Published on **ONJava.com** (http://www.onjava.com/)

See this if you're having trouble printing code examples

Reflections on Java Reflection

by Russ Olsen

03/15/2007

In ordinary life, a reflection is what you see when you look in the mirror. In the world of programming, *reflection* is what you call it when a program looks at and possibly even modifies its own structure. The Java reflection API allows you to do exactly that by giving you a window into the fundamental features of the language –– classes and fields and methods –– via an ordinary Java API. Understanding reflection will help you understand the tools that you use every day. How does Eclipse manage to do all that helpful auto–completion of method names? How does Tomcat go from a class name in a *web.xml* file to a running servlet fielding web requests? And how does Spring do all of that magic dependency injection stuff? In your own programs, you can use reflection to write code that is more flexible and dynamic: with reflection your program can cope gracefully with classes it has never met before.

Conjuring Up A Class

As I said, the fundamental idea of reflection is to give you an ordinary Java API into the inner workings of your program. Since the most basic idea in Java is the class (try to write a Java program without one), a good place to start is by looking at the `Class` class. No, that is not a typing stutter: you can actually get hold of an object which contains everything you might ever need to know about a Java class. That object will have a type of `Class`. Once you have a `Class` instance, you can extract all kinds of information about the class from it, including the name of the class, whether it is public or abstract or final, and even its super–class.

Enough theory: let's turn our reflection microscope on the following, very simple `Employee` class:

```
package com.russolsen.reflect;

public class Employee
{
   public String _firstName;
   public String _lastName;
   private int _salary;

      public Employee()
```

```
    {
       this( "John", "Smith", 50000);
    }

    public Employee(String fn, String ln, int salary)
    {
       _firstName = fn;
       _lastName = ln;
       _salary = salary;
    }

       public int getSalary()
    {
       return _salary;
    }

    public void setSalary(int salary)
    {
       _salary = salary;
    }

    public String toString()
    {
       return "Employee: " + _firstName +  " "
              + _lastName + " " + _salary;
    }

 }
```

The easiest way to get a class object is to just ask an object instance for its class with the `getClass` method. The code below creates a `Employee` instance, asks for its class, and prints out various bits of information about the class:

```
package com.russolsen.reflect;

import java.lang.reflect.Modifier;

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

      Employee employee = new Employee();

      Class klass = employee.getClass();

      System.out.println( "Class name: " + klass.getName());
      System.out.println(
            "Class super class: " + klass.getSuperclass());

      int mods = klass.getModifiers();
      System.out.println(
            "Class is public: " + Modifier.isPublic(mods));
      System.out.println(
            "Class is final: " +  Modifier.isFinal(mods));
      System.out.println(
            "Class is abstract: " + Modifier.isAbstract(mods));

   }

}
```

Run the code above and you will see something like:

```
Class name: com.russolsen.reflect.Employee
Class super class: class java.lang.Object
Class is public: true
Class is final: false
```

```
Class is abstract: false
```

As the example shows, getting the class name and super–class is as easy as calling some accessors. If you are interested in knowing if the class is public or abstract or final, the process is only slightly more complicated. All of this information comes packaged up in a single `int` returned by `getModifiers`. Fortunately, Java supplies you with the `Modifier` class, which is contains a bunch of static methods that will help you make sense of the number returned by `getModifiers`.

Calling `getClass` on an instance is not the only way to get hold of a class object. You can also get it directly from a class name:

```
Class klass = Employee.class;
```

The third way of getting hold of a class is the interesting one: you can create a `Class` object from a string -- well only if that string happens to contain the name of a class. The way to do this is to call the `forName` class method on `Class`:

```
      Class klass = Class.forName("com.russolsen.reflect.Employee");

      System.out.println( "Class name: " + klass.getName());
      System.out.println(
            "Class super class: " + klass.getSuperclass());

      // Print out the rest...
```

One thing to keep in mind about `forName` is that you need to supply the full, complete with the package, name of the class. Plain old "Employee" will not do, it has to be "com.russolsen.reflect.Employee". With `forName` we begin to get a glimpse at some of the fundamental power (and coolness) of the reflection API: you can start with a class name in a string and end up with the class.

