, can be a List, a Map, a Queue, etc.
- `Array` is an object containing a fixed number of values. Its length is fixed and is specified on creation.
- `Dynamic proxy` is a class implementing a list of interfaces specified at runtime. They use the class `java.lang.reflect.Proxy`. We will see this more in detail in the next chapters.
- `Class loader` is an object in charge of loading classes given the name of a class. In Java, every class provide methods to retrieve the class loader: `Class.getClassLoader()`.
- `Generics` were introduced in java update 5. They offer compile time safety by indicating what type or sub types a collection is going to use. For example using generics you can prevent that an application using a list containing strings would try to add a Double to the list in compile time.

The different nature of these components is important in order to use reflection within them. Is not the same to try to invoke a private method than a public one; it is different to get an annotation name or an interface one, etc. We will see examples for all of these in the next chapters.

Chapter 5

Classes

Everything in Java is about classes, reflection as well. Classes are the starting point when we talk about reflection. The class `java.lang.Class` contains several methods that allow programmers to retrieve information about classes and objects (and other elements) at runtime.

In order to retrieve the class information from a single instance we can write (in this case, for the `String` class):

```
Class<? extends String> stringGetClass = stringer.getClass();
```

Or directly from the class name without instantiation:

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

or using the `java.lang.Class.forName(String)` method:

```
Class.forName( "java.lang.String" )
```

From a class object we can retrieve all kind of information like declared methods, constructors, visible fields, annotations, types... In this tutorial all these is explained in the following chapters.

It is also possible to check properties for a given class like for example if a class is a primitive, or an instance:

```
stringGetClass.isInstance( "dani" );
stringGetClass.isPrimitive();
```

It is also possible to create new instances of a given class using the method `java.lang.Class.newInstance()` passing the right arguments:

```
String newInstanceStringClass = stringclass.newInstance();
String otherInstance = (String)Class.forName( "java.lang.String" ).newInstance();
```

The `java.lang.Class.newInstance()` method can be used only when the class contains a public default constructor or a constructor without arguments, if this is not the case, this method cannot be used. In these cases where the `java.lang.Class.newInstance()` method cannot be used the solution is to retrieve a proper constructor at runtime and create an instance using this constructor with the arguments that it is expecting. We will see in the chapter related to constructors.

Chapter 6

Interfaces

Interfaces are elements that cannot be instantiated and that contain the exposed methods that should be implemented by their subclasses. Related to reflection there is nothing special regarding interfaces.

Interfaces can be accessed like a class using their qualified name. All methods available for classes are available for interfaces as well. Here is an example of how to access interface class information at runtime:

```
// can be accessed like a class
System.out.println( "interface name: " + InterfaceExample.class.getName() );
```

Assuming that the `InterfaceExample` element is an interface.

One obvious difference between classes and interfaces is that interfaces cannot be instantiated using reflection via the `newInstance()` method:

```
// cannot be instantiated:  java.lang.InstantiationException
InterfaceExample.class.newInstance();
```

The snippet above will throw an `InstantiationException` at runtime. At compile time no error appears.

Chapter 7

Enums

Enums are special java types that allow variables to be a set of constants. These constants are predefined in the enum declaration:

```
enum ExampleEnum
{
        ONE, TWO, THREE, FOUR
};
```

Java contains several enums specific methods:

- `java.lang.Class.isEnum()`: Returns true if the element is of the type enum. False otherwise
- `java.lang.Class.getEnumConstants()`: Gets all constants for the given element (which is an enum). In case the element is not an enum an exception is thrown.
- `java.lang.reflect.Field.isEnumConstant()`: Returns true in case the field used is an enum constant. False otherwise. Only applicable to fields.

