Reflection API

Java Reflection is a process of examining or modifying the capabilities of a class at run time. Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The **java.lang** and **java.lang.reflect** packages provide classes for java reflection.

Java Reflection Example

Here is a quick Java Reflection example to show you what using reflection looks like:

```
Method[] methods = MyObject.class.getMethods();

for(Method method : methods){
    System.out.println("method = " + method.getName());
}
```

This example obtains the `Class` object from the class called `MyObject`. Using the class object the example gets a list of the methods in that class, iterates the methods and print out their names.

Exactly how all this works is explained in further detail throughout the rest of this tutorial (in other texts).

Java Class Object

When using Java reflection the starting point is often a `Class` object representing some Java class you want to inspect via reflection. For instance, to obtain the `Class` object for a class named `MyObject` you could write:

```
Class myObjectClass = MyObject.class;
```

Now you have a reference to the `Class` object for the `MyObject` class.

The `Class` object is described in more detail in the **Java Reflection Class tutorial**.

Fields

Once you have a reference to the `Class` object representing some class, you can see what fields that class contains. Here is an example of accessing fields of a Java class:

```
Class myObjectClass = MyObject.class;

Field[] fields   = myObjectClass.getFields();
```

With a reference to a Java reflection `Field` instance you can start inspecting the field. You can read its name, access modifiers etc. You can read more about what you can do with a Java reflection `Field`instance in my **Java Reflection Field** tutorial.

Constructors

Using Java Reflection it is possible find out what **constructors** a given Java class contains and what parameters they take etc. I have explained how to do that in my **Java Reflection - Constructor** tutorial.

Methods

You can also see what methods a given class has from its `Class` object. Here is an example of accessing the methods a given class via Java reflection:

```
Class myObjectClass = MyObject.class;

Method[] methods = myObjectClass.getMethods();
```

Once you have references to the a Java reflection `Method` instance you can start inspecting it. You can read the method's name, what parameters it takes, its return type etc. You can read more about what you can do with a Java reflection `Method` instance in my **Java Reflection Method** tutorial.

Getters and Setters

You can also use Java reflection to find out what getter and setter methods a class has. This is covered in more detail in the tutorial about **Java Reflection - Getters and Setters** tutorial.

Private Fields and Methods

You can even access private fields and methods via Java reflection - even from outside the class that owns the private field or method. I have explained how to do that in my **Java Reflection - Private Fields and Methods** .

Java Reflection – Classes

Using Java Reflection you can inspect Java classes at runtime. Inspecting classes is often the first thing you do when using Reflection. From the classes you can obtain information about

- **Class Name**
- **Class Modifies (public, private, synchronized etc.)**
- **Package Info**
- **Superclass**
- **Implemented Interfaces**
- **Constructors**
- **Methods**
- **Fields**
- **Annotations**

plus a lot more information related to Java classes. For a full list you should consult the **JavaDoc for java.lang.Class**. This text will briefly touch upon all accessing of the above mentioned information. Some of the topics will also be examined in greater detail in separate texts. For instance, this text will show you how to obtain all methods or a specific method, but a separate text will show you how to invoke that method, how to find the method matching a given set of arguments if more than one method exists with the same name, what exceptions are thrown from method invocation via reflection, how to spot a getter/setter etc. The purpose of this text is primarily to introduce the `Class` object and the information you can obtain from it.

The Class Object

Before you can do any inspection on a class you need to obtain its `java.lang.Class` object. All types in Java including the primitive types (int, long, float etc.) including arrays have an associated `Class` object.

- If you know the name of the class at compile time you can obtain a `Class` object like this:

```
Class myObjectClass = MyObject.class
```

- If you don't know the name at compile time, but have the class name as a string at runtime, you can do like this:

String className = ... //obtain class name as string at runtime,

Class class = Class.forName(className);

When using the `Class.forName()` method you must supply the fully qualified class name. That is the class name including all package names. For instance, if MyObject is located in package `com.jenkov.myapp` then the fully qualified class name is `com.jenkov.myapp.MyObject`

The `Class.forName()` method may throw a `ClassNotFoundException` if the class cannot be found on the classpath at runtime.

Class Name

From a `Class` object you can obtain its name in two versions. The fully qualified class name (including package name) is obtained using the `getName()` method like this:

```
Class aClass = ... //obtain Class object. See prev. section
String className = aClass.getName();
```

If you want the class name without the pacakge name you can obtain it using the `getSimpleName()` method, like this:

```
Class  aClass          = ... //obtain Class object. See prev. section
String simpleClassName = aClass.getSimpleName();
```

Modifiers

You can access the modifiers of a class via the `Class` object. The class modifiers are the keywords "public", "private", "static" etc. You obtain the class modifiers like this:

```
Class  aClass = ... //obtain Class object. See prev. section
int modifiers = aClass.getModifiers();
```

