

Reflection

I've been doing a little "reflecting" lately. This column began in January 1999 with these words:

"Hello again. *import java.** invites you, the C/C++ programmer, to learn Java... I will be examining all aspects of the language, library and culture, but from a C/C++ perspective."

To explore the entire Java library would take more years than I care to think about, but I believe I've covered most of the language. Since I have recently accepted the position of Senior Editor for CUJ, this will have to be the last installment of *import java.**, so I would like to take this opportunity to cover the one language feature I haven't yet mentioned: reflection.

Meta Programming

The first thing to say about reflection in Java is that you should almost never use it. Reflective programming, also known as "meta programming", gives you access to information you usually don't need. It allows you, for instance, to inspect an object at runtime to determine its dynamic type. Whether a particular object reference actually points to an object of the indicated type or a subclass is irrelevant, however, in most situations, because you want to let polymorphism do its work.

So when do you need reflection? If you're in the business of writing debuggers or class browsers or some such software, then you need it, and nothing else will do. Under the assumption that you will find yourself there someday, this article shows you the basics.

Class Objects

For every class in Java there is a corresponding singleton object of type `Class`, instantiated when the class is loaded, and which holds information about that class. You can obtain that object in four different ways. If you have the name of the class available when you're coding, you can use a class literal, as in

```
Class c = MyClass.class;
```

For completeness, the primitive types also have a class object, and you can get your hands on them with the class literal technique, or from the `TYPE` field of the corresponding wrapper class:

```
Class c = int.class;  
Class c2 = Integer.TYPE; // same object as c
```

A second and more flexible avenue to a class object is to give the fully qualified class name as a parameter to `Class.forName`, as the following statements illustrate.

```
Class c = Class.forName("MyClass");  
Class c = Class.forName("java.lang.String");
```

A third method for getting class objects is to call `Object.getClass` on behalf of an object instance:

```
// "o" can be any object:  
Class c = o.getClass();
```

The fourth way of getting class objects is to call the reflection methods defined in `Object.Class` itself, which I use throughout this article starting in the next section. The program in Listing 1 illustrates the first three techniques. Whatever technique you use, you always get the class corresponding to the dynamic type of an object. In Listing 1, for example, it doesn't matter that I store a reference to a `Sub` object in a `Super` variable – since the actual object is an instance of `Sub`, then the class object returned is `Sub.class`. The `Class.forName` method will throw a `ClassNotFoundException` if the class can't be loaded. As you can see, the `Class.toString` prints the word "class" or "interface" as needed. You can also see that arrays are instances of classes with interesting names, formed

with left brackets (as many as there are array dimensions) following by one of the following letters, depending on the underlying component class:

B	byte
C	char
D	double
F	float
I	int
J	long
L<classname>;	
S	short
Z	boolean

This explains why you see “class [F” and “class [LSuper;” in the output of Listing 1.

Okay, now that you have a class object, what can you do with it? Lots. The only interesting information in a primitive type’s class object is its name, but for real classes you can find out just about everything. In Listing 2 I trace a class’s ancestry all the way to the `Object` root class with the `Class.getSuperClass` method. Since we typically depict superclasses above subclasses, I use a stack¹ to store all the class names as I walk up the inheritance tree, so I can eventually print them out in that top-down order. The class `java.lang.Object` is the only class that returns null for the `Class.getSuperClass` method, which fact I use to halt the process.

The program in Listing 3 goes a little further by also detecting any interfaces a class implements by calling `Class.getInterfaces`, or, if the class represents an array, discovers the type of its components by calling `Class.getComponentType`. If a class object returns false for calls to `isInterface`, `isArray`, and `isPrimitive`, then it must be a simple class. The output reveals that arrays are implicitly cloneable and serializable, which is a Good Thing, since there is no syntax that allows you to declare them so.

On Further Reflection

If you’re writing a browser or debugger you will surely want more than just the name of a class. There are methods to get all the fields, constructors and methods, and modifiers of a class, represented by instances of the classes `Field`, `Constructor`, `Method`, and `Modifier`, respectively, all defined in the package `java.lang.reflect`. The modifiers include the access specifiers `public`, `private`, and `protected`, as well as `static`, `final`, and `abstract`. If a class represents an interface, the keyword “interface” appears in the list of modifiers as well. These keywords are encoded in an integer returned by `Class.getModifiers`. You can query the integer code with suitable `Modifier` class constants, or you can just get the string representation for all modifiers in a single string by passing the integer to the static method `Modifier.toString`, as I do in Listing 4.