We are going to see an example of how to use the main enum methods related to reflection. First of all we create an instance of the enum:

```
ExampleEnum value = ExampleEnum.FOUR;
```

We can check if the element is an enum using the method `isEnum()`:

```
System.out.println( "isEnum " + value.getClass().isEnum() );
```

In order to retrieve all the enum constants we can do something like the following using the method `getEnumConstants()`:

```
ExampleEnum[] enumConstants = value.getClass().getEnumConstants();
for( ExampleEnum exampleEnum : enumConstants )
{
        System.out.println( "enum constant " + exampleEnum );
}
```

Finally we can check how to use the field related method `isEnumConstants()`. First we retrieve all declared fields for the given class (we will see more in detail in the next chapters all methods related reflection utilities) and after that we can check if the field is an enum constant or not:

```
Field[] flds = value.getClass().getDeclaredFields();
for( Field f : flds )
{
        // check for each field if it is an enum constant or not
        System.out.println( f.getName() + " " + f.isEnumConstant() );
}
```

The output of all these pieces of code will be something like the following:

```
isEnum true
enum constant ONE
enum constant TWO
enum constant THREE
enum constant FOUR
ONE true
TWO true
THREE true
FOUR true
ENUM$VALUES false
```

The string `ENUM$VALUES false` refers to the internal enum values field. For more information about enums and how to handle them, please visit http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

Chapter 8

Primitive types

In Java, there are a couple of types that are handled differently because of its nature and behavior: when we are talking about reflection, primitive types like int, float, double, etc. can be accessed and used almost like any other classes. Here are a couple of examples of how to use reflection when we are working with primitive types:

- It is possible to retrieve a class object from a primitive type as for any other non primitive type:

```
Class<Integer> intClass = int.class;
```

- But It is not possible to create new instances for primitive types using reflection:

```
Integer intInstance = intClass.newInstance();
```

- It is possible to check if a given class belongs to a primitive type or not using the method `java.lang.Class.isPrimitive()`:

```
System.out.println( "is primitive: " + intClass.isPrimitive() );
```

In this case an exception of the type `java.lang.InstantiationException` is going to be thrown.

Chapter 9

Fields

Class fields can be handled in runtime using reflection. Classes offer several methods to access their fields at runtime. The most important ones are:

`java.lang.Class.getDeclaredFields()`: It returns an array with all declared fields for the class. It returns all private fields as well. `java.lang.Class.getFields()`: It returns an array with all accessible fields for the class. `java.lang.Class.getField(String)`: It returns a field with the name passed as parameter. It throws an exception if the field does not exist or is not accessible. `java.lang.Class.getDeclaredFields()`: It returns a field with the given name, if the field does not exist it throws an exception. These methods return an array of elements (or a single one) of the type `java.lang.reflect.Field`. This class contains several interesting methods that can be used at runtime that allow a programmer to read the properties and the values of the specific field.

Here is a class that uses this functionality:

```
String stringer = "this is a String called stringer";

Class<? extends String> stringGetClass = stringer.getClass();

Class<String> stringclass = String.class;

Field[] fields = stringclass.getDeclaredFields();

for( Field field : fields )
{
System.out.println( "************************" );
System.out.println( "Name: " + field.getName() );
System.out.println( "Type: " + field.getType() );

// values
if( field.isAccessible() )
{
        System.out.println( "Get: " + field.get( stringer ) );
        // depending on the type we can access the fields using these methods
        // System.out.println( "Get boolean: " + field.getBoolean( stringer ) );
        // System.out.println( "Get short: " + field.getShort( stringer ) );
        // ...
}
System.out.println( "Modifiers:" + field.getModifiers() );
System.out.println( "isAccesible: " + field.isAccessible() );

}

// stringclass.getField( "hashCode" );//exception

Field fieldHashCode = stringclass.getDeclaredField( "hash" );// all fields can be accessed ↩
    this way
```

```
// fieldHashCode.get( stringer ); // this produces an java.lang.IllegalAccessException

// we change the visibility
fieldHashCode.setAccessible( true );

// and we can access it
Object value = fieldHashCode.get( stringer );
int valueInt = fieldHashCode.getInt( stringer );
System.out.println( value );

System.out.println( valueInt );
```

In the snippet shown above you can see that the `Field` contains several methods to get the values of a given field like `get()` or type specific ones like `getInt()`. We also can see in the pasted code how we can change the way the visibility of a given field by using the method `setAccessible()`.

This is not always possible and under specific conditions and environments may be prevented. However this allows us to make a private field accessible and access its value and properties via reflection. This is very useful in testing frameworks like `Mockito` or `PowerMock`.

