Stating Your Preferences

James W. Cooper

Sometimes you don’t remember what you prefer. For example, when you dine at the famous French restaurant Picholine, and are confronted with their mammoth cheese trolley, did you prefer the Perail over the Coolea? It would be nice if we had written this down in our Pomme Pilot, but we probably didn’t.

Java programmers have frequently faced the same problem. Short of hard coding a bunch of constants, how can we write an application that remembers data from one application to another. This is particularly difficult, when different users may have different preferences or requirements (such as Abbaye de Belloc or Zamorano).

The recent release of JDK 1.4 has provided us with an excellent new API for solving such preference problems. Specifically, you can save data in a global database, labeled by user and application and retrieve it at another time. The structure and location of that database are not specified, and it is designed for robust access, but not necessarily for speed. In Windows, for example, these data are stored in the Registry, and in Linux, the data are stored in the file system. All of this is carried out below your eye level, and from the point of view of the Java programmer, it is unimportant how it is stored.

The API

The java.util.prefs package contains just a few classes, most notably the abstract Preferences class. While this class is abstract, it does contain a few concrete static methods for obtaining a Preference object. Preferences are organized into two trees: System and User, which could contain related information. The idea, of course, is that some program data is global, and regardless of who runs it the some constant values apply. And, on the other hand, some data is user specific, and each user should be able to store and retrieve his own unique values.

Data in the Preferences database can be strings, just as it can in Properties files and ini-files, but the data can also be integer, long, float, double, Boolean or a byte array. Each entry is limited to 8192 bytes, however.

To begin using Preferences, you need to get an instance of either the userNode or the systemNode for your program package:

```
Preferences prefs = Preferences.userNodeForPackage(getClass());
```

Note that the Preferences API acts a Factory and returns an instance of a Preferences object using the current class as an argument. What it actually does is look at that class’s *package* and return a Preferences object for that package. The Preference system then stores each tag name you select prefixed with the package name.

So, let us suppose we want to store the final size of the window when we close it, so it will be that size when we re-open it in a later session. When the system closes, we just get the current size and store it in the Preferences database:

```
     private final String stXsize="xsize";
```

```
      private final String stYsize="ysize";
      private final String appTitle ="prefdemo";

public void WindowClose() {
      Dimension dim = frame.getSize();
      prefs.putInt(appTitle + xsize, dim.width);
      prefs.putInt(appTitle + ysize, dim.height);
      System.exit(0);
}
```

Note that we are prefixing the name of the tag with the title of the actual application. This helps make each entry unique. So in this example, the tags are actually named

```
prefdemoxsize
```

and

```
prefdemoysize
```

You can name a tag anything you want, although it is conventional to prefix the variable name with the name of the application. Names may be any set of numbers and characters, but cannot contain the forward slash "/" character.

Since you name the variables by application program name, you could access and change these variable from other applications as long as you know the program name you are looking for. This allows you to write an administration program that could set default parameters for other programs in the same package by simply using the names of those programs.

In this example, we are storing integer values. For other types of data there are analogous methods: put Boolean, putByteArray, putDouble, putFloat, and putLong.

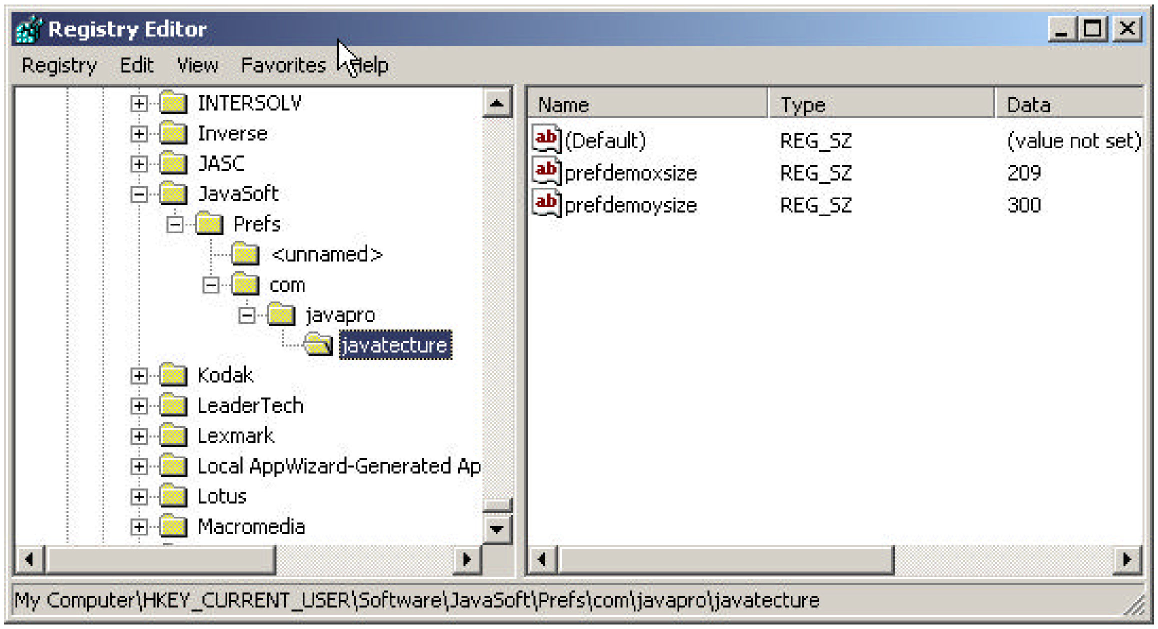
Not every small program is put into a package, however. If you simply write a program as part of the default package, (the unnamed package), the data are stored as part of a package called "<unnamed>".