This program produces a listing similar in syntax to the original class definition, minus the method bodies (similar to a class declaration with prototypes in C++). The first thing `ClassInspector2.inspect` does is determine the package, if any, for the outer class it received as a parameter. Then it gets the class’s modifiers and prints them along with the keyword “class”, if applicable, and finally the class or interface name. It then displays the superclass extended and any interfaces implemented by the class, like Listing 3 did.

Now the fun begins. My `doFields` method calls `Class.getDeclaredFields`, which returns an array of `Field` objects representing *all* of the fields declared in the class, irrespective of access specification. There is another method, `Class.getFields`, not used in this example, which returns all of the *public* fields in a class as well as all of the public fields inherited from all superclasses. From each `Field` instance I extract its modifiers, type, and name for display. The methods `doCtors` and `doMethods` do the analogous actions for all constructors and methods declared in the class. Constructors and methods are treated differently because constructors don’t have return types, and because `Constructor` objects can be used to create objects dynamically through class objects. `Class.getParameterTypes` returns an (possibly empty) array of class objects representing the types of the parameters the method takes (see the calls to `doParameters` in `doCtors` and `doMethods`). You can also get call `getExceptionTypes` for constructors and methods, although I decided not to for this example.

The method `Class.getDeclaredClasses` returns an array of class objects containing type information for all the nested classes declared inside the class that `inspect` is processing. All I have to do for these is call `inspect` recursively, indenting appropriately for readability (that's what the field `level` is for). The sample output from this program, found in Listing 5, is for the class `java.util.TreeMap`, which I chose because it illustrates all the features supported by `ClassInspector2`. Most of the methods and constructors have been omitted from the output listing to save page real estate.

You may find it interesting that you can get information on private fields and methods. Since the methods illustrated in Listing 4 only give you declaration information, it's really no different than having access to an include file containing a class definition in C++. You can *view* the private declarations, but you have no access to the actual data they represent. If you're writing a debugger, though, you need to be able to access the private data. How to get that access is easy to illustrate but difficult to thoroughly explain. The program in Listing 6 inspects arbitrary objects by determining all fields at runtime, including inherited fields (ignoring those in `java.lang.Object`). The method `Field.get` yields the value of a field as an instance of `java.lang.Object`. If the field is a primitive, the value is automatically wrapped in an instance of the corresponding object wrapper class. Whenever you try to set or get a field, the runtime system verifies your access rights, just as the compiler does with normal, non-reflected code. If I hadn't called `AccessibleObject.setAccessible(fields, true)`² in `ObjectInspector.inspect`, I would have been greeted with an `IllegalAccessException` the first time I tried to access one of the fields in the `TreeMap` object, since none is public. Whether you ultimately get access permission depends on the class loader and security manager in use – both topics outside the scope of this article. Suffice it to say that the default case for an application (not an applet) allows me to get the output that appears in Listing 6 without error.

Meta Execution

I'm a little reluctant to write this section. When I tell you that you can mimic C-like function pointers in Java, I just know you're going to be tempted to use them the same way you do in C, and you shouldn't. There are better ways to pass functions in C++ and Java (e.g., function objects), but that is, alas, yet another topic for another day. Anyway, yes, you can pass method references around, and, as you'd expect, you can determine which method you want to execute at runtime by its name string. To get a method reference, you need its name, and a list of the types of its arguments. In Listing 7, I've defined a class `Foo` and a class `Bar`, each with like-named methods (am I creative with names, or what?). To get a method reference at runtime, call `Class.getMethod` with two arguments: the method name as a `String`, and an array of argument types. You can use `null` or an empty `Class` array to match methods that take no arguments. You can only get references to public methods, but if the method you're after isn't in the class itself, the runtime system will look in superclasses to find it. Be prepared to handle a `NoSuchMethodException` if the method doesn't exist. To invoke the method, you pass the object to invoke it for, if it's a non-static method, and a list of expected parameters in array of `Object`, to `Method.invoke`.

The program in Listing 8 shows how to invoke static methods – you just use `null` as the first parameter to `Method.invoke`. It's important to remember to place the array of `String`, which is the argument to `main`, of course, in the first position of the `Object` array representing the arguments passed to `main`. This particular program is a program launcher – it finds the `main` method for the class represented by `args[0]`, and passes the remaining command-line arguments to that `main`.

