Java programming dynamics, Part 3: Applied reflection

Building a framework for command line arguments

Dennis Sosnoski July 15, 2003

Command line argument processing is one of those nasty chores that seems to keep coming around no matter how many times you've dealt with it in the past. Rather than writing variations of the same code over and over, why not use reflection to simplify the job of argument processing? Java consultant Dennis Sosnoski shows you how. In this article, Dennis outlines an open source library that makes command line arguments practically handle themselves.

View more content in this series

In last month's article, I introduced the Java Reflection API and ran through some of its basic capabilities. I also looked into reflection performance, and ended with some guidelines for when reflection should or should not be used in an application. This month I'm going even further by looking at an application that seems to be a good match for reflection's strengths and weaknesses: a library for command line argument processing.

I'll start by first defining the problem to be solved, then designing an interface for the library before actually getting into the implementation code. My actual experience developing the library was not so structured -- I started out trying to simplify existing code in a family of applications that use a common code base, then generalized from there. The linear "define-design-build" sequence in this article is much more concise than a full description of the development process, though, and in the process of organizing it this way I've revised some of my initial assumptions and cleaned up several aspects of the library code. Hopefully, you'll find it useful as a model for developing your own reflection-based applications.

Defining the problem

I've written many Java applications using arguments from the command line. Most started out really small, but several ended up growing way beyond my initial plans. There's a standard pattern I've observed in how this process works for me:

Don't miss the rest of this series

Part 1, "Classes and class loading" (April 2003)

Part 2, "Introducing reflection" (June 2003)
Part 4, "Class transformation with Javassist" (September 2003)
Part 5, "Transforming classes on-the-fly" (February 2004)
Part 6, "Aspect-oriented changes with Javassist" (March 2004)
Part 7, "Bytecode engineering with BCEL" (April 2004)
Part 8, "Replacing reflection with code generation" (June 2004)

1. Start with one or two required arguments in a particular order.
2. Think of more things the application should do, then add more arguments.
3. Get tired of typing in all the arguments every time, so make some of the arguments optional, with default values.
4. Forget the order of the arguments, so change the code to allow them in any order.
5. Give the application to other people who are interested. They don't know what the arguments are supposed to be, so add better error checking and "help" descriptions for the arguments.

By the time I get to step 5 I usually regret that I began the whole process in the first place. Fortunately, I tend to forget the latter stages pretty quickly, and within a week or two I'll think of yet another simple little command line application I'd like to have. After that it's just a matter of time before the whole ugly cycle repeats.

There are several libraries available to help with command line argument processing. I'm going to ignore them, however, and go my own way in this article. This isn't (just) because I have a "not invented here" attitude, but rather to use argument processing as an example. As it happens, the strengths and weakness of reflection are a good match with the requirements for an argument processing library. In particular, an argument processing library:

- Needs a flexible interface to support a variety of applications
- Must be easy to configure for each application
- Doesn't require top performance, because the arguments are only processed once
- Has no access security issues, because command line applications generally run without a security manager

The actual reflection code within this library represents only a small portion of the full implementation, so I'll focus primarily on the aspects that are most relevant to reflection. If you want to find out more about the library (and perhaps use it for your own simple command line applications), you'll find the link to the Web site in the Resources section.

Ask the expert: Dennis Sosnoski on JVM and bytecode issues

For comments or questions about the material covered in this article series, as well as anything else that pertains to Java bytecode, the Java binary class format, or general JVM issues, visit the JVM and Bytecode discussion forum, moderated by Dennis Sosnoski.

Sketching out a design

Probably the most convenient way for an application to access argument data is through fields of the application's main object. For example, suppose you're writing an application that generates business plans. You might want to use a `boolean` flag to control whether the business plan is

concise or wordy, an `int` for first year revenue, a `float` for the expected compound revenue growth rate, and a `String` for the product description. I'll call these variables that influence the operation of the application *parameters* to distinguish them from the actual *arguments* (the values for the parameter variables) supplied on the command line. Using fields for these parameters will make them easily available at any point in the application code where they're needed. It's also easy to set defaults for any of the parameters right at the point of definition when using fields, as shown in Listing 1:

Listing 1. Business plan generator (partial listing)

```
public class PlanGen {
    private boolean m_isConcise;          // rarely used, default false
    private int m_initialRevenue = 1000;  // thousands, default is 1M
    private float m_growthRate = 1.5;     // default is 50% growth rate
    private String m_productDescription = // McD look out, here I come
        "eFood - (Really) Fast Food Online";
    ...
    private int revenueForYear(int year) {
        return (int)(m_initialRevenue * Math.pow(m_growthRate, year-1));
    }
    ...
```

Reflection will give direct access to these private fields, allowing the argument processing library to set values without any special hooks in the application code. I *do* need some way for the library to relate these fields to particular command line arguments, though. Before I can define how this linkage between an argument and a field is communicated to the library, I first need to decide how I want to format the command line arguments.