To get the values back again, when you restart the program, you just reverse this process:

```
public int getXsize() {
      return prefs.getInt(appTitle + stXsize, 400);
}
```

Note that the get methods always require a default value as a second argument in case no value has yet been entered in the Preferences database.

In Figure 1, we show how the Registry stores these data. You will see our program preference under the com.javapro.javatecture package.



Name	Type	Data
(Default)	REG_SZ	(value not set)
prefdemoxsize	REG_SZ	209
prefdemoysize	REG_SZ	300

My Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs\com\javapro\javatecture

Figure 1 - Preference data for our simple demo program

Chacun a son Gout

Now, every user may have his own preferences, even for something as simple as window size and position. So the preference system allows for this, by keeping the user branch of the Preference tree by user. So it is possible to store your favorite preferences for window size and then log off and log back on as a different user and set them differently. Note that there is nothing specifically in our application code to switch between users: this is handled automatically by the Preferences system.

If instead of accessing the User preferences, you access the System preferences, these are truly user-independent and will save return the same values regardless of which user ID they are obtained from.

```
Preferences pref = systemNodeForPackage(getClass());
```

Since you might conceivably keep around 2 or more Preferences objects, some user and some system, the API cleverly provides an isUserNode() method so you can determine which kind of Preferences object you are dealing with.

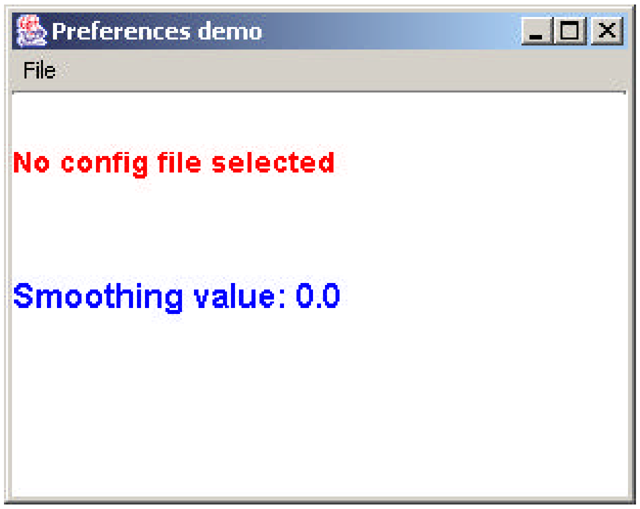
More Preferences

I write Java applications all the time that do reasonably useful things within my work. Very frequently, these programs operate using some set of data parameters which I like to vary over time. For example, the name of the database I interact with could well be a user-level Preference parameter, but the value of some computation parameters like, for instance, a data smoothing constant, could vary from run to run within a single day. I call this third kind of preference an “application-level” preference, and would like to be able to change a number of these between program runs without either building an excessively complex user interface, or saving and restoring a lot of data in the Registry.

It is this sort of application-level data that I would normally store in a initialization file. Such a file is a pure text file that I can easily edit between runs to affect the outcome of my computation.

The way I handle this using the Preferences API is to specify a Configuration filename and path from within the program one time when we first run it and then get that filename from the Preferences database each time the program starts.

Let's consider the following simple program which can read in Windows-style ini-files as configuration files. When it first starts, it looks in the Preferences database for a configuration filename, and not finding one, it displays a message that it has no configuration file, as we see in Figure 2.