Summary

I haven't shown it here, but there is a `newInstance` method in the `Constructor` class for creating objects, and there is also an `Array` class in `java.lang.reflect` for creating and manipulating arrays dynamically. Isn't reflection fun? Almost too fun. I hope you find little need for it. I'm sure you can appreciate how it is useful for inspecting objects at runtime, like in a class browser or a program that processes JavaBeans. If that's not what you're doing, let polymorphism and good object-oriented design solve your problems for you.

Listing 1 – File ClassName.java: Illustrates Class Objects

```
interface Inter
{}

class Super
{}

class Sub extends Super
{}

class ClassName
{
    public static void displayClassName(Object o)
    {
        System.out.println(o.getClass());
    }

    public static void main(String[] args)
    {
        // Verify dynamic type is obtained
        Super s = new Sub();
        displayClassName(s);

        // Primitives have a Class object too
        Class c = int.class;
        System.out.println(c);

        // So do interfaces
        try
        {
            c = Class.forName("Inter");
            System.out.println(c);
        }
        catch (ClassNotFoundException x)
        {
            System.out.println(x);
        }

        // As do arrays of a given type
        float[] arr = new float[4];
        displayClassName(arr);

        Super[] arr2 = new Super[5];
        displayClassName(arr2);
    }
}

/* Output:
class Sub
int
interface Inter
class [F
class [LSuper;
*/
```

Listing 2 – File Ancestry.java: Prints a Class's Superclasses

```
import java.util.*;

class Ancestry
{
    public static void printAncestry(Class c)
    {
        // Walk up the inheritance path
        LinkedList stack = new LinkedList();
        stack.addFirst(c);
        while ((c = c.getSuperclass()) != null)
            stack.addFirst(c);

        // Print with java.lang.Object at top
        for (int i = 0; stack.size() > 0; ++i)
        {
            if (i > 0)
            {
                System.out.println("  ^");
            }
            Class cl = (Class)stack.removeFirst();
            System.out.println(cl.getName());
        }
    }

    public static void main(String[] args)
        throws ClassNotFoundException
    {
        printAncestry(Class.forName("Sub"));
    }
}

/* Output:
java.lang.Object
 ^
Super
 ^
Sub
*/
```

Listing 3 – File ClassInspector.java: Prints minimal class information, including interfaces implemented

```
class ClassInspector
{
    public static void inspect(Class c)
    {
        // Determine type of class
        String type;
        if (c.isInterface())
            type = "interface";
        else if (c.isArray())
            type = "array";
        else if (c.isPrimitive())
            type = "primitive";
        else
            type = "class";

        // Print name
        System.out.println(type + " " + c.getName());

        // Print component type, if array
        if (type == "array")
            System.out.println("\tarray of " +
                c.getComponentType().getName());
    }
}
```

```

    // Print Superclass
    Class superClass = c.getSuperclass();
    if (superClass != null)
        System.out.println("\textends " +
                            superClass.getName());

    // Print interfaces implemented, if any
    Class[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
        System.out.println("\timplements " +
                            interfaces[i].getName());
}

public static void main(String[] args)
    throws ClassNotFoundException
{
    inspect(Class.forName("Sub"));
    inspect(Class.forName("Super"));
    inspect(Class.forName("java.lang.Object"));
    inspect(int.class);
    Super[] arr = new Super[2];
    inspect(arr.getClass());
    inspect(Class.forName("Inter"));
}
}

/* Output:
class Sub
    extends Super
class Super
    extends java.lang.Object
class java.lang.Object
primitive int
array [LSuper;
    array of Super
    extends java.lang.Object
    implements java.lang.Cloneable
    implements java.io.Serializable
interface Inter
*/

```

Listing 4 – File ClassInspector2.java: Lists all declarations of a class recursively

```

import java.lang.reflect.*;

class ClassInspector2
{
    static int level = 0;    // indentation depth
    static final String indentString = "  ";

    static void indent()
    {
        for (int i = 0; i < level; ++i)
            System.out.print(indentString);
    }

    static void indent(String s)
    {
        indent();
        System.out.print(s);
    }
}

