Courage in Profiles

James W. Cooper

Toc: How to use Properties or Windows-style ini files to store program parameters.

Deck: Change program parameters without recompiling.

You don't have to write Java programs for very long before you find yourself needing a way to handle variable data without recompiling. If there are just a few values that change, you could use the command line argument strategy to pass them into the program. But, let's face it, most users and probably most programmers are not comfortable typing long command strings into a window anymore. It's clumsy, error-prone and a vestige of the non-windowing environments of early Unixes (Unices?). Today we expect to be able to start up a program by clicking on some convenient icon and have it know all it has to know without our typing in a lot of garp.

Another way programmers used to use to pass arguments to programs is through environment variables. This, however, can lead to a lot of clutter and possible collisions between programs' requirements. Further, the System.getenv() method is now deprecated in Java, so we really aren't supposed to use it.

The Java-endorsed method of getting information into your program is using Properties objects, and the System.getProperties() method. This code gets the system properties and lists them to the console:

```
Properties p = System.getProperties ();
p.list(System.out);
```

If you run a short program containing these statements, you will get a list of about 44 named parameters, some of which are shown below:

```
java.vm.version=1.3.0_02
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
user.dir=D:\Projects\JavaPro\Profiles\Properties
user.home=C:\Documents and Settings\jim
```

There are only a small number of these properties that get set by the system, and not many of them are that useful. Since Java does not have the concept of a "current directory," programmers often use the "user.dir" property to find out the directory that started the program. Then you can find other files in that directory since you know its path:

```
//get the current directory
String path = p.getProperty ("user.dir");
File fls = new File(path);

//get the list of files in this directory
String[] files = fls.list();
```

```
        for(int i=0; i< files.length; i++) {
            System.out.println(files[i]);
        }
```
So, if all you want to know is where the Java class file is located and read something else from that same directory, the above approach will work.

However, suppose you want to keep some more configuration information in that external file and read and write that information during the life of your Java application. You can't add to the properties that System.getProperties returns for you. But you can read and write your own properties file; as long as you have a location you know you can look in to find it.

One helpful trick is to create a Properties object using System.getProperties, and then add some more data to it and save it into a file:

```
  private final String
      propPath="d:\\Projects\\prop\\prop.save";

//constructor
    public prop() {
        Properties p =
            System.getProperties ();
        //set a new property
        System.setProperty ("Datapath", "D:\\data);
        //create an output fle
        File fl = new File(propPath);
        try {
            FileOutputStream fout = new FileOutputStream(fl);
            //and save the properties
            p.store (fout,"Saved properties");
        } catch (IOException e) {
            System.out.println ("save error");
        }
```
These save properties are in an almost editable text file, and look similar to, but not identical to the properties you list on the screen. Here are the same 6 entries as stored in the file:

```
java.vm.version=1.3.0_02
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http\://java.sun.com/
path.separator=;
user.dir=D\:\\Projects\\JavaPro\\Profiles\\Properties
user.home=C\:\\Documents and Settings\\jim
```

Then you can read this file back in later and use it using the Properties load method.

```
//now let's load that property file
        Properties newProp = new Properties();
        try {
            //make an input stream
            FileInputStream fin = new FileInputStream(fl);
            p.load (fin);   //load the file
            //and list its values
            p.list(System.out);
        } catch (IOException e) {
            System.out.println(e.getMessage ());
```

```
        }
```

Properties versus Ini-files

Windows introduced the ini-file, which has a simple two-level syntax. Thus you not only have name-value pairs but paragraph names at one higher level. A simple ini-file might look like this:

```
[Database]
      name=Customers
      password=snarch
      username=Jim
[Files]
      directory=d:\temp\
      name=mydata.txt
```

In this syntax, you ask for the value of a keyword in a particular paragraph. In the above example, there is a *name* keyword in both the Database and Files paragraphs. The convention is that leading and trailing blanks are skipped and that blank lines are skipped. Windows systems treat a blank following the equals sign as significant, but we can choose to trim it off if we like. The paragraph names and keywords are case insensitive, but the values of the keywords preserve the original case.

In addition to the additional level of tagging, ini-files are editable text files that are stored exactly as you type them, without any intervening slash or escape characters.