No config file selected

Smoothing value: 0.0

Figure 2 - Our application before a configuration file is specified

You can see us selecting the menu item to obtain a configuration file in Figure 3

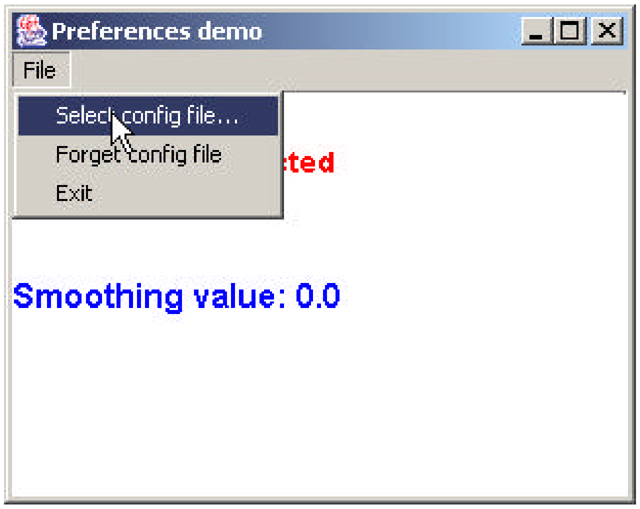
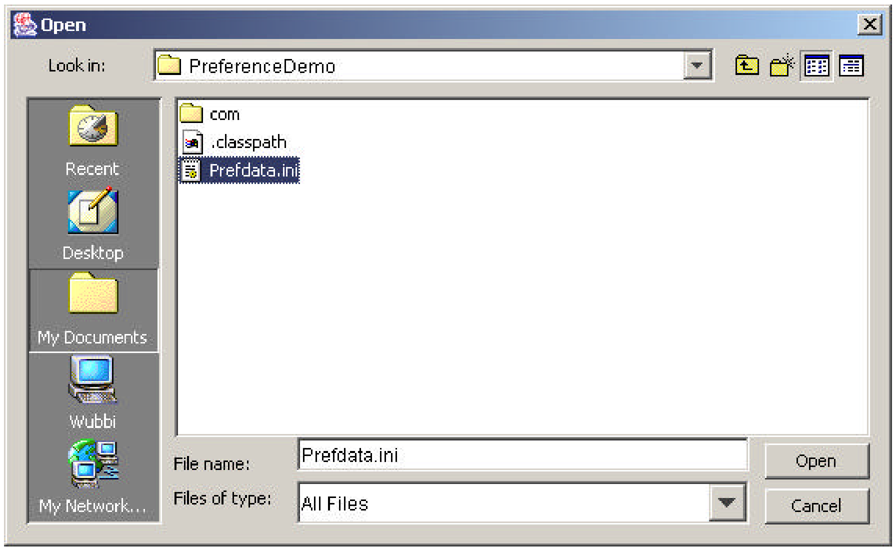


Figure 3 - The menu item to select a configuration file.

And the JFileChooser box where we select it in Figure 4.



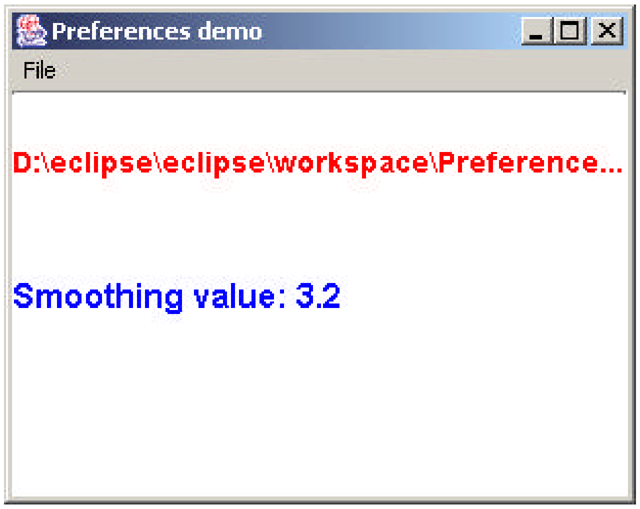
File name: Prefdata.ini
Files of type: All Files

Figure 4 - Selecting the configuration file.

Here is the code the launches the JfileChoose and saves the filename:

```
public void selectConfigFile() {
      JFileChooser chooser = new JFileChooser();
      int returnVal = chooser.showOpenDialog(frame);
      if (returnVal == JFileChooser.APPROVE_OPTION) {
            configFile = chooser.getSelectedFile().getAbsolutePath() ;
            prefs.put(appTitle + configName, configFile);
      }
}
```

After this, our application displays the name of the file and the current value of the smoothing constant as you see in Figure 5.



D:\eclipse\eclipse\workspace\Preference...

Smoothing value: 3.2

Figure 5 - The application showing the configuration file and the smoothing constant it contains.