For this article, I'll define a command line format that's a simplified version of UNIX conventions. Argument values for parameters can be supplied in any order, with a leading hyphen to indicate that an argument gives one or more single-character parameter flags (as opposed to actual parameter values). For the business plan generator, I'll pick these parameter flag characters:

- c -- concise plan
- f -- first year revenue ($ thousands)
- g -- growth rate (yearly multiplier)
- n -- product name

The `boolean` parameters only need the flag character itself to set a value, but the other types of parameters require some sort of additional argument information. I'll just append the value of a numeric argument immediately following the parameter flag character (which means digits can't be used as flag characters), while for `String`-valued parameters I'll use the argument following the flag character on the command line as the actual value. Finally, if there are required parameters (such as an output file name for the business plan generator), I'll assume the argument values for these follow the optional parameter values on the command line. Given these conventions, a command line for the business plan generator might look like this:

```
java PlanGen -c -f2500 -g2.5 -n "iSue4U - Litigation at Internet Speed" plan.txt
```

When it's all put together, the meaning of each argument is:

- `-c` -- generates concise plan
- `-f2500` -- first year revenue of $2,500,000
- `-g2.5` -- growth rate of 250 percent per year
- `-n "iSue4U . . ."` -- product name is "iSue4U . . ."
- `plan.txt` -- required name of the output file

At this point I've got a basic functional specification for the argument processing library. The next step is to define a specific interface for the application code to use the library.

Choosing the interface

You can handle the actual processing of command line arguments with a single call, but the application first needs a way to define its particular parameters to the library. These parameters can be of several different types (in the case of the business plan generator example, they can be `boolean`, `int`, `float`, and `java.lang.String`). Each type may also have some special requirements. For instance, it would be nice to let the `boolean` parameters be defined as `false` if the flag character is present, rather than always `true` if present. It would also be useful to define a valid range for an `int` value.

I'll handle these differing requirements by using a base class for all parameter definitions, subclassing it for each specific type of parameter. This approach lets the application supply the parameter definitions to the library as an array of instances of the base parameter definition class, while the actual definitions can use the specific subclass matching each parameter type. For the business plan generator example, this could take the form shown in Listing 2:

Listing 2. Parameter definitions for business plan generator

```
private static final ParameterDef[] PARM_DEFS = {
    new BoolDef('c', "m_isConcise"),
    new IntDef('f', "m_initialRevenue", 10, 10000),
    new FloatDef('g', "m_growthRate", 1.0, 100.0),
    new StringDef('n', "m_productDescription")
}
```

With the allowed parameters defined in an array, the call from the application program to the argument processing code can be kept as simple as a single call to a static method. To allow for added arguments beyond those defined in the parameter array (either required values or variable length sets of values), I'll have the call return the actual number of arguments processed. This lets the application check for additional arguments and use them appropriately. The end result looks like Listing 3:

Listing 3. Using the library

```
public class PlanGen
{
    private static final ParameterDef[] PARM_DEFS = {
        ...
    };

    public static void main(String[] args) {

        // if no arguments are supplied, assume help is needed
```

```
        if (args.length > 0) {

            // process arguments directly to instance
            PlanGen inst = new PlanGen();
            int next = ArgumentProcessor.processArgs
                (args, PARM_DEFS, inst);

            // next unused argument is output file name
            if (next >= args.length) {
                System.err.println("Missing required output file name");
                System.exit(1);
            }
            File outf = new File(args[next++]);
            ...
        } else {
            System.out.println("\nUsage: java PlanGen " +
            "[-options] file\nOptions are:\n  c  concise plan\n" +
            "f  first year revenue (K$)\n  g  growth rate\n" +
            "n  product description");
        }
    }
}
```

The only part remaining is the handling of error reporting (such as an unknown parameter flag character or a numeric value out of range). For this purpose I'll define `ArgumentErrorException` as an unchecked exception to be thrown if one of these errors occurs. If this exception isn't caught, it will immediately kill the application with an error message and stack trace dumped to the console. As an alternative, you can catch this exception directly in your code and handle it some other way (perhaps printing out the actual error message along with usage information, for instance).

Implementing the library

For the library to use reflection as planned, it needs to look up the fields specified by the array of parameter definitions and then store the appropriate values to these fields from corresponding command line arguments. This task could be handled by just looking up the field information as needed for the actual command line arguments, but I've instead made a choice to separate the lookup from the usage. I'll find all the fields in advance, then just use the information that's already been found during processing of the arguments.

Finding all the fields in advance is a defensive programming step to eliminate one of the potential problems with using reflection. If I only looked up the fields as needed, it would be easy to break a parameter definition (for instance, by mistyping the corresponding field name) without realizing that anything was wrong. There would be no compile-time errors because the field names are passed as `String`s, and the program would even execute fine as long as no argument matching the broken parameter definition was specified on the command line. This type of masked error can easily result in shipping broken code.

Given that I want to look up the field information before actually processing the arguments, Listing 4 shows a base class implementation for the parameter definitions with a `bindToClass()` method that handles the field lookup.