The output or the program would be:

```
*************************
Name: value
Type: class [C
Modifiers:18
isAccesible: false
*************************
Name: hash
Type: int
Modifiers:2
isAccesible: false
*************************
Name: serialVersionUID
Type: long
Modifiers:26
isAccesible: false
*************************
Name: serialPersistentFields
Type: class [Ljava.io.ObjectStreamField;
Modifiers:26
isAccesible: false
*************************
Name: CASE_INSENSITIVE_ORDER
Type: interface java.util.Comparator
Modifiers:25
isAccesible: false
0
0
```

Chapter 10

Methods

In order to retrieve all visible methods for a given class we can do the following:

```
Class<String> stringclass = String.class;
Method[] methods = stringclass.getMethods();
```

Using the method `java.lang.Class.getMethods()` all visible or accessible methods for a given class are retrieved. We can also retrieve an specific method using its name and the type of the arguments he is expecting to receive, as an example:

```
Method methodIndexOf = stringclass.getMethod( "indexOf", String.class );
```

For a given method (an instance of the type `java.lang.reflect.Method`), we can access all its properties. The following snippet shows a couple of them like name, default values, return type, modifiers, parameters, parameter types or the exceptions thrown, we can also check if a method is accessible or not:

```
// All methods for the String class
for( Method method : methods )
{
System.out.println( "***************************************************" );
System.out.println( "name: " + method.getName() );
System.out.println( "defaultValue: " + method.getDefaultValue() );

System.out.println( "generic return type: " + method.getGenericReturnType() );
System.out.println( "return type: " + method.getReturnType() );

System.out.println( "modifiers: " + method.getModifiers() );

// Parameters
Parameter[] parameters = method.getParameters();
System.out.println( parameters.length + " parameters:" );
// also method.getParameterCount() is possible
for( Parameter parameter : parameters )
{
        System.out.println( "parameter name: " + parameter.getName() );
        System.out.println( "parameter type: " + parameter.getType() );
}
Class<?>[] parameterTypes = method.getParameterTypes();
System.out.println( parameterTypes.length + " parameters:" );
for( Class<?> parameterType : parameterTypes )
{
        System.out.println( "parameter type name: " + parameterType.getName() );
}

// Exceptions
Class<?>[] exceptionTypes = method.getExceptionTypes();
```

```
System.out.println( exceptionTypes.length + " exception types: " );
for( Class<?> exceptionType : exceptionTypes )
{
        System.out.println( "exception name " + exceptionType.getName() );
}

System.out.println( "is accesible: " + method.isAccessible() );
System.out.println( "is varArgs: " + method.isVarArgs() );

}
```

It is also possible to instantiate given methods for specific objects passing the arguments that we want, we should assure that the amount and type of the arguments is correct:

```
Object indexOf = methodIndexOf.invoke( stringer, "called" );
```

This last feature is very interesting when we want to execute specific methods at runtime under special circumstances. Also in the creation of Invocation handlers for dynamic proxies is very useful, we will see this point at the end of the tutorial.

Chapter 11

Constructors

Constructors can be used via reflection as well. Like other class methods they can be retrieved in runtime and several properties can be analyzed and checked like the accessibility, the number of parameters, their types, etc.

In order to retrieve all visible constructors from a class, we can do something like:

```
// get all visible constructors
Constructor<?>[] constructors = stringGetClass.getConstructors();
```

In the snippet above we are retrieving all visible constructors. If we want to get all the constructors, including the private ones, we can do something like:

```
//all constructors
Constructor<?>[] declaredConstructors = stringclass.getDeclaredConstructors();
```

General information about constructors such as parameters, types, names, visibility, annotations associated, etc. can be retrieved in the following way:

```
for( Constructor<?> constructor : constructors )
{
        int numberParams = constructor.getParameterCount() ;
        System.out.println( "constructor " + constructor.getName() );
        System.out.println( "number of arguments " + numberParams);
        // public, private, etc.
        int modifiersConstructor = constructor.getModifiers();
        System.out.println( "modifiers " + modifiersConstructor );
        // array of parameters, more info in the methods section
        Parameter[] parameters = constructor.getParameters();
        // annotations array, more info in the annotations section
        Annotation[] annotations = constructor.getAnnotations();

}
```

Constructors can also be used to create new instances. This may be very useful in order to access private or not visible constructors. This should be done only under very special circumstances and depending on the system where the application is running may not work because of security reasons as explained at the beginning of this tutorial.