The modifiers are packed into an `int` where each modifier is a flag bit that is either set or cleared. You can check the modifiers using these methods in the class `java.lang.reflect.Modifier`:

```
Modifier.isAbstract(int modifiers)
Modifier.isFinal(int modifiers)
Modifier.isInterface(int modifiers)
Modifier.isNative(int modifiers)
Modifier.isPrivate(int modifiers)
Modifier.isProtected(int modifiers)
Modifier.isPublic(int modifiers)
Modifier.isStatic(int modifiers)
Modifier.isStrict(int modifiers)
Modifier.isSynchronized(int modifiers)
Modifier.isTransient(int modifiers)
```

```
    Modifier.isVolatile(int modifiers)
```

Package Info

You can obtain information about the package from a `Class` object like this:

```
Class  aClass = ... //obtain Class object. See prev. section
Package package = aClass.getPackage();
```

From the `Package` object you have access to information about the package like its name. You can also access information specified for this package in the `Manifest` file of the JAR file this package is located in on the classpath. For instance, you can specify package version numbers in the `Manifest` file. You can read more about the `Package` class here: **java.lang.Package**

Superclass

From the `Class` object you can access the superclass of the class. Here is how:

```
Class superclass = aClass.getSuperclass();
```

The superclass class object is a `Class` object like any other, so you can continue doing class reflection on that too.

Implemented Interfaces

It is possible to get a list of the interfaces implemented by a given class. Here is how:

```
Class  aClass = ... //obtain Class object. See prev. section
Class[] interfaces = aClass.getInterfaces();
```

A class can implement many interfaces. Therefore an array of `Class` is returned. Interfaces are also represented by `Class` objects in Java Reflection.

NOTE: Only the interfaces specifically declared implemented by a given class is returned. If a superclass of the class implements an interface, but the class doesn't specifically state that it also implements that interface, that interface will not be returned in the array. Even if the class in practice implements that interface, because the superclass does.

To get a complete list of the interfaces implemented by a given class you will have to consult both the class and its superclasses recursively.

Constructors

You can access the constructors of a class like this:

```
 Constructor[] constructors = aClass.getConstructors();
```

Constructors are covered in more detail in the text on **Constructors**.

Methods

You can access the methods of a class like this:

```
 Method[] method = aClass.getMethods();
```

Methods are covered in more detail in the text on **Methods**.

Fields

You can access the fields (member variables) of a class like this:

```
Field[] method = aClass.getFields();
```

Fields are covered in more detail in the text on **Fields**.

Annotations

You can access the class annotations of a class like this:

```
Annotation[] annotations = aClass.getAnnotations();
```

Annotations are covered in more detail in the text on **Annotations**.

Java Reflection – Constructors

Using Java Reflection you can inspect the constructors of classes and instantiate objects at runtime. This is done via the Java class `java.lang.reflect.Constructor`. This text will get into more detail about the Java `Constructor` object.

Obtaining Constructor Objects

The `Constructor` class is obtained from the `Class` object. Here is an example:

```
Class aClass = ...//obtain class object
Constructor[] constructors = aClass.getConstructors();
```

The `Constructor[]` array will have one `Constructor` instance for each public constructor declared in the class.

If you know the precise parameter types of the constructor you want to access, you can do so rather than obtain the array all constructors. This example returns the public constructor of the given class which takes a `String` as parameter:

```
Class aClass = ...//obtain class object
Constructor constructor =
        aClass.getConstructor(new Class[]{String.class});
```

If no constructor matches the given constructor arguments, in this case `String.class`, a `NoSuchMethodException` is thrown.

Constructor Parameters

You can read what parameters a given constructor takes like this:

```
Constructor constructor = ... // obtain constructor - see above
Class[] parameterTypes = constructor.getParameterTypes();
```

Instantiating Objects using Constructor Object

You can instantiate an object like this:

```
//get constructor that takes a String as argument
Constructor constructor = MyObject.class.getConstructor(String.class);

MyObject myObject = (MyObject)
      constructor.newInstance("constructor-arg1");
```

The Constructor.newInstance() method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the constructor you are invoking. In this case it was a constructor taking a String, so one String must be supplied.

Java Reflection – Methods

Obtaining Method Objects

The `Method` class is obtained from the `Class` object. Here is an example:

```
Class aClass = ...//obtain class object
Method[] methods = aClass.getMethods();
```

The `Method[]` array will have one `Method` instance for each public method declared in the class.

If you know the precise parameter types of the method you want to access, you can do so rather than obtain the array all methods. This example returns the public method named "doSomething", in the given class which takes a `String` as parameter:

```
Class  aClass = ...//obtain class object
Method method =
    aClass.getMethod("doSomething", new Class[]{String.class});
```

If no method matches the given method name and arguments, in this case `String.class`, a `NoSuchMethodException` is thrown.

