Using the Preferences plugin to save plugin preferences across Sibelius sessions

Bob Zawalich January 29, 2011

The Preferences plugin (Plugins > Other > Preferences) was written by Hans-Christoph Wirth to allow us to save plugin settings across Sibelius sessions. It grew out of a mechanism we had been using that wrote settings to either Sibelius scores or text files, and has the additional benefit of providing an editable user interface for the data stored in its database.

Its use is described in the ManuScript document, but it is not the most obvious description I have ever used, and there are a few things that can catch you out.

When I need to save preferences, I usually start with a template plugin that sets up the access to preferences and provides me with stub methods. This document will describe how this was done, and will this provide at least an example of a plugin that works.

There are a large number of plugins on the download page that use this code, and include examples of varying complexity, and these can be examined as well.

The Preferences code in the plugin *Minimum Plugin Preferences*

I try to isolate the calls to the Preferences code to the Run() method. Typically, what needs to be done is to initialize the database, get the saved preferences if any, run a dialog, save the changes made in the dialog, and close the Preferences database.

The example Run() method looks like this:

```
if (Sibelius.ProgramVersion < zg_Sib6Version)
{
      MyMessageBox(_msgVersionTooEarly);
      return False;
}

// update zg_VersionNumber when changes are made.
_Version = BuildVersionText(zg_VersionNumber);

if (Sibelius.ScoreCount = 0)
{
      Sibelius.MessageBox(_ScoreError);
      return False;
}

score = Sibelius.ActiveScore;
selection = score.Selection;
```

```
if (score.StaffCount = 0)
{
    MyMessageBox(_ScoreError);
      return False;
}

fProcessEntireScore = False;
if (IsEmptySelection(score, True) = True)
{
      fContinue = MyYesNoMessageBox(_msgSelectWholeScore); //True means
Yes, continue and process score
      if (fContinue = False)
      {
            return False; // they said no, so stop the plugin
      }
      fProcessEntireScore = True;
}

//OpenPreferences();
//GetPreferences();

//ok = DoDialog();
//if (ok = False)
//{
      //CleanupPreferences();
//    return False;
//}

//SavePreferences();

score.Redraw = False;
selection.StoreCurrentSelection();

if (fProcessEntireScore = True)
{
      selection.SelectPassage(1, score.SystemStaff.BarCount, 1,
score.StaffCount); // select all staff items
}

numColored = ProcessSelection(score, selection);

//CleanupPreferences();

selection.RestoreSelection(); // now the original selection is
restored.
score.Redraw = True;
```

There is some generic initial setup, then a block of code that is commented out in the example – the first thing I do is uncomment it:

```
//OpenPreferences();
//GetPreferences();

//ok = DoDialog();
//if (ok = False)
//{
      //CleanupPreferences();
//    return False;
//}
```

(I generally leave the code commented out until I know what I will need to save and restore, and that is not usually true until I am into the plugin for a while).

OpenPreferences opens the data base, and in my code, it sets a global variable `g_valPreferencesOpen` for the other Preferences routines. I use the global because I often call SavePreferences out of a dialog handler routine, to which parameters cannot be passed. The code looks like this:

```
// set g_valPreferencesOpen if ok, -1 otherwise

strName = "" & _PluginMenuName;
g_valPreferencesOpen = Preferences.Open(strName,
zg_PreferencesVersionNumber);

return g_valPreferencesOpen;
```

You open the database using an identifier, and _PluginMenuName is the easiest choice. The cast to strName suggests that there were problems passing a global variable to Open, but I do not remember the details of any problems. But this scheme definitely works.

`zg_PreferencesVersionNumber` is another global, and it is the version of Preferences. This has not changed in a long time, (`zg_PreferencesVersionNumber "020000"`)so it would probably not hurt to use the hard-coded value inline, but I like to use a global so I can change it if needed without messing with the actual code.

Open returns

```
* -2 other error
* -1 library does not support requested feature set
* 0 no common preferences database found
* 1 no preferences found for current plug-in
* 2 preferences for current plug-in loaded
```

and anything greater or equal to 0 is useable. I check the value in `GetPreferences()`, and if nothing is available, I use default values rather than cancelling the plugin.

GetPreferences() is where it gets a bit complex, though once you are past the initial setup, not much code changes from plugin to plugin. Let's skip it for the moment, though, and handle the others, which are generally simpler.