In order to create a new instance of a class using a specific constructor we can do something like the following:

```
// can be used to create new instances (no params in this case)
String danibuizaString = (String)constructor.newInstance(  );
```

In has to be taken into consideration that the amount of parameters and their type should match the constructor instance ones. Also the accessibility of the constructor has to be set to true in order to invoke it (if it was not accesible). This can be done in the same way as we did for class methods and fields.

Chapter 12

Getters and Setters

Getters and setters are not different to any other class method inside a class. The main difference is that they are a standard way to access private fields.

Here is a description of both:

- Getters are used to retrieve the value of a private field inside a class. Its name starts with `get` and ends with the name of the property in camel case. They do not receive any parameter and their return type is the same than the property that they are returning. They are public.
- Setters are used to modify the value of a private field inside a class. Its name starts with "set" and ends with the name of the property in camel case. They receive one parameters of the same type than the property that they are modifying and they do not return any value (void). They are public.

For example, for the private property `private int count;` we can have the getter and setter methods:

```
public int getCount(){
        return this.count;
}

public void setCount(int count){
        this.count = count;
}
```

Following these standards we can use reflection to access (read and modify) at runtime all the private properties of a class exposed via getters and setters. This mechanism is used by several known libraries like Spring Framework or Hibernate, where they expect classes to expose their properties using these kinds of methods.

Here is an example of how to use getters and setters using reflection:

```
Car car = new Car( "vw touran", "2010", "12000" );

Method[] methods = car.getClass().getDeclaredMethods();

// all getters, original values
for( Method method : methods )
{
        if( method.getName().startsWith( "get" ) )
        {
                System.out.println( method.invoke( car ) );
        }
}

// setting values
for( Method method : methods )
```

```
{

        if( method.getName().startsWith( "set" ) )
        {
                method.invoke( car, "destroyed" );
        }
}

// get new values
for( Method method : methods )
{
        if( method.getName().startsWith( "get" ) )
        {
                System.out.println( method.invoke( car ) );
        }
}
```

Where the class `Car`. looks like the following:

```
public class Car
{

        private String name;
        private Object price;
        private Object year;

        public Car( String name, String year, String price )
        {
        this.name = name;
        this.price = price;
        this.year = year;
        }

        public String getName()
        {
        return name;
        }

        public void setName( String name )
        {
        this.name = name;
        }

        public Object getPrice()
        {
        return price;
        }

        public void setPrice( Object price )
        {
        this.price = price;
        }

        public Object getYear()
        {
        return year;
        }

        public void setYear( Object year )
        {
        this.year = year;
        }
```

```
}
```

The output will be something like:

```
        vw touran
        2010
        12000
        destroyed
        destroyed
        destroyed
```

Chapter 13

Static elements

Static classes, methods and fields behave completely different than instance ones. The main reason is that they do not need to be instantiated or created before they are invoked. They can be used without previous instantiation.

This fact changes everything: static methods can be invoked without instantiating their container classes, static class fields are stateless (so thread safe), static elements are very useful in order to create singletons and factories. Summarizing, static elements are a very important mechanism in Java.

In this chapter we are going to show the main differences between static and instance elements in relation to reflection: How to create static elements at runtime and how to invoke them.

For example, for the next static inline class:

```
public class StaticReflection
{

        static class StaticExample
        {
        int counter;
        }
...
```

In order to retrieve static inline classes we have the following options:

```
// 1 access static class
System.out.println( "directly " + StaticExample.class.getName() );
```

```
//2 using for name directly throws an exception
Class<?> forname = Class.forName("com.danibuiza.javacodegeeks.reflection.StaticReflection. ↩
    StaticExample" );
```

```
//3 using $ would work but is not that nice
Class<?> forname = Class.forName("com.danibuiza.javacodegeeks.reflection. ↩
    StaticReflection$StaticExample" );
```

```
// 4 another way iterating through all classes declared inside this class
Class<?>[] classes = StaticReflection.class.getDeclaredClasses();
for( Class<?> class1 : classes )
{
        System.out.println( "iterating through declared classes " + class1.getName() );
}
```

The main difference is that the class is contained inside another class; this has nothing to do with reflection but with the nature of inline classes.

