Take an in-depth look at the Java Reflection API

Learn about the new Java 1.1 tools for finding out information about classes

By Chuck Mcmanis, JavaWorld.com, 09/01/97

In last month's "Java In-Depth," I talked about introspection and ways in which a Java class with access to raw class data could look "inside" a class and figure out how the class was constructed. Further, I showed that with the addition of a class loader, those classes could be loaded into the running environment and executed. That example is a form of *static* introspection. This month I'll take a look at the Java Reflection API, which gives Java classes the ability to perform *dynamic* introspection: the ability to look inside classes that are already loaded.

The utility of introspection

One of Java's strengths is that it was designed with the assumption that the environment in which it was running would be changing dynamically. Classes are loaded dynamically, binding is done dynamically, and object instances are created dynamically on the fly when they are needed. What has not been very dynamic historically is the ability to manipulate "anonymous" classes. In this context, an anonymous class is one that is loaded or presented to a Java class at run time and whose type was previously unknown to the Java program.

Anonymous classes
Supporting anonymous classes is hard to explain and even harder to design for in a program. The challenge of supporting an anonymous class can be stated like this: "Write a program that, when given a Java object, can incorporate that object into its continuing operation." The general solution is rather difficult, but by constraining the problem, some specialized solutions can be created. There are two examples of specialized solutions to this class of problem in the 1.0 version of Java: Java applets and the command-line version of the Java interpreter.

Java applets are Java classes that are loaded by a running Java virtual machine in the context of a Web browser and invoked. These Java classes are anonymous because the run time does not know ahead of time the necessary information to invoke each individual class. However, the problem of invoking a particular class is solved using the Java class `java.applet.Applet`.

Common superclasses, like `Applet`, and Java interfaces, like `AppletContext`, address the problem of anonymous classes by creating a previously agreed upon contract. Specifically, a runtime environment supplier advertises that she can use any object that conforms to a specified interface, and the runtime environment consumer uses that specified interface in any object he intends to supply to the run time. In the case of applets, a well-specified interface exists in the form of a common superclass.

The downside of a common superclass solution, especially in the absence of multiple inheritance, is that the objects built to run in the environment cannot also be used in some other system unless that system implements the entire contract. In the case of the `Applet` interfaces, the hosting environment has to implement `AppletContext`. What this means for the applet solution is that the solution only works when you are loading applets. If you put an instance of a `Hashtable` object on your Web page and point your browser to it, it would fail to load because the applet system cannot operate outside its limited range.

In addition to the applet example, introspection helps to solve a problem I mentioned last month: figuring out how to start execution in a class that the command-line version of the Java virtual machine has just loaded. In that example, the virtual machine has to invoke some static method in the loaded class. By convention, that method is named `main` and takes a single argument -- an array of `String` objects.

The motivation for a more dynamic solution

The challenge with the existing Java 1.0 architecture is that there are problems that could be solved by a more dynamic introspection environment -- such as loadable UI components, loadable device drivers in a Java-based OS, and dynamically configurable editing environments. The "killer app," or the issue that caused the Java Reflection API to be created, was the development of an object component model for Java. That model is now known as JavaBeans.

User interface components are an ideal design point for an introspection system because they have two very different consumers. On the one hand, the component objects are linked together to form a user interface as part of some application. Alternatively, there needs to be an interface for tools that manipulate user components without having to know what the components are, or, more importantly, without access to the components' source code.

The Java Reflection API grew out of the needs of the JavaBeans user interface component API.

What is reflection?

Fundamentally, the Reflection API consists of two components: objects that represent the various parts of a class file, and a means for extracting those objects in a safe and secure way. The latter is very important, as Java provides many security safeguards, and it would not make sense to provide a set of classes that invalidated those safeguards.

The first component of the Reflection API is the mechanism used to fetch information about a class. This mechanism is built into the class named `Class`. The special class `Class` is the universal type for the meta information that describes objects within the Java system. Class loaders in the Java system return objects of type `Class`. Up until now the three most interesting methods in this class were:

- `forName`, which would load a class of a given name, using the current class loader
- `getName`, which would return the name of the class as a `String` object,which was useful for identifying object references by their class name

- `newInstance`, which would invoke the null constructor on the class (if it exists) and return you an object instance of that class of object

To these three useful methods the Reflection API adds some additional methods to class `Class`. These are as follows:

- `getConstructor`, `getConstructors`, `getDeclaredConstructor`
- `getMethod`, `getMethods`, `getDeclaredMethods`
- `getField`, `getFields`, `getDeclaredFields`
- `getSuperclass`
- `getInterfaces`
- `getDeclaredClasses`

In addition to these methods, many new classes were added to represent the objects that these methods would return. The new classes mostly are part of the `java.lang.reflect` package, but some of the new basic type classes (`Void`, `Byte`, and so on) are in the `java.lang` package. The decision was made to put the new classes where they are by putting classes that represented meta-data in the reflection package and classes that represented types in the language package.