```

```

public static void inspect(Class c)
{
    if (level == 0)
    {
        // Get Package
        Package pkg = c.getPackage();
        if (pkg != null)
            indent("package " + pkg.getName() + ";\n\n");
    }

    // Get Modifiers & Name
    String mods = Modifier.toString(c.getModifiers());
    indent();
    if (mods.length() > 0)
        System.out.print(mods + " ");
    if (!c.isArray() && !c.isInterface() && !c.isPrimitive())
        System.out.print("class ");
    System.out.println(c.getName());

    // Get Superclass
    Class sup = c.getSuperclass();
    if (sup != null)
    {
        ++level;
        indent("extends " + sup.getName() + "\n");
        --level;
    }

    // Get Interfaces
    Class[] interfaces = c.getInterfaces();
    if (interfaces.length > 0)
    {
        ++level;
        for (int i = 0; i < interfaces.length; ++i)
            indent("implements " + interfaces[i].getName() + "\n");
        --level;
    }

    // Start of class definition body
    indent("{\n");

    // Get Fields
    doFields(c);

    // Get Constructors
    doCtors(c);

    // Get Methods
    doMethods(c);

    // Get Nested classes (recursive)
    Class[] classes = c.getDeclaredClasses();
    if (classes.length > 0)
    {
        ++level;
        indent("// Nested Classes\n");
        for (int i = 0; i < classes.length; ++i)
            inspect(classes[i]);
        --level;
    }

    // End of class definition body
    indent("}\n");
}

```

```

static void doCtors(Class c)
{
    Constructor[] ctors = c.getDeclaredConstructors();
    if (ctors.length == 0)
        return;

    ++level;
    indent("// Constructors\n");
    for (int i = 0; i < ctors.length; ++i)
    {
        Constructor ctor = ctors[i];
        String mods = Modifier.toString(ctor.getModifiers());
        indent();
        if (mods.length() > 0)
            System.out.print(mods + " ");
        System.out.print(ctor.getName() + "(");

        // Print parameters
        doParameters(ctor.getParameterTypes());

        System.out.println(");");
    }
    System.out.println();
    --level;
}

static void doFields(Class c)
{
    Field[] fields = c.getDeclaredFields();
    if (fields.length == 0)
        return;

    ++level;
    indent("// Fields\n");
    for (int i = 0; i < fields.length; ++i)
    {
        Field fld = fields[i];
        String type = fld.getType().getName();
        String mods = Modifier.toString(fld.getModifiers());
        indent();
        if (mods.length() > 0)
            System.out.print(mods + " ");
        System.out.println(type + " " + fld.getName() + ";");
    }
    System.out.println();
    --level;
}

static void doMethods(Class c)
{
    Method[] methods = c.getDeclaredMethods();
    if (methods.length == 0)
        return;

    ++level;
    indent("// Methods\n");
    for (int i = 0; i < methods.length; ++i)
    {
        Method m = methods[i];
        Class retType = m.getReturnType();
        String mods = Modifier.toString(m.getModifiers());
        indent();

```

```

        if (mods.length() > 0)
            System.out.print(mods + " ");
        System.out.print(retType.getName() + " " +
            m.getName() + "(");

        // Print parameters
        doParameters(m.getParameterTypes());

        System.out.println(");");
    }
    System.out.println();
    --level;
}

static void doParameters(Class[] parms)
{
    int nParms = parms.length;
    for (int j = 0; j < nParms; ++j)
    {
        Class parmType = parms[j];
        if (j > 0)
            System.out.print(", ");
        System.out.print(parmType.getName());
    }
}

public static void main(String[] args)
    throws ClassNotFoundException
{
    inspect(Class.forName("java.util.TreeMap"));
}
}

```

Listing 5 – Output from Listing 4 (using java.util.TreeMap)

```

package java.util;

public class java.util.TreeMap
    extends java.util.AbstractMap
    implements java.util.SortedMap
    implements java.lang.Cloneable
    implements java.io.Serializable
{
    // Fields
    private java.util.Comparator comparator;
    private transient java.util.TreeMap$Entry root;
    ...
    private static final boolean RED;
    private static final boolean BLACK;
    private static final long serialVersionUID;

    // Constructors
    public java.util.TreeMap(java.util.Map);
    public java.util.TreeMap();
    public java.util.TreeMap(java.util.Comparator);
    public java.util.TreeMap(java.util.SortedMap);

    // Methods
    public java.lang.Object put(java.lang.Object, java.lang.Object);
    public java.lang.Object clone();
    public java.lang.Object get(java.lang.Object);
    private int compare(java.lang.Object, java.lang.Object);
    ...
    static int access$1400(java.util.TreeMap);
}