The actual configuration file just contains the following two lines:

```
[data]
smoothing=3.2
```

We read this file using the ini-file classes we developed in our November, 2001 column (Courage in Profiles). Here is an example:

```
public float getSmoothing() {
      File fini = new File(configFile);
      IniFile ini = null;
      float smooth = 0.0f;
      if (fini.exists()) {
            try {
                  ini = new IniFile(configFile);
            } catch (IOException e) {
                  smooth = 0.0f;
            }
            smooth = new Float(
                  ini.getProfile("Data", "smoothing")).floatValue();
      }
      return smooth;
}
```

Using Our Application

So every time we start up this application, it goes to the Preferences system and fetches the xsize and ysize and the configuration filename. Then if that filename exists it loads the current smoothing constant. This is very convenient, because I can now vary several application level parameters by just editing the configuration file, and running the application with these new values.

I structured this program so all of the computations for manipulating preferences are in a simple PMediator class, that handles the reading and writing of Preferences and configuration file parameters. You can see the entire class in Listing 1.

```
/**
 * Preferences file Mediator
*/
package com.javapro.javatecture;
import java.util.prefs.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class PMediator {
      private Preferences prefs;
      private JFrame frame;
      private String appTitle;
      private String configFile;
      private final String configName = "configFile";
      private final String stXsize="xsize";
      private final String stYsize="ysize";
      public PMediator(JFrame frame, String appTitle) {
            prefs = Preferences.userNodeForPackage(getClass());
            this.frame = frame;
            this.appTitle = appTitle;
            configFile = prefs.get(appTitle + configName, "");
      }
      //--------
      public float getSmoothing() {
            File fini = new File(configFile);
            IniFile ini = null;
            float smooth = 0.0f;
            if (fini.exists()) {
                  try {
                        ini = new IniFile(configFile);
                  } catch (IOException e) {
                        smooth = 0.0f;
                  }
                  smooth = new Float(ini.getProfile("Data",
"smoothing")).floatValue();
            }

            return smooth;
      }
      //--------
      public void forgetConfig() {
            configFile = "";
            prefs.remove(appTitle + configName);
      }
      //--------
      public String getConfigFilename() {
            if (configFile.length() < 1)
                  return "No config file selected";
            else
                  return configFile;
      }
      //--------
      public void selectConfigFile() {
            JFileChooser chooser = new JFileChooser();
            int returnVal = chooser.showOpenDialog(frame);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
                  configFile =
chooser.getSelectedFile().getAbsolutePath() ;
                  prefs.put(appTitle + configName, configFile);
            }
      }
      //--------
      public void WindowClose() {
            Dimension dim = frame.getSize();
            prefs.putInt(appTitle + stXsize, dim.width);
            prefs.putInt(appTitle + stYsize, dim.height);
            System.exit(0);
      }
      //--------
      public int getXsize() {
            return prefs.getInt(appTitle + stXsize, 400);
      }
      //--------
      public int getYsize() {
            return prefs.getInt(appTitle + stYsize, 300);
      }
}
```

Listing 1 –The PMediator class for manipulating Preferences and the configuration file.

You can see how we use this class in the fragment of the main Swing application show in Listing 2.

```
public PDemo() {
      super("Preferences demo");          //set title bar
      med = new PMediator(this, "prefdemo");
      super.setMediator(med);       //tell frame about mediator
      int xSize = med.getXsize();   //get the window sizes
      int ySize = med.getYsize();
      setSize(xSize, ySize);
      setGUI();                           //set up the display
      setVisible(true);             //display the window
}
//------------
private void setGUI() {
      setMenus(); //create the menus
      //layout the screen
      getContentPane().setLayout(new BorderLayout());
      JPanel jp = new JPanel();
      jp.setLayout(new GridLayout(3, 1));
      getContentPane().add(jp);
      //display the configuration time
      label = new JLabel(med.getConfigFilename());
      label.setFont( new Font("SansSerif", Font.BOLD,14  ));
      label.setForeground( Color.red);
      jp.add("Center", label);
      //display the current smoothing constant
      smooth = new JLabel();
      smooth.setFont( new Font("SansSerif", Font.BOLD,16  ));
      smooth.setForeground(Color.blue);
      jp.add("Center", smooth);
      jp.setBackground(Color.white);
      loadSmooth();     //get the smoothing function
}
//-------------
//get the current smoothing constant and display it
private void loadSmooth() {
      smooth.setText("Smoothing value: " +
            new Float(med.getSmoothing()).toString());
}
```

Listing 2 – A section of the main application showing how the PMediator class is used.

Summary

You can store any kind of preference information in the Preferences system, whether you know a Camembert from a ComboBox. You can even use it effectively to provide parameters to JSP beans and servlets more conveniently. I’m already using it every day.