In order to get static methods from a class, there are no differences with the access to instance ones (this applies to fields as well):

```
// access static methods in the same way as instance ones
Method mathMethod = Math.class.getDeclaredMethod( "round", double.class );
```

In order to invoke static methods or fields we do not need to create or specify an instance of the class, since the method (or the field) belongs to the class itself, not to a single instance:

```
// methods: object instance passed can be null since method is static
Object result = mathMethod.invoke( null, new Double( 12.4 ) );

// static field access, instance can be null
Field counterField = Counter.class.getDeclaredField( "counter" );
System.out.println( counterField.get( null ) );
```

Chapter 14

Arrays

The class `java.lang.reflect.Array` offers several functionalities for handling arrays; it includes various static reflective methods:

`java.lang.reflect.Array.newInstance(Class<?>, int)`: Creates a new instance of an array of the type passed as parameter with the length given in the second argument. Is similar to the method with the same name in the `java.lang.Class` class but this one contains parameters that allows the programmer to set the type of the array and its length. `java.lang.reflect.Array.set(Object, int, Object)`: Sets an element (passed index) of the given array with the object passed as argument. `java.lang.reflect.Array.getLength(Object)`: Returns the length of the array as int. `java.lang.reflect.Array.get(Object, int)`: Retrieves the element of the array positioned in the pased index. Returns an object. `java.lang.reflect.Array.getInt(Object, int)`: Similar method for the primitive type int. Returns an int. There are methods available for all primitive types.

Here is an example of how we can use all these methods:

```
// using the Array class it is possible to create new arrays passing the type and the  ↩
    length via reflection
String[] strArrayOne = (String[])Array.newInstance( String.class, 10 );

// it contains utility methods for setting values
Array.set( strArrayOne, 0, "member0" );
Array.set( strArrayOne, 1, "member1" );
Array.set( strArrayOne, 9, "member9" );

// and for getting values as well
System.out.println( "strArrayOne[0] : " + Array.get( strArrayOne, 0 ) );
System.out.println( "strArrayOne[1] : " + Array.get( strArrayOne, 1 ) );
System.out.println( "strArrayOne[3] (not initialized) : " + Array.get( strArrayOne, 3 ) );
System.out.println( "strArrayOne[9] : " + Array.get( strArrayOne, 9 ) );

// also methods to retrieve the lenght of the array
System.out.println( "lenght strArrayOne: " + Array.getLength( strArrayOne ) );

// primitive types work as well
int[] intArrayOne = (int[])Array.newInstance( int.class, 10 );

Array.set( intArrayOne, 0, 1 );
Array.set( intArrayOne, 1, 2 );
Array.set( intArrayOne, 9, 10 );

// and specific getters and setters for primitive types
for( int i = 0; i < Array.getLength( intArrayOne ); ++i )
{
        System.out.println( "intArrayOne[" + i + "] : " + Array.getInt( intArrayOne, i ) );
}
```

The output of the program above would be:

```
strArrayOne[0] : member0
strArrayOne[1] : member1
strArrayOne[3] (not initialized) : null
strArrayOne[9] : member9
lenght strArrayOne: 10
intArrayOne[0] : 1
intArrayOne[1] : 2
intArrayOne[2] : 0
intArrayOne[3] : 0
intArrayOne[4] : 0
intArrayOne[5] : 0
intArrayOne[6] : 0
intArrayOne[7] : 0
intArrayOne[8] : 0
intArrayOne[9] : 10
```

The class contains a method that permits to check if an instance is an array or not, obviously, this method is available for all classes in Java. It is called `java.lang.Class.isArray()` and can be used in the following way:

```
// retrieve the class from an instance
Class<String[]> stringArrayClassUsingInstance = String[].class;
System.out.println( "stringArrayClassUsingInstance is array: " +  ↩
    stringArrayClassUsingInstance.isArray() );

// using class for name and passing [I
Class<?> intArrayUsingClassForName = Class.forName( "[I" );
System.out.println( "intArrayUsingClassForName is array: " + intArrayUsingClassForName. ↩
    isArray() );

// or [Ljava.lang.String
Class<?> stringArrayClassUsingClassForName = Class.forName( "[Ljava.lang.String;" );
System.out.println( "stringArrayClassUsingClassForName is array: "
        + stringArrayClassUsingClassForName.isArray() );

// this has no much sense in my opinion since we are creating an array at runtime and
// getting the class to create a new one...
Class<? extends Object> stringArrayClassUsingDoubleLoop = Array.newInstance( String.class,  ↩
    0 ).getClass();
System.out.println( "stringArrayClassUsingClassForName is array: " +  ↩
    stringArrayClassUsingDoubleLoop.isArray() );
```