Thus, the Reflection API represents a number of changes to class `Class` that let you ask questions about the internals of the class, and a bunch of classes that represent the answers that these new methods give you.

How do I use the Reflection API?

The question "How do I use the API?" is perhaps the more interesting question than "What is reflection?"

The Reflection API is *symmetric*, which means that if you are holding a `Class` object, you can ask about its internals, and if you have one of the internals, you can ask it which class declared it. Thus you can move back and forth from class to method to parameter to class to method, and so on. One interesting use of this technology is to find out most of the interdependencies between a given class and the rest of the system.

A working example
On a more practical level, however, you can use the Reflection API to dump out a class, much as my `dumpclass` class did in last month's column.

To demonstrate the Reflection API, I wrote a class called `ReflectClass` that would take a class known to the Java run time (meaning it is in your class path somewhere) and, through the Reflection API, dump out its structure to the terminal window. To experiment with this class, you will need to have a 1.1 version of the JDK available.

Note: Do not *try to use a 1.0 run time as it gets all confused, usually resulting in an incompatible class change exception.*

The class `ReflectClass` begins as follows:

```
import java.lang.reflect.*;
import java.util.*;
public class ReflectClass {
```

As you can see above, the first thing the code does is import the Reflection API classes. Next, it jumps right into the main method, which starts out as shown below.

```
        public static void main(String args[]) {
        Constructor   cn[];
        Class         cc[];
        Method        mm[];
        Field         ff[];
        Class         c = null;
        Class         supClass;
        String      x, y, s1, s2, s3;
        Hashtable classRef = new Hashtable();
        if (args.length == 0) {
            System.out.println("Please specify a class name on the command line.");
            System.exit(1);
        }
        try {
            c = Class.forName(args[0]);
        } catch (ClassNotFoundException ee) {
            System.out.println("Couldn't find class '"+args[0]+"'");
            System.exit(1);
        }
```

The method `main` declares arrays of constructors, fields, and methods. If you recall, these are three of the four fundamental parts of the class file. The fourth part is the attributes, which the Reflection API unfortunately does not give you access to. After the arrays, I've done some command-line processing. If the user has typed a class name, the code attempts to load it using the `forName` method of class `Class`. The `forName` method takes Java class names, not file names, so to look inside the `java.math.BigInteger` class, you simply type "java ReflectClass java.math.BigInteger," rather than point out where the class file actually is stored.

Identifying the class's package
Assuming the class file is found, the code proceeds into Step 0, which is shown below.

```
        /*
         * Step 0: If our name contains dots we're in a package so put
         * that out first.
         */
        x = c.getName();
        y = x.substring(0, x.lastIndexOf("."));
        if (y.length() > 0) {
            System.out.println("package "+y+";\n\r");
        }
```

In this step, the name of the class is retrieved using the `getName` method in class `Class`. This method returns the fully qualified name, and if the name contains dots, we can presume that the class was defined as part of a package. So Step 0 is to separate the package name part from the class name part, and print out the package name part on a line that starts with "package...."

Collecting class references from declarations and parameters
With the package statement taken care of, we proceed to Step 1, which is to collect all of the *other* class names that are referenced by this class. This collection process is shown in the code below. Remember that the three most common places where class names are referenced are as types for fields (instance variables), return types for methods, and as the types of the parameters passed to methods and constructors.

```
        ff = c.getDeclaredFields();
        for (int i = 0; i < ff.length; i++) {
            x = tName(ff[i].getType().getName(), classRef);
        }
```

In the above code, the array `ff` is initialized to be an array of `Field` objects. The loop collects the type name from each field and process it through the `tName` method. The `tName` method is a simple helper that returns the shorthand name for a type. So `java.lang.String` becomes `String`. And it notes in a hashtable which objects have been seen. At this stage, the code is more interested in collecting class references than in printing.

The next source of class references are the parameters supplied to constructors. The next piece of code, shown below, processes each declared constructor and collects the references from the parameter lists.

```
cn = c.getDeclaredConstructors();
        for (int i = 0; i < cn.length; i++) {
            Class cx[] = cn[i].getParameterTypes();
            if (cx.length > 0) {
                for (int j = 0; j < cx.length; j++) {
                    x = tName(cx[j].getName(), classRef);
                }
            }
        }
```

As you can see, I've used the `getParameterTypes` method in the `Constructor` class to feed me all of the parameters that a particular constructor takes. These are then processed through the `tName` method.