As you can see, it is quite simple to write a program in any number of ways to parse ini-files like this and write them back with modifications. We start with a simple class to parse and hold one line of an ini-file:

```
public class iniElement {
    private String tag, val;
    private boolean error_flag;
//parses one element of an ini-file
//into a tag and a value
    public iniElement(String s)  {
        error_flag = false;
        int i = s.indexOf("=");
        if (i >0) {
            tag = s.substring(0, i).trim().toLowerCase();
            val = s.substring(i + 1);
        } else
            error_flag = true;
    }
    public boolean error() {
        return error_flag;
    }
    public String tagName(){
        return tag;
    }
    public String valueName() {
        return val;
    }
}
```

Then we use the InputFile and OutputFile classes we have described before to read and write lines to the ini-file. These are just convenient encapsulations of the BufferedReader and BufferedWriter classes.

The main class is the iniParagraph class. It reads from one paragraph delimiter enclosed in square brackets to the next one, and stores a hash table of the tags and their values.

```
public class iniParagraph {
    private String paragraph_name;
    private InputFile f;
    private Hashtable tags;
    private String line;
    private boolean error_flag;
    private boolean eof;
    //---------------------------------
    public iniParagraph(InputFile fl) {
        f = fl;
        eof = false;
        String s = getNextLine();
        read_to_nextparagraph(s);
    }
    //---------------------------------
    public Enumeration getTags() {
        return tags.keys();
    }
    //---------------------------------
    public void setValue(String tag, String value) {
        tags.put(tag.toLowerCase(), value);  //may replace old value
        Enumeration e = tags.elements();
    }
    //---------------------------------
    private void read_to_nextparagraph(String s) {
        error_flag = false;
        tags = new Hashtable();
        while ((s != null) && (! s.startsWith("[")))
            s = getNextLine();

        //remove brackets
        if (s != null) {
            s = s.substring(1); //all but first char
            int i = s.indexOf("]");
            if (i > 0) {
                s = s.substring(0, i);
                store_paragraph(s);
            } else {
                error_flag = true;
            }
        } else {
            eof = true;
            error_flag = false;
        }
    }
    //---------------------------------
    public String getName() {
        return paragraph_name.toLowerCase();
    }
    //---------------------------------
```

```
    public String tagValue(String tagname) {
        String ans;

        ans = (String)tags.get(tagname.toLowerCase());
        if (ans == null)
            return "";
        else
            return ans;
    }
    //---------------------------------
    private void store_paragraph(String s) {
        String next;
        iniElement ini;
        paragraph_name = s;
        next = getNextLine();
        while ((next != null) && (! next.startsWith("["))) {
            ini = new iniElement(next);
            if (! ini.error()) {
                tags.put(ini.tagName(), ini.valueName());
            }
            next = getNextLine();
        }
        if (tags.size()>0)
            error_flag = false;
    }
    //---------------------------------
    public iniParagraph(InputFile fl, String pname) {
        f = fl;
        read_to_nextparagraph(pname);
    }
    //---------------------------------
    public iniParagraph(String pname)  {
        //call this constructor to create a new paragraph
        tags = new Hashtable();
        paragraph_name = pname;
    }
    //---------------------------------
    private String getNextLine() {
        line = f.readLine();
        while ((line != null) && (line.length() <1))
            line = f.readLine();
        if (line == null)
            error_flag = true;
        else
            line = line.trim();
        return line;
    }
}
```