The output of the snippet above would be:

```
stringArrayClassUsingInstance is array: true
intArrayUsingClassForName is array: true
stringArrayClassUsingClassForName is array: true
stringArrayClassUsingClassForName is array: true
```

Chapter 15

Collections

Collections do not have many remarkable specific features related to reflection. Here is an example of how we can handle collection based elements. As already said, there are not many differences to any other Java type.

The following method prints all the class names of all elements of a collection, it previously checks if the passed element is a collection instance or not:

```
private static void reflectionCollections( Object ref )
{
//check is collection
        if( ref instanceof Collection )
        {
                System.out.println( "A collection:  " + ref.getClass().getName() );
                @SuppressWarnings( "rawtypes" )
                // not nice
                Iterator items = ( (Collection)ref ).iterator();
                while( items != null &amp;amp;&amp;amp; items.hasNext() )
                {
                        Object item = items.next();
                        System.out.println( "Element of the collection:  " + item.getClass ↩
                            ().getName() );
                }
        }
        else
        {
                System.out.println( "Not a collection:  " + ref.getClass().getName() );
        }
}
```

In the code shown, reflection is just used to check the instance type passed as parameter and to retrieve the class names of the elements inside the collection. We can call this method in the following way using different elements (some are collection based, some not):

```
Map<String, String> map = new HashMap<String, String>();
map.put( "1", "a" );
map.put( "2", "b" );
map.put( "3", "c" );
map.put( "4", "d" );

reflectionCollections( map );
reflectionCollections( map.keySet() );
reflectionCollections( map.values() );

List<String> list = new ArrayList<String>();
list.add( "10" );
```

```
list.add( "20" );
list.add( "30" );
list.add( "40" );

reflectionCollections( list );
reflectionCollections( "this is an string" );
```

And we would get the following output:

```
Not a collection:  java.util.HashMap
A collection:  java.util.HashMap$KeySet
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
A collection:  java.util.HashMap$Values
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
A collection:  java.util.ArrayList
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Element of the collection:  java.lang.String
Not a collection:  java.lang.String
```

Chapter 16

Annotations

All annotations of a class, package, method, field, etc. can be retrieved using reflection. Annotations can be evaluated in runtime for each element and their values can be retrieved. The following snippet shows how to retrieve all annotations for a given class and how to print out their properties and values:

```
Class<ReflectableClass> object = ReflectableClass.class;
// Retrieve all annotations from the class
Annotation[] annotations = object.getAnnotations();
for( Annotation annotation : annotations )
{
System.out.println( annotation );
}
```

The following example explains how to check if an element (field, method, class...) is marked with an specific annotation or not:

```
// Checks if an annotation is present
if( object.isAnnotationPresent( Reflectable.class ) )
{
        // Gets the desired annotation
        Annotation annotation = object.getAnnotation( Reflectable.class );

        System.out.println( annotation + " present in class " +                          ↩
            object.getClass() );// java.lang.class
        System.out.println( annotation + " present in class " +         object.getTypeName ↩
            () );//       com.danibuiza.javacodegeeks.reflection.ReflectableClass

}
```

These snippets may be applicable to methods, fields and all elements that can be annotated.

You can find a very good article and extensive tutorial about Java annotations with several examples related to Java reflection in the following link: http://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html.

Chapter 17

Generics

Generics were introduced in Java in the update 5 and since then are a very important feature that helps to maintain the code clean and more usable. Parameterized elements are not different than other elements in Java, so all the topics explained in this tutorial apply to these elements as well.

Java contains specific reflection mechanisms to handle generics as well. It is possible to check at runtime if a specific element (class, method or field) is parameterized or not. It is also possible to retrieve the parameters type using reflection as well.