An interesting thing to note here is the difference between the method `getDeclaredConstructors` and the method `getConstructors`. Both methods return an array of constructors, but the `getConstructors` method only returns those constructors that are accessible to your class. This is useful if you want to know if you actually can invoke the constructor you've found, but it isn't useful for this application because I want to print out all of the constructors in the class, public or not. The field and method reflectors also have similar versions, one for all members and one only for public members.

The final step, shown below, is to collect the references from all of the methods. This code has to get references from both the type of the method (similar to the fields above) and from the parameters (similar to the constructors above).

```
        mm = c.getDeclaredMethods();
        for (int i = 0; i < mm.length; i++) {
            x = tName(mm[i].getReturnType().getName(), classRef);
            Class cx[] = mm[i].getParameterTypes();
            if (cx.length > 0) {
                for (int j = 0; j < cx.length; j++) {
                    x = tName(cx[j].getName(), classRef);
                }
            }
        }
```

In the above code, there are two calls to `tName` -- one to collect the return type and one to collect each parameter's type.

Once these three loops have completed, the method is ready to generate the import statements for the class. This is done simply by enumerating the hashtable and printing out the keys, as shown below.

```
        // Don't import ourselves ...
        classRef.remove(c.getName());
        for (Enumeration e = classRef.keys(); e.hasMoreElements(); ) {
            System.out.println("import "+e.nextElement()+";");
        }
        System.out.println();
```

Another feature of the above code is that it removes the name of the base class to prevent generating an import statement for this class.

Generate the class or interface declarations
Now with the imports out of the way, the code can generate the outline for the class itself. The header of the class always takes the form:

```
    [ optional modifiers (public private etc.)]
    class/interface ClassName extends ClassName
    {
```

The code that generates this is shown next.

```
        int mod = c.getModifiers();
        System.out.print(Modifier.toString(mod));
        if (Modifier.isInterface(mod)) {
            System.out.print(" interface ");
        } else {
            System.out.print(" class ");
        }
        System.out.print(tName(c.getName(), null));
        supClass = c.getSuperclass();
        if (supClass != null) {
            System.out.print(" extends "+tName(supClass.getName(), classRef));
        }
        System.out.println(" {");
```

Walking through the code above, the first thing it does is collect the modifiers. These are then printed using a method in the `Modifier` class. That method prints out all of the modifiers that make sense for a type declaration such as *public*, *private*, *static*, and so on. Next I've used the `isInterface` method to check to see if the class I've loaded is an interface. If it isn't, I print out the keyword `class`; otherwise, I print out the keyword `interface`. Then comes the name, which is followed by the `extends` keyword and the name of this class's superclass. The code finishes up by printing the opening curly brace.

Generate the field (variable) declarations
Variables that are declared within the class -- both static variables (class variables) and non-static variables (instance variables) -- are then printed. Whether they are static or dynamic, variable declarations in the class are called fields.

After the class declaration has been printed, the `ReflectClass` class prints out all of the fields that the class defines. Throughout the code, the `tName` method is being used to provide the shorthand versions of the type names. The fields are printed with the following code:

```
        System.out.println("\n\r/*\n\r * Field Definitions.\r\n */");
        for (int i = 0; i < ff.length; i++) {
            Class ctmp = ff[i].getType();
            int md     = ff[i].getModifiers();
            System.out.println("    "+Modifier.toString(md)+" "+
                    tName(ff[i].getType().getName(), null) +" "+
                    ff[i].getName()+";");
        }
```

The type of the field is collected using the `getType` method. The field is then printed, modifiers first, then the type name (int, FooClass, and so on), and finally the name of the field.

Generate the constructor declarations
After the fields, I've chosen to print out the constructors. This primarily is a convention that I use when I write my own Java code, but there isn't anything special about the printing order.

```
        System.out.println("\n\r/*\n\r * Declared Constructors. \n\r */");
        x = tName(c.getName(), null);
        for (int i = 0; i < cn.length; i++) {
            int md = cn[i].getModifiers();
            System.out.print("    " + Modifier.toString(md) + " " + x);
            Class cx[] = cn[i].getParameterTypes();
            System.out.print("( ");
            if (cx.length > 0) {
                for (int j = 0; j < cx.length; j++) {
                    System.out.print(tName(cx[j].getName(), null));
                    if (j < (cx.length - 1)) System.out.print(", ");
                }
            }
            System.out.print(") ");
            System.out.println("{ ... }");
        }
```

The above code is a bit more complicated because, in addition to the name, a set of optional parameters have to be printed out as well. Note that if the parameter length was 0, then the code prints "( )". Finally, a pair of curly braces are printed to indicate where the code would go in the actual source.