Finally, at the highest level, we write the IniFile class, which keeps a hash table of paragraphs by name.

```
public class IniFile {
    private String filename, path, fullpath;
    private Hashtable paragraphs;
    private iniParagraph para;
    private InputFile f;
```

```
//-----------------------------------------------------
    //Opens an existing iniFile
    public IniFile (String pathname, String fname) throws IOException {
        filename = fname;
        path = pathname;
        if (path.length()==0)
            fullpath= filename;
        else
            fullpath = path +File.separator + filename;
        checkFile(fullpath);
        f = new InputFile(fullpath);
        getParagraphs();
    }
    //-------------------------------------------------------
    private synchronized void getParagraphs() {
      //make room for object table
        paragraphs = new Hashtable();
        if (! f.checkErr()) {
            para = new iniParagraph(f);

            if (! para.error()) {
                paragraphs.put (para.getName (), para);
                while ((para.nextLine()!=null)
                              &&
                  (para.nextLine().length() > 0)) {
                    para = new iniParagraph(f, para.nextLine());
                    paragraphs.put(para.getName (), para);
                }
            }
            f.close();
        }
    }
    //-------------------------------------------------------
    private synchronized void putParagraphs() throws IOException {
        OutputFile f = new OutputFile(fullpath);
        Enumeration enum = paragraphs.elements ();
        while(enum.hasMoreElements ())   {
            iniParagraph p = (iniParagraph)enum.nextElement ();
            f.println("["+ p.getName() + "]");


            Enumeration e = p.getTags();
            while (e.hasMoreElements()) {
                String tag = (String)e.nextElement();
                f.println(tag.toLowerCase()+"="+p.tagValue(tag));
            }
        }
        f.close();
    }
    //-------------------------------------------------------
    //Gets a profile string from an ini file
    public String getProfile(String para, String entry) {
        iniParagraph ini = null;
        int i =0;
        boolean found = false;
        ini = (iniParagraph)paragraphs.get (para.toLowerCase ());
        if (ini != null)  //now look for entry in that paragraph
```

```
            return ini.tagValue(entry).trim();
        else
            return "";
    }
    //-----------------------------------------------------
    //Gets a profile string from an ini file
    //returns default value if no such entry exists

    public String getProfile(String para,
                 String entry, String defalt) {
        String value = getProfile(para, entry);
        if (value.length()<1)
            value = defalt;
        return value.trim();
    }
    //-----------------------------------------------------
    //Puts a profile entry into the ini file

    public void putProfile(String para,
            String tag, String value) throws IOException {
        int i = 0;
        boolean found =false;
        iniParagraph p = null;
        Enumeration enum = paragraphs.elements ();
        while (enum.hasMoreElements () && ! found) {
            p =(iniParagraph)enum.nextElement ();
            found =(para.equalsIgnoreCase( p.getName()));
        }
        if (found) {
            p.setValue(tag, value);
        } else {
            p = new iniParagraph(para);
            paragraphs.put (p.getName (), p);
            p.setValue(tag.toLowerCase(), value);
        }
        putParagraphs();    //rewrite entire ini-file
    }
}
```

Where to Keep Them?

The real question about these parameter files is where best to keep them. In this, we are no better of than we were in using the Properties files. We still have to have a place for them. For most Java applications, the user.dir property will give you the application's current directory. But this can be misleading if the JVM is started from an upper directory of a package or if the code is in a jar file somewhere. And it can be absolutely wrong if the code is started by a servlet engine or is a bean referred to by a JSP. In these cases, the directory path will be that of the servlet class, not the bean.

I have two proposals for ways to get around this. One is to define a default directory like c:\inifiles, and keep all the inifiles for your system there. This, is a little awkward, and is certainly not very cross-platform. A slightly more flexible approach is to use the System property "user.home" to obtain the user's home directory and keep the ini-files under that path. You only need to make sure that every program reads its own ini-file. Along these lines, the way I have finally adopted is to keep each program name in a special ini-file in

the user.home directory, and have its value be the absolute path where the real ini-file for that program will be located. Thus, the ini-file in the user.home directory is just a program registry of all the Java programs that require initialization information and the paths to where those programs actual parameter files are located.

Here's a simple example:

```
//class constructor
    public TermLoader(String projectName) {
        String userPath=System.getProperty ("user.home");
        try {
            //get the program registry
            IniFile iniIndex =
                  new IniFile(userPath + "\\iniFiles.ini");
            iniFileName = iniIndex.getProfile("Programs", projectName);
            //get the actual ini-file
            ini = new IniFile(iniFileName);
        } catch (IOException e) {
            System.out.println("No ini file found:" + iniFileName);
            System.exit(1); //hard error
        }
        //start getting data
        String path = ini.getProfile("documents", "path");
```

Ini Conclusion

Properties files provide a way to read simple name-value pairs. But you can implement the Windows ini-file convention in Java and use it on any platform to provide one more level of detail. You can use these methods to devise a simple program registry that points to absolute paths for each program's ini-file. We'll look at some alternative ways in future columns. If you have some ideas for other approaches, let us hear about them.