LANGUAGE SUPPORT IN SKETCHUP

LibTraductor.rb

A RUBY LIBRARY FOR SKETCHUP V5 AND V6

Foreword

When I designed the script ***bezierspline.rb*** I indeed bumped into the problem of language translation, both as a script programmer and as a script user. I made some research on what existed in terms of practices and tools and found very little. In general, scripts are written in a single language (English usually), and then the script is converted by someone into other languages, by modifying strings in the code and renaming the script file. For instance you have *bezier.rb* and *fr_bezier.rb* on the Crai Ruby Depot.

There are a few issues with this approach:

- Having several versions of the file makes it more difficult to keep potential upgrades and bug fixing in sync.
- Some errors can be made when translating strings embedded in the source code. You can forget to translate some, or accidentally translate strings that you should not.

So I came up with this small library *LibTraductor.rb* and a set of practices to rationalize a little bit the coding and the language translation management of Sketchup scripts.

The principle is to provide a way to define and use strings that carry their own translated version.

The macro provides utility functions to support:

- The definition and usage of multi-language strings, including substitution patterns with %1, …, %9
- Mass assignment of strings to variables
- The designing of dialog boxes, with language translation and built-in validation
- A few utilities on hash arrays (marshaling, un-marshaling, pretty print)

Usage: just drop the macro file ***LibTraductor.rb*** in the Plugins folder of Sketchup, and in your script just insert a statement:

```
require "LibTraductor.rb".
```

Test utility: I published a test macro ***LibTraductor_test.rb*** to show some examples. It adds 4 menu items in the *Plugins* menu of Sketchup.

Compatibility: the macro should work with **Sketchup v5 and v6** (Pro and free versions, English and French). I tested it on Windows XP and Windows Vista. I don't know however if it works on Mac

1. Multi-Language Strings

1) Language setup

The macro uses the standard 2-character language code, for instance 'EN' for English, 'FR' for French, 'SP' for Spanish, etc…The 'current' language is normally determined by the **language of your operating system**, actually what is returned by a call to `Sketchup.get_locale`.

You can also force the language by editing the configuration file *LibTraductor.def* and declaring the 2-letter code of your language for the parameter **TRADUCTOR_DEFAULT**: For instance. For instance if **TRADUCTOR_DEFAULT** = "**FR**", then all texts will be displayed in French, regardless of your operating system. The statement **TRADUCTOR_DEFAULT** = "" will put back the default to the operating system language.

You can get the current language by calling `curlang = Traductor.get_language`

You can change the current language by calling `Traductor.get_language newlang`. If `newlang` is nil or empty, then the default language is reset.

Note: I know there is actually a trend to go down to idiomatic versions of the same language, for instance 'EN_US' for American English, versus 'EN_EN' for Shakespeare English, but I thought that it was too much of an effort in the context of Sketchup extensions, where language translation is not so critical and where anyway the Sketchup application is itself available in very few languages (so far only English for Version 6, and just recently French).

2) Declaring the multi-language strings

The convention for multi-language strings is to embed a sequence of strings separated by language delimiters made of the 2-character language code, comprised between vertical bars. For instance

```
Mystring = "Silence |EN| Welcome  |DE| Wilkommen |FR| Bienvenue"
```

The first part of the string, here 'Silence', is the **default string** when the language is not supported. In most cases, it would be English, so that you don't really need to use |EN|.

For convenience, the macro supports the declaration of translatable strings as an array of strings instead. This is some time easier when you declare long texts. For instance:

```
Mystring = ["Silence", "|EN| Welcome", "|DE| Wilkommen",
            "|FR| Bienvenue"]
```

which you can also write in a more readable way:

```
Mystring = ["Silence",
            "|EN| Welcome",
            "|DE| Wilkommen",
            "|FR| Bienvenue"]
```

or even with mixing both styles:

```
Mystring = ["Silence |EN| Welcome |DE| Wilkommen",
            "|FR| Bienvenue"]
```

Note that when translating the string, the macro will **ignore the leading and trailing spaces between the `|XX|` delimiters**. So,

```
Mystring = "   Welcome Fredo     |FR|    Salut Fredo    "
```

is actually treated as

```
Mystring = "Welcome Fredo|FR|Salut Fredo"
```

3) Using multi-language strings in your code

The macro provides a versatile method, **Translator.s**, to compute and return the relevant translated string from a multi-language string.. There are several ways of calling this method[1].

1) With one argument:

It is a good habit to define message strings as constants:

```
MY_STRING = "Silence |EN| Welcome  |DE| Wilkommen |FR| Bienvenue"
```

```
Traductor.s MY_STRING          → "Wilkommen" (if language is DE)
```

You can of course use Traductor.s with normal strings, deferring the time when you will take care of the translation:

```
Traductor.s "Hello world"                    → "Hello World"
```

Remember however that the method removes leading and trailing spaces:

```
Traductor.s "      Hello world        "      → "Hello World"
```

In such case, you can even pass the argument as a string with embedded evaluation

```
version = 6
Traductor.s "Hello Sketchup v.#{version}"   → "Hello Sketchup v.6"
```

2) With multiple arguments to substitute tokens:

If you pass more arguments, they will be substituted in the string following the convention %1, %2, …till %9.

For instance:

```
version = 6
Traductor.s "%1 Sketchup v.%2", 'Hello', version
 → "Hello Sketchup v.6"
```

The order of replacement tokens has no importance, and each can be used more than once:

```
version = 6
Traductor.s "%2 Sketchup v.%1, %2", version, 'Hello'
 → "Hello Sketchup v.6, Hello"
```

Arguments that have no replacement tokens are simply not used.

```
version = 6
Traductor.s "Hello Sketchup v.%1", version, 'bonjour'
 → "Hello Sketchup v.6"
```

3) With default:

If you pass a second argument, but no substitution token in the string, then it will be used as the default value returned in case the translatable string is nil.

```
sresult = get_a_string a, b, c       #this call may return nil
Traductor.s sresult, "sresult was empty |FR|sresult est vide"
```

→ **"sresult est vide"** if it happens that **sresult** is nil (and language is French).

---

[1] Be careful to call the method with a capital 'T' in Traductor. <traductor.s> will generate an error

Finally, note that if there only one argument and it is nil, then the method does not generate an error but returns an empty string. So,

```
Traductor.s nil          → ""
```

4) Alternate form

The Traductor module also provides an alternate form of call, with the brackets. However, this form should be taken with care, as **you must have NO space between 'Traductor' and the opening bracket**.

```
Traductor["hello |FR|Bonjour"]      → "Bonjour"
```

But

```
Traductor ["hello |FR|Bonjour"]     → will generate an error
```

4) Mass assignment of multi-language strings to variables

In the real world of coding, you have to adopt a strategy for including multi-language strings in your code.

- Either you always call `Traductor.s` when needed. The benefit is that you can then change the language dynamically. The drawback if that you will process the calculation of the translation in the current language (though this is not that long!).
- Or you compute the translations once and put them into module or class variables so that they can be used anywhere without recalculation.