Here is an example of how we can do this:

```
Method getInternalListMethod = GenericsClass.class.getMethod( "getInternalList", null );

// we get the return type
Type getInternalListMethodGenericReturnType = getInternalListMethod.getGenericReturnType();

// we can check if the return type is parameterized (using ParameterizedType)
if( getInternalListMethodGenericReturnType instanceof ParameterizedType )
{
        ParameterizedType parameterizedType = (ParameterizedType) ↩
            getInternalListMethodGenericReturnType;
        // we get the type of the arguments for the parameterized type
        Type[] typeArguments = parameterizedType.getActualTypeArguments();
        for( Type typeArgument : typeArguments )
        {
                // warning not nice
                // we can work with that now
                Class typeClass = (Class)typeArgument;
                System.out.println( "typeArgument = " + typeArgument );
                System.out.println( "typeClass = " + typeClass );
        }
}
```

In the code listed above we can see that the main class related to reflection and generics is `java.lang.reflect.ParameterizedType` and one of its most important methods `java.lang.reflect.ParameterizedType.getActualTypeArguments()`.

Chapter 18

Class Loaders

Elements in Java are loaded using class loaders. Class loaders can be implemented by implementing the abstract class `ClassLoader`. They provide functionalities to load classes using their names as parameters. A typical mechanism for loading classes is to find in the class path the file that belongs to the given name and open it converting it into a Java class. Every system (JVM) has a default one.

In order to retrieve the systems default class loader we can do the following:

```
ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
```

A programmer can also get the specific class loader used for loading a given class or instance. If nothing else is specified or configured, the default one is used. For example:

```
ClassLoader classClassLoader = ReflectableClass.class.getClassLoader();
```

Using a class loader we can load classes at runtime passing as parameter the qualified name of a class, this name can be generated dynamically:

```
Class<?> reflectableClassInstanceLoaded = systemClassLoader
                .loadClass( "com.danibuiza.javacodegeeks.reflection.ReflectableClass" );
```

Another possibility is to to this using `Class.forName()` method and specify the class loader that we want to use as parameter:

```
Class<?> reflectableClassInstanceForName = Class
                .forName( "com.danibuiza.javacodegeeks.reflection.ReflectableClass", true,  ↩
                    systemClassLoader );
```

For more information about class loaders in Java please visit the official API where you can find more information:

https://docs.oracle.com/javase/7/docs/api/java/lansystemClassLoaderg/reflect/InvocationHandler.html

Chapter 19

Dynamic Proxies

Dynamic proxies are classes that implement a list of interfaces specified at runtime, that is, specified when the class is created. We may have proxy interfaces, classes and instances as for any other java element.

Each instance of a proxy class contains an invocation handler object. An invocation handler object is an instance of a class that implements the interface `InvocationHandler`.

When a proxy instance is used to invoke a given method, this one will be forwarded to the invoke method of the proxy instance invocation handler. The method (using reflection) and the expected arguments are passed to the invocation handler. The result of the original invocation is the returned result from the invocation handler. In order to explain that in an easier way we are going to show an example.

For more information about java dynamic proxies please visit https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html. In this tutorial we are going to show how to create a simple proxy class and how to redirect methods through the proxy and execute the desired actions using reflection.

The main example program would look like:

```
// an invocation handler for our needs
InvocationHandler myHandler = new HandlerImpl();

// we can create dynamic proxy clases using the Proxy class
InformationInterface proxy = InformationInterface)Proxy.newProxyInstance( ↩
    InformationInterface.class.getClassLoader(),  new Class[] {InformationInterface.class }, ↩
      myHandler);

// all calls to the proxy will be passed to the handler -> the handler  implementation can  ↩
    be
// decided on runtime as well
System.out.println( proxy.getInfo() );
```

We can see that we have 3 main topics to explain here. The first one is the invocation handler; we need to create an instance of an invocation handler and implement there the actions that we should take when methods of the proxy are called and forwarded to it.