Listing 4. Base class for parameter definitions

```
public abstract class ParameterDef
```

```
{
    protected char m_char;          // argument flag character
    protected String m_name;        // parameter field name
    protected Field m_field;        // actual parameter field

    protected ParameterDef(char chr, String name) {
        m_char = chr;
        m_name = name;
    }

    public char getFlag() {
        return m_char;
    }

    protected void bindToClass(Class clas) {
        try {

            // handle the field look up and accessibility
            m_field = clas.getDeclaredField(m_name);
            m_field.setAccessible(true);

        } catch (NoSuchFieldException ex) {
            throw new IllegalArgumentException("Field '" +
                m_name + "' not found in " + clas.getName());
        }
    }

    public abstract void handle(ArgumentProcessor proc);
}
```

The actual library implementation involves a few classes beyond what I've mentioned in this article. I'm not going to go through the whole list here, since most are irrelevant to the reflection aspect of the library. What I will mention is that I chose to store the target object as a field of the `ArgumentProcessor` class and implement the actual setting of a parameter field within this class. This approach gives a simple pattern for argument processing: the `ArgumentProcessor` class scans the arguments to find parameter flags, looks up the corresponding parameter definition for each flag (which will always be a subclass of `ParameterDef`), and calls the `handle()` method of the definition. The `handle()` method in turn calls a `setValue()` method of the `ArgumentProcessor` after interpreting the argument value. Listing 5 shows a partial version of the `ArgumentProcessor` class, including the parameter binding calls in the constructor and the `setValue()` method:

Listing 5. Partial listing of main library class

```
public class ArgumentProcessor
{
    private Object m_targetObject;  // parameter value object
    private int m_currentIndex;     // current argument position
    ...
    public ArgumentProcessor(ParameterDef[] parms, Object target) {

        // bind all parameters to target class
        for (int i = 0; i < parms.length; i++) {
            parms[i].bindToClass(target.getClass());
        }

        // save target object for later use
        m_targetObject = target;
    }

    public void setValue(Object value, Field field) {
        try {
```

```
            // set parameter field value using reflection
            field.set(m_targetObject, value);

        } catch (IllegalAccessException ex) {
            throw new IllegalArgumentException("Field " + field.getName() +
                " is not accessible in object of class " +
                m_targetObject.getClass().getName());
        }
  }

    public void reportArgumentError(char flag, String text) {
      throw new ArgumentErrorException(text + " for argument '" +
        flag + "' in argument " + m_currentIndex);
    }

    public static int processArgs(String[] args,
        ParameterDef[] parms, Object target) {
        ArgumentProcessor inst = new ArgumentProcessor(parms, target);
        ...
    }
}
```

Finally, Listing 6 shows a partial implementation of the parameter definition subclass for `int` parameter values. This includes an override of the base class `bindToClass()` method (from Listing 4) that first calls the base class implementation and then checks that the field found matches the expected type. The subclasses for other specific parameter types (`boolean`, `float`, `String`, and so on) are very similar.

Listing 6. int parameter definition class

```
public class IntDef extends ParameterDef
{
    private int m_min;              // minimum allowed value
    private int m_max;              // maximum allowed value

    public IntDef(char chr, String name, int min, int max) {
        super(chr, name);
        m_min = min;
        m_max = max;
    }

    protected void bindToClass(Class clas) {
        super.bindToClass(clas);
        Class type = m_field.getType();
        if (type != Integer.class && type != Integer.TYPE) {
            throw new IllegalArgumentException("Field '" + m_name +
                "'in " + clas.getName() + " is not of type int");
        }
    }

    public void handle(ArgumentProcessor proc) {

        // set up for validating
        boolean minus = false;
        boolean digits = false;
        int value = 0;

        // convert number supplied in argument list to 'value'
        ...

        // make sure we have a valid value
        value = minus ? -value : value;
        if (!digits) {
```

```
            proc.reportArgumentError(m_char, "Missing value");
        } else if (value < m_min || value > m_max) {
            proc.reportArgumentError(m_char, "Value out of range");
        } else {
            proc.setValue(new Integer(value), m_field);
        }
    }
}
```

Closing the library

In this article, I've run through the design of a library for processing command line arguments as an example of reflection in action. This library makes a good illustration of how to use reflection effectively -- it simplifies application code without sacrificing significant performance. How much performance is sacrificed? In some quick tests on my development system, a simple test program averaged about 40 milliseconds longer to run with argument processing using the full library, as compared to no argument processing at all. Most of that time represents the loading of the library classes and other classes used by the library code, so even for applications with many command line parameters defined and many argument values, it's unlikely to be much worse than this. For my command line applications, the additional 40 milliseconds isn't something I'm going to notice.

The full library code is available from the link in Resources. It includes several features I've left out of this article, including such niceties as hooks to easily generate a formatted list of parameter flags and descriptions to help supply usage instructions for an application. You're welcome to use the library in your own programs and extend it in any way you find useful.

Now that I've covered the basics of Java classes in Part 1, and the principles of the Java Reflection API in Part 2 and Part 3, the rest of this series is going to head off along the less-traveled path of bytecode manipulation. I'll start off easy in Part 4 with a look at the user-friendly Javassist library for working with binary classes. Do you want to try transforming methods, but are reluctant to start programming in bytecode? Javassist may be just the tool to suit your needs. Find out how next month.