Instances Instantly

Getting hold of a class and finding all about it is interesting and useful in its own right, but actually *doing* something with it is where reflection gets really exciting. The most obvious thing to do with a `Class` object is to make a new instance of that class. The simple way to do this is with the `newInstance` method. To show all this in action, the little program below takes a command line argument (which should contain a class name) creates a class from it and then makes a new instance of that class:

```
package com.russolsen.reflect;

public class NewInstanceExample
{
   public static void main(String[] args)
      throws ClassNotFoundException,
      InstantiationException, IllegalAccessException
   {
```

```
        Class klass = Class.forName(args[0]);
        Object theNewObject = klass.newInstance();
        System.out.println("Just made: " + theNewObject);
    }
}
```

Run the code above with "com.russolsen.reflect.Employee" as an argument and you will manufacture a brand new `Employee` object:

```
Just made: Employee: John Smith 50000
```

Run it again, but this time feed it "java.util.Date" and you will get:

```
Just made: Tue Feb 27 20:25:20 EST 2007
```

Think about how much flexibility you get with just a few lines of code: the program above really knows nothing about `Employee` or `Date` and yet it can create a new instance of either. This is a different way of doing Java.

Beyond the no-args Constructor

It turns out that calling the `Class.newInstance` method is identical to doing an ordinary `new` with no arguments. But what happens if you call `newInstance` on a class that lacks a no argument constructor? Nothing good: you are in for an unpleasant `InstantiationException`.

The good news is that you can dynamically create new instances of a class that requires constructor arguments -- you just have to work a bit harder. What you need to do is to find the constructor that you need from the class and call it with the right arguments. The way to find the constructor of your dreams is to call the `getConstructor` method with a description of the constructor that you are looking for. What you will get back is a `Constructor` object, which you can use to create a new instance.

Let's see how this all works in code:

```
        Class klass = Class.forName("com.russolsen.reflect.Employee");

        Class[] paramTypes = {
              String.class,
              String.class,
              Integer.TYPE };

        Constructor cons = klass.getConstructor(paramTypes);

        System.out.println( "Found the constructor: " + cons);


        Object[] args = {
              "Fred",
              "Fintstone",
              new Integer(90000) };

        Object theObject = cons.newInstance(args);
        System.out.println( "New object: " + theObject);
```

The only difference between the constructors are the parameters they take, so the way that you tell `getConstructor` which constructor you are looking for is to pass it an array of `Class`'s, one for each parameter. The example above goes looking for a constructor that takes two strings and an `int`. Once you have a `Constructor`, creating a new object instance with it is very simple: you just call the `newInstance` method, passing in another array, this time an array full of the actual argument values.

There is a nice little land mine hidden in `getConstructor`. When you are specifying the parameter types for the constructor you are looking for, you need to carefully distinguish between a primitive argument and the related object based wrapper. Does the constructor take as an argument the primitive `int` or its uncle, a object of class `java.lang.Integer`? If you are looking for a constructor that takes the object wrapper type, something like `java.lang.Integer`, just use the wrapper class, `Integer.class` If you mean the primitive `int` you will need use `Integer.TYPE`, which is a sort of placeholder for the class that the primitive lacks. All of the wrapper classes have a static `TYPE` field which you can use to indicate a primitive of that type.

Digging Deeper into A Class

As you saw in the first example, a `Class` object can supply you with all of the basic information about a class, things like its name and super class. But you can go way beyond this name, rank and serial number level of information. You can, for example, find out about all of the public methods supported by a class with the `getMethods` method:

```
Class klass = Class.forName("com.russolsen.reflect.Employee");

Method[] methods = klass.getMethods();

for(Method m : methods )
{
   System.out.println( "Found a method: " + m );
}
```

`getMethods` returns an array of `Method` objects, one for each public method that instances of the class will answer to:

```
Found a method: public java.lang.String com.russolsen.reflect.Employee.toString()
Found a method: public int com.russolsen.reflect.Employee.getSalary()
Found a method: public void com.russolsen.reflect.Employee.setSalary(int)
Found a method: public native int java.lang.Object.hashCode()
Found a method: public final native java.lang.Class java.lang.Object.getClass()
Found a method: public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
Found a method: public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
Found a method: public final void java.lang.Object.wait() throws java.lang.InterruptedException
Found a method: public boolean java.lang.Object.equals(java.lang.Object)
Found a method: public java.lang.String java.lang.Object.toString()
Found a method: public final native void java.lang.Object.notify()
Found a method: public final native void java.lang.Object.notifyAll()
```

Since `getMethods` takes a client's view of the class, the array will contains *all* of the public methods, including those defined in the class itself and in its super class and its super–duper class and so on up to `Object`.