A new class implementing `java.lang.reflect.InvocationHandler` is created and the method `com.danibuiza.javacodegeeks.reflection.HandlerImpl.invoke(Object, Method, Object[])` is overriden:

```
public class HandlerImpl implements InvocationHandler
{

    @Override
    public Object invoke( Object obj, Method method, Object[] arguments ) throws Throwable
    {
        System.out.println( "using proxy " + obj.getClass().getName() );
        System.out.println( "method " + method.getName() + " from interface " + method. ↩
            getDeclaringClass().getName() );
```

```
        // we can check dynamically the interface and load the implementation that we want
        if( method.getDeclaringClass().getName().equals( "com.danibuiza.javacodegeeks. ↩
            reflection.InformationInterface" ) )
        {
            InformationClass informationImpl = InformationClass.class.newInstance();
            return method.invoke( informationImpl, arguments );
        }

        return null;
    }
}
```

Using reflection we can take the actions that we want in the `invoke()` method.

The next step is to create the proxy itself. We do this by using the invocation handler instance created before and the class (and its interface) that we want to proxy.

```
InformationInterface proxy = (InformationInterface)Proxy.newProxyInstance(
InformationInterface.class.getClassLoader(), new Class[] {
InformationInterface.class }, myHandler );
```

Once the proxy and invocation handler are created, we can start to use them. All methods of the proxy are going to be forwarded now to the invocation handler and will be handled there.

```
//the getInfo() method will be forwarded via the invocation handler
System.out.println( proxy.getInfo() );
```

This is a very simple example of dynamic proxies in Java but it explains how reflection can be used in this scenario.

Chapter 20

Java 8 Reflection features

There are not many changes and enhancements in Java 8 related to reflection. The most important one is the possibility to retrieve the method parameters with their proper names. Until now it was only possible to get the parameters with the names "arg0", "arg1", etc. This was kind of confusing and not that clean to work with.

Here is an example:

```
Class<String> stringClass = String.class;
for( Method methodStringClass : stringClass.getDeclaredMethods() )
{
        System.out.println( "method " + methodStringClass.getName() );
        for( Parameter paramMethodStringClass : methodStringClass.getParameters() )
        {
                // arg0, arg1, etc because the eclipse compiling tool (different
                    than javac) does
                // not support -parameters option yet
                System.out.println( " parameter name " + paramMethodStringClass.getName() )
                    ;
                System.out.println( " parameter type " + paramMethodStringClass.getType() )
                    ;
        }
}
```

The mentioned method is `java.lang.reflect.Executable.getParameters()`.

It has to be mentioned that in order to get this feature working you need to compile your application with the javac argument -parameters:

```
javac -parameters <class>
```

Or using maven you should pass this parameter in the pom.xml file:

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
 <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <compilerArgument>-parameters</compilerArgument>
 </configuration>
</plugin>
```

For those using Eclipse, it has to be mentioned that Eclipse does not use javac as compiler. At the moment of writing this article, the Eclipse compiler did not support this feature, so you need to compile your sources yourself using maven, ant or any other tool.

For the ones with big interest in Java 8 Lambdas here is a white paper with extensive information about the translation of java 8 Lambdas into "normal" java, reflection is involved.

Another interesting link about the use of the `getParameters()` in Java 8: https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html.

Chapter 21

Summary

So that's it. In this tutorial we explained in detail what is software reflection, how it can be used in Java and under what circumstances should and should not be used.

Reflection is a mechanism that allows programmers to analyze, modify and invoke code at runtime without previously knowing exactly what code they are executing.

Different use cases and applications can fit for using reflection like auto completion features in IDEs, unit testing processors, dependency injection or code analysis tools. It can also be used in normal applications for specific tasks but it should be avoided if possible since the performance and the security of the code will be affected. The maintenance effort will be increased in case the source code contains a lot of reflection.

Reflection in Java can be used in classes, methods, packages, fields, interfaces, annotations, etc. Using reflection it is possible to create new instance of any kind of object at runtime and to check its properties and values.

Since Java 8 the method `getParameters()` has been slightly modified and allows programmers to retrieve the actual names of method parameters, we explained this in the Java 8 related chapter.

The intention of this tutorial is to give an introduction and a deep overview of all the possibilities that Java offers in terms of reflection. For more specific tasks that may not be covered here, please refer to the links and resources section.

All examples shown in this tutorial can be downloaded in the download section.