`CleanupPreferences()` is simple. The code just looks like:

```
if (g_valPreferencesOpen >= 0)
{
      Preferences.Close();
}

g_valPreferencesOpen = -1;  // reset
```

Once you have opened the database, be sure that you close it via `ClosePreferences`, even if you exit on an error. So in the example, if the user cancels the dialog you still need to close Preferences:

```
ok = DoDialog();
if (ok = False)
{
      CleanupPreferences();
      return False;
}
```

`SavePreferences()` can be complex if you are saving multiple groups of data in different scopes, but commonly you are just saving strings or numbers or arrays. If you save Booleans, you should convert the values to a 0 or a 1 before you save. I always save the version number of the plugin as part of the data so I can check when I get the preferences if the version of the plugin is >= the version in which the data was stored.

```
// store these in the central preferences file using Hans-Christoph"s
Preferences library

if (g_valPreferencesOpen < 0)  // g_valPreferencesOpen is return value
from ptrPreferences.Open. > = 0 means options can be written
{
      return False;
}

Preferences.SetKey("Version", 0 + zg_VersionNumber);

xxx // force crash if used without changing
Preferences.SetKey("MoveInVoice", "" & g_strVoice);

return True;
```

In my sample code I add xxx to make a syntax error in case I forgot to update this routine. If I run it, it will crash as a reminder. The line that calls SetKey for MoveInVoice is just an example line...

Note that if I am saving off a global variable like g_strVoice, I concatenate "" to get its value rather than its address. Similarly, for a number, add 0 (as in the version number). I use a routine called TrueFalseAsNumber() to convert a boolean to a 0 or 1:

```
Preferences.SetKey('fStrings', TrueFalseAsNumber(dlg_fStrings));
```

You can store arrays easily as well.

```
Preferences.SetArray('RGBList', g_arrRGBValues, -1);
```

For an example of a relatively simple plugin that stores several keys and an array see Color Enharmonic Pitches.SavePreferences() at http://www.sibelius.com/download/plugins/index.html?plugin=263.

For a more complex plugin that saves multiple sets of data in different scopes, see Add Fingering To Notes.DoSaveCustom() at http://www.sibelius.com/download/plugins/index.html?plugin=318

GetPreferences()

There is a fair bit of code here, but the part that has to change is pretty small. For the most part you just need to Get the same set of variables you Set in SavePreferences().

Here is the sample code:

```
// get preferences from database if available, otherwise use global
defaults

if (g_valPreferencesOpen < 2)  // g_valPreferencesOpen is return value
from ptrPreferences.Open. 2 = Preferences found
{
      // trace("GetOptions - no preferences file found, use defaults -
open returns " & g_valPreferencesOpen);
      return -1;
}

ver = Preferences.GetKey("Version");
//trace("GetOptions ver, zg_VersionNumber,
zg_VersionSavedPreferencesMin = "  & ver & ", " & zg_VersionNumber & ",
" &  zg_VersionSavedPreferencesMin);
if (ver = Preferences.VOID)  // something wrong, just clear without
asking
{
```

```
      Preferences.RemoveAllIds();  // removes all preferences for
current plugin.
      return -1;
}

if (ver > zg_VersionNumber)
{
      MyMessageBox(_DefaultsFormatError);
      return -1;
}

//dlg_strLineStyle = "" & GetPrefKeyNoVoid("strLineStyle", ("" &
dlg_strLineStyle));
//dlg_fAddLine = 0 + GetPrefKeyNoVoid("fAddLine",
(TrueFalseAsNumber(dlg_fAddLine)));

return 1;
```

The code is assuming that if it cannot find any Preferences data, the global variables have been already set up with appropriate defaults. So do that in the caller if you want values other that what is stored in the global variables themselves (though for me, that is the usual situation).

The lines that change every time are the commented out calls to `GetPrefKeyNoVoid()`, and there should be 1 entry for each value saved. `GetPrefKeyNoVoid()` calls `GetKey`, and if that call returned the special void value, it uses the value that was previously in the global. Look at `GetPrefKeyNoVoid()` for details.

If you use multiple scopes, this routine will be more complex. You need to declare local variables at the start, and be careful only to declare them when the database is first created, or the variables tend to disappear. Here is an example from `Add Fingering To Notes`, showing a variant of the starting code above. See the plugin itself to see a full implementation of multi-scope variables.