Generate the method declarations
The final chunk of code prints out the method declarations and their parameters. This code is shown here:

```
        System.out.println("\n\r/*\n\r * Declared Methods.\n\r */");
        for (int i = 0; i < mm.length; i++) {
            int md = mm[i].getModifiers();
            System.out.print("    "+Modifier.toString(md)+" "+
                    tName(mm[i].getReturnType().getName(), null)+" "+
                    mm[i].getName());
            Class cx[] = mm[i].getParameterTypes();
            System.out.print("( ");
            if (cx.length > 0) {
                for (int j = 0; j < cx.length; j++) {
                    System.out.print(tName(cx[j].getName(), classRef));
                    if (j < (cx.length - 1)) System.out.print(", ");
                }
            }
            System.out.print(") ");
            System.out.println("{ }");
        }
```

The additional complexity in the above code comes from printing both a return type, like the fields, and a list of parameters, like the constructors.

When I ran `ReflectClass` and fed it the name of the `ReflectClass` class, it produced the following output:

```
import java.util.Hashtable;
import java.lang.String;
public synchronized class ReflectClass extends Object {
/*
 * Field Definitions.
 */
/*
 * Declared Constructors.
 */
    public ReflectClass( ) { ... }
/*
 * Declared Methods.
 */
    static String tName( String, Hashtable) { ... }
    public static void main( String[]) { ... }
}
```

The complete source to the `ReflectClass` class is available in the Resources section.

Going further

The Reflection API is a powerful tool for inspecting classes. Further, unlike last month where we were left only with information about the class, the methods represented by the `Method` class and the constructors represented by the `Constructor` class can actually be invoked. Unlike the situation last month, however, we are unable to modify the classes before they are actually loaded, and there is no access to attributes in the class file itself. Thus, there are times when introspection through file inspection is more useful than introspection through reflection. The Reflection API does solve that long-standing problem with dynamic Java execution of creating an instance of a dynamically loaded class that did not have a null constructor.

The issue with null constructors can be stated as follows, "The `newInstance` method in class `Class` implicitly invokes the public constructor in the object that takes no parameters (the null constructor)." If you attempt to dynamically load a class with the `forName` method of class `Class`, and it does not have an accessible null constructor, historically there has been no way to create an instance of that class. With the Reflection API you can get the constructors with the `getConstructors` method of class `Class`, and, using a public constructor, invoke it using the `newInstance` method of the `Constructor` class. `Constructor`'s `newInstance` method takes an array of objects that supply the arguments to non-null constructors.

This ability to invoke arbitrary methods and constructors also provides the opportunity to solve the hack in the Java interpreter that requires all application base classes to have a static `main` method that takes an array of `String` objects as an argument. With the Reflection API, one could construct an interpreter that takes a typed method invocation line, such as:

```
    jvm myClass(parm1, parm2).someMethod("This is the base method to start");
```

In my fictitious example, the command *jvm* takes as its argument a method invocation. The command line is parsed, and first the class is loaded, then reflection is used to identify its constructors. A constructor that takes two parameters is located and the method is found. Finally, the parameter types taken by the method are parsed from the command line, and the object is created and invoked without ever having an artificial "starting" method named `main`. A really creative individual might use this technique as the basis for a command shell written in Java. Certainly something to think about until next month.

About the author

Chuck McManis currently is the director of system software at FreeGate Corp., a venture-funded start-up that is exploring opportunities in the Internet marketplace. Before joining FreeGate, Chuck was a member of the Java Group. He joined the Java Group just after the formation of FirstPerson Inc. and was a member of the portable OS group (the group responsible for the OS portion of Java). Later, when FirstPerson was dissolved, he stayed with the group through the development of the alpha and beta versions of the Java platform. He created the first "all Java" home page on the Internet when he did the programming for the Java version of the Sun home page in May 1995. He also developed a cryptographic library for Java and versions of the Java class loader that could screen classes based on digital signatures. Before joining FirstPerson, Chuck worked in the operating systems area of SunSoft, developing networking applications, where he did the initial design of NIS+. Check out his home page.

Read more about Core Java in JavaWorld's Core Java section.