If the method you are trying to access takes no parameters, pass `null` as the parameter type array, like this:

```
Class  aClass = ...//obtain class object
Method method =
    aClass.getMethod("doSomething", null);
```

Method Parameters and Return Types

You can read what parameters a given method takes like this:

```
Method method = ... // obtain method - see above
Class[] parameterTypes = method.getParameterTypes();
```

You can access the return type of a method like this:

```
Method method = ... // obtain method - see above
Class returnType = method.getReturnType();
```

Invoking Methods using Method Object

You can invoke a method like this:

```
//get method that takes a String as argument
Method method = MyObject.class.getMethod("doSomething", String.class);

Object returnValue = method.invoke(null, "parameter-value1");
```

The `null` parameter is the object you want to invoke the method on. If the method is static you supply `null`instead of an object instance. In this example, if `doSomething(String.class)` is not static, you need to supply a valid `MyObject` instance instead of `null`;

The `Method.invoke(Object target, Object ... parameters)` method takes an optional amount of parameters, but you must supply exactly one parameter per argument in the method you are invoking. In this case it was a method taking a `String`, so one `String` must be supplied.

Java Reflection - Getters and Setters

Using Java Reflection you can inspect the methods of classes and invoke them at runtime. This can be used to detect what getters and setters a given class has. You cannot ask for getters and setters explicitly, so you will have to scan through all the methods of a class and check if each method is a getter or setter.

First let's establish the rules that characterizes getters and setters:

- **Getter**
  A getter method have its name start with "get", take 0 parameters, and returns a value.

- **Setter**
  A setter method have its name start with "set", and takes 1 parameter.

Setters may or may not return a value. Some setters return void, some the value set, others the object the setter were called on for use in method chaining. Therefore you should make no assumptions about the return type of a setter.

Here is a code example that finds getter and setters of a class:

```
public static void printGettersSetters(Class aClass){
  Method[] methods = aClass.getMethods();

  for(Method method : methods){
    if(isGetter(method)) System.out.println("getter: " + method);
    if(isSetter(method)) System.out.println("setter: " + method);
  }
}

public static boolean isGetter(Method method){
  if(!method.getName().startsWith("get"))      return false;
  if(method.getParameterTypes().length != 0)   return false;
  if(void.class.equals(method.getReturnType()) return false;
  return true;
}

public static boolean isSetter(Method method){
  if(!method.getName().startsWith("set")) return false;
  if(method.getParameterTypes().length != 1) return false;
  return true;
}
```

Java Reflection - Private Fields and Methods

Accessing Private Fields

To access a private field you will need to call the `Class.getDeclaredField(String name)` or `Class.getDeclaredFields()` method. The methods `Class.getField(String name)` and `Class.getFields()` methods only return public fields, so they won't work. Here is a simple example of a class with a private field, and below that the code to access that field via Java Reflection:

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
    this.privateString = privateString;
  }
}
PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.
            getDeclaredField("privateString");

privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);
System.out.println("fieldValue = " + fieldValue);
```

This code example will print out the text "fieldValue = The Private Value", which is the value of the private field `privateString` of the `PrivateObject` instance created at the beginning of the code sample.

Notice the use of the method `PrivateObject.class.getDeclaredField("privateString")`. It is this method call that returns the private field. This method only returns fields declared in that particular class, not fields declared in any superclasses.

Notice the line in bold too. By calling `Field.setAcessible(true)` you turn off the access checks for this particular `Field` instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the field using normal code. The compiler won't allow it.

Accessing Private Methods

To access a private method you will need to call the `Class.getDeclaredMethod(String name, Class[] parameterTypes)` or `Class.getDeclaredMethods()` method. The methods `Class.getMethod(String name, Class[] parameterTypes)` and `Class.getMethods()` methods only return public methods, so they won't work. Here is a simple example of a class with a private method, and below that the code to access that method via Java Reflection:

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
    this.privateString = privateString;
  }

  private String getPrivateString(){
```

```
    return this.privateString;
  }
}
PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.
        getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)
        privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);
```

This code example will print out the text "returnValue = The Private Value", which is the value returned by the method `getPrivateString()` when invoked on the `PrivateObject` instance created at the beginning of the code sample.

Notice the use of the method `PrivateObject.class.getDeclaredMethod("privateString")`. It is this method call that returns the private method. This method only returns methods declared in that particular class, not methods declared in any superclasses.

Notice the line in bold too. By calling `Method.setAcessible(true)` you turn off the access checks for this particular `Method` instance, for reflection only. Now you can access it even if it is private, protected or package scope, even if the caller is not part of those scopes. You still can't access the method using normal code. The compiler won't allow it.

Q] Can we call a private method from outside the class? If so then the property of Private access modifier is lost – so why are we doing that?

Ans: Yes, we can do that by the reflection API. By setting the accessability true.

```
public class Basics {

static void main (String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException, NoSuchMethodException,
SecurityException, IllegalArgumentException, InvocationTargetException {
            //Obtaining the Class Object
            Class c=Class.forName("com.basics.reflectionAPI.Test");

            //Creating the object of the Class c[Test]
            Test test=(Test) c.newInstance();

            //Creating object of Method Class
            Method method=c.getDeclaredMethod("show", null);
            method.setAccessible(true);   //To make the private method Accessable
            method.invoke(test, null);    //Calling the method
      }
}

class Test{
      private void show() {
            System.out.println("Inside private method show()");
      }
}
```