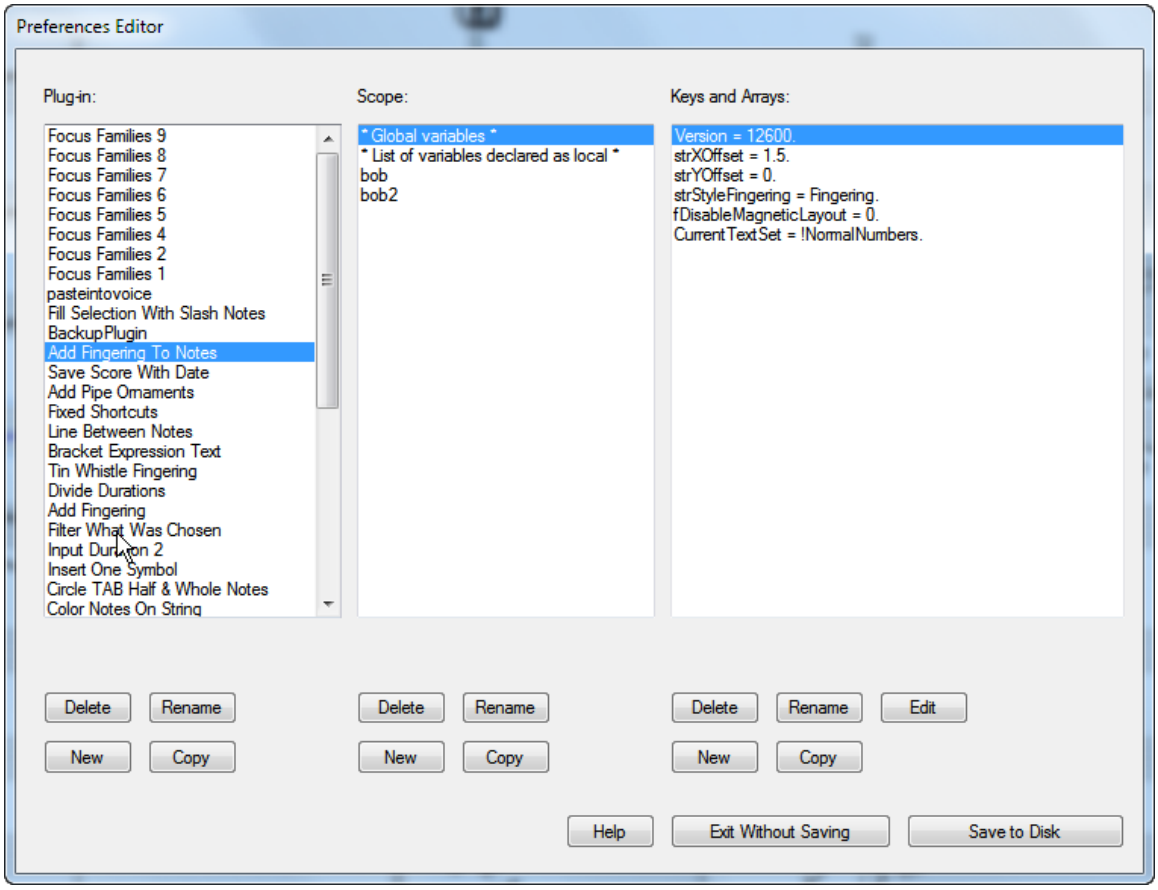
In my experience you do not need them very often, and it is a bit messy, though very powerful, when you do need them.

```
if (g_valPreferencesOpen < 2)  // g_valPreferencesOpen is return value
from ptrPreferences.Open. 2 = Preferences found
{
      if ((g_valPreferencesOpen = 0) or (g_valPreferencesOpen = 1))
      {
            // only need to declare these as local once when the DB is
first created
            Preferences.DeclareIdAsLocal('TextSet');
            Preferences.DeclareIdAsLocal('strXOffsetC');
            Preferences.DeclareIdAsLocal('strYOffsetC');
            Preferences.DeclareIdAsLocal('strStyleFingeringC');
            Preferences.DeclareIdAsLocal('fPositionAtNoteheadC');
            Preferences.DeclareIdAsLocal('fDisableMagneticLayoutC');
      }
```

```
      // trace('GetOptions - no preferences file found, use defaults -
open returns ' & g_valPreferencesOpen);
      return -1;
}
```

Checking what was written

Plugins >Other >Preferences has a pretty cool editing interface that will show you what was written, and let you edit the data in the data file (which is SibeliusPluginPreferences.dat, stored in the user Sibelius 6 folder, above the Plugins folder). Here is an example of what I see when I run it.



I hope this has been useful. You should find the `Minimum Plugin Preferences` plugin in the same zip file where you found this file.