```

```

// Nested Classes
static class java.util.TreeMap$Entry
    extends java.lang.Object
    implements java.util.Map$Entry
{
    // Fields
    java.lang.Object key;
    ...
    boolean color;

    // Constructors
    ...

    // Methods
    public int hashCode();
    ...
    public java.lang.Object setValue(java.lang.Object);
}
private class java.util.TreeMap$Iterator
    extends java.lang.Object
    implements java.util.Iterator
{
    // Fields
    private int type;
    ...
    private final java.util.TreeMap this$0;

    // Constructors
    ...
    java.util.TreeMap$Iterator(java.util.TreeMap, int);

    // Methods
    public java.lang.Object next();
    public void remove();
    public boolean hasNext();
}
private class java.util.TreeMap$SubMap
    extends java.util.AbstractMap
    implements java.util.SortedMap
    implements java.io.Serializable
{
    // Fields
    private static final long serialVersionUID;
    ...
    private final java.util.TreeMap this$0;

    // Constructors
    ...

    // Methods
    public java.lang.Object put(java.lang.Object, java.lang.Object);
    ...
    static java.lang.Object access$1900(java.util.TreeMap$SubMap);

    // Nested Classes
    private class java.util.TreeMap$SubMap$EntrySetView
        extends java.util.AbstractSet
    {

```

```

        // Fields
        private transient int size;
        private transient int sizeModCount;
        private final java.util.TreeMap$SubMap this$1;

        // Constructors
        ...

        // Methods
        public int size();
        public boolean contains(java.lang.Object);
        public boolean remove(java.lang.Object);
        public boolean isEmpty();
        public java.util.Iterator iterator();
    }
}
}

```

Listing 6 – File ObjectInspector.java: Prints all accessible fields in an object

```

import java.lang.reflect.*;

class ObjectInspector
{
    public static void inspect(Object o)
        throws IllegalAccessException
    {
        // Get class and fields for object
        Class c = o.getClass();

        while (c != Object.class)
        {
            System.out.println("// Fields in " + c);
            Field[] fields = c.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);

            // Print field values
            for (int i = 0; i < fields.length; ++i)
            {
                Field fld = fields[i];
                System.out.print(fld.getName() + ": ");
                System.out.println(fld.get(o));
            }
            System.out.println();
            c = c.getSuperclass();
        }
    }

    public static void main(String[] args)
    {
        try
        {
            inspect(new java.util.TreeMap());
        }
        catch (IllegalAccessException x)
        {
            System.out.println(x);
        }
    }
}

```

```

/* Output:
// Fields in class java.util.TreeMap
comparator: null
root: null
size: 0
modCount: 0
keySet: null
entrySet: null
values: null
KEYS: 0
VALUES: 1
ENTRIES: 2
RED: false
BLACK: true
serialVersionUID: 919286545866124006

// Fields in class java.util.AbstractMap
keySet: null
values: null

*/

```

Listing 7 – File Invoke.java: Illustrates Method.invoke

```

import java.lang.reflect.*;

class Foo
{
    public void foo()
    {
        System.out.println("Foo.foo");
    }
}

class Bar extends Foo
{
    public void bar(String s)
    {
        System.out.println("Bar.bar: " + s);
    }
}

class Invoke
{
    public static void main(String[] args)
        throws Exception
    {
        Bar b = new Bar();
        Class clas = b.getClass();

        // Invoke foo() reflectively
        Method method = clas.getMethod("foo", null);
        method.invoke(b, null);

        // Invoke bar() reflectively
        Class[] argTypes = new Class[] {String.class};
        method = clas.getMethod("bar", argTypes);
        Object[] argVals = new Object[] {"baz"};
        method.invoke(b, argVals);
    }
}

```

```
/* Output:
Foo.foo
Bar.bar: baz
*/
```

Listing 8 – File ClassRunner.java: Illustrates invoking static methods

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

class ClassRunner
{
    public static void main(String[] args)
        throws Exception
    {
        // Determine class and arguments for its main
        Class clas = Class.forName(args[0]);
        String[] newArgs = new String[args.length - 1];
        System.arraycopy(args, 1, newArgs, 0, args.length - 1);
        Object[] newArgVals = new Object[] {newArgs};

        // Find its main and invoke it
        Class[] argTypes = new Class[] {String[].class};
        Method mainMethod = clas.getMethod("main", argTypes);
        mainMethod.invoke(null, newArgVals);
    }
}

class Echo
{
    public static void main(String[] args)
    {
        // Echo args
        System.out.println("Running Echo.main:");
        for (int i = 0; i < args.length; ++i)
            System.out.println("\targs[" + i + "] = " + args[i]);
    }
}

/* Output to "java ClassRunner Echo one two three":
Running Echo.main:
    args[0] = one
    args[1] = two
    args[2] = three
*/
```

¹ I'm using the `LinkedList` collection class to implement a stack. For more on collections see the September 2000 installment of *import java.**.

² `Field`, `Constructor`, and `Method` all derive from `AccessibleObject`.