If you are interested in a single method, you can use `getMethod` (note that it is singular), which works a lot like `getConstructor`, except that you need to pass in a method name as well as the parameter types. The code below looks for a method called `setSalary` that takes a single `int` parameter:

```
Class klass = Class.forName("com.russolsen.reflect.Employee");
Class[] paramTypes = {Integer.TYPE };
Method setSalaryMethod =
   klass.getMethod("setSalary", paramTypes);

System.out.println( "Found method: " + setSalaryMethod);
```

Calling a method via reflection turns out to be very similar calling a constructor. All you need is the `Method` object that we talked about earlier, an array of arguments to pass to the method and the instance whose method you are calling. The code below calls the `setSalary` method on an `Employee` object to give our man a raise:

```
Class klass = Class.forName("com.russolsen.reflect.Employee");

Class[] paramTypes = {Integer.TYPE };
Method setSalaryMethod =
   klass.getMethod("setSalary", paramTypes);

Object theObject = klass.newInstance();
Object[] parameters = { new Integer(90000) };

setSalaryMethod.invoke(theObject, parameters);
```

Why would you bother going through all of this instead of just typing `theObject.setSalary(90000)`? Well, take another look at the code above. Other than the string on the first line, the program above is completely generic: you could use it to call the `setSalary` method on any instance of any class. With a little rearrangement, you can use the code above to call any method on any class.

Messing with Fields

You are not limited to just calling methods via reflection: you also have full reign over fields. Similar to `getMethods`, `getFields` returns an array of `Field` objects, one for each public instance field supplied by the class or its super classes:

```
Class klass = Class.forName("com.russolsen.reflect.Employee");

System.out.println( "Class name: " + klass.getName());

Field[] fields = klass.getFields();

for(Field f : fields )
{
   System.out.println( "Found field: " + f);
}
```

Since `Employee` has exactly two public fields, you get a two member array back:

```
Class name: com.russolsen.reflect.Employee
Found field: public java.lang.String com.russolsen.reflect.Employee._firstName
Found field: public java.lang.String com.russolsen.reflect.Employee._lastName
```

You can also look for a specific field by name with the `getField` method:

```
Field field = klass.getField("_firstName");
System.out.println("Found field: " + field);
```

Once you have a `Field` object, getting the field value is as simple as calling the `get` method, while setting the value is just a call to `set` away:

```
Object theObject = new Employee("Tom", "Smith", 25);

Class klass = Class.forName("com.russolsen.reflect.Employee");

Field field = klass.getField("_firstName");

Object oldValue = field.get(theObject);
System.out.println( "Old first name is: " + oldValue);

field.set( theObject, "Harry");
Object newValue = field.get(theObject);
System.out.println( "New first name is: " + newValue);
```

Run the code above and watch the value of the `_salary` field change before your very eyes:

```
Old first name is: Tom
New first name is: Harry
```

Breaking the Rules

No look at reflection can be complete without talking about how to break one of the most sacred of all the rules in Java. Everyone knows that you cannot call a private method from outside of the class that defines it. Right? Well you can't if you stick to conventional techniques, but with reflection you can do darn near anything.

The first thing you need to call a private method is the `Method` instance that goes with the method you want to call. You can't get this from `getMethod`; it only returns public methods. The way to get hold of a private (or protected) method is to use `getDeclaredMethod`. While `getMethod` takes a client's view of a class and only returns public methods, `getDeclaredMethod` returns *all* of the methods declared by one class. In the example below, we use it to get at the `Method` object for the very private `removeRange` method on the `java.util.ArrayList` class:

```
ArrayList list = new ArrayList();
list.add("Larry");
list.add("Moe");
list.add("Curley");

System.out.println("The list is: " + list);

Class klass = list.getClass();
```

```
Class[] paramTypes = { Integer.TYPE, Integer.TYPE };
Method m = klass.getDeclaredMethod("removeRange", paramTypes);

Object[] arguments = { new Integer(0), new Integer(2) };
m.setAccessible(true);
m.invoke(list, arguments);
System.out.println("The new list is: " + list);
```

Once you have a `private` method, calling it is a simple matter of clicking off the final `setAccessable` safety and having at it:

```
The list is: [Larry, Moe, Curley]
The new list is: [Curley]
```

The `removeRange` method appears to remove the given range of items from the list. This is pretty strong voodoo: you just reached in and called a private method on a `java.util` class. Would you want to make a habit of circumventing the clear intent of the code and calling `private` methods? No! But such things have been known to come in handy in a pinch.

Conclusion

Reflection allows your programs to do things that seem to defy the rules of Java. You can write code that can find out all about a class to which it has never been properly introduced. And your code can act on that dynamically discovered knowledge: it can create new instances, call methods and set and get the value of fields. In extreme circumstances you can even mess with the private internals of a class. A good understanding of reflection is what you need to grasp the workings of some of the more sophisticated tools of the Java world; it is also what you need to write programs that go beyond what "ordinary" Java programs can do.

Resources

- Reflection tutorial
- In depth article about reflection from IBM's developerWorks

Russ Olsen is currently a senior engineer with FGM, where he builds information systems with both J2EE and Rails. Russ spends a lot of his otherwise free time writing about technology.

Return to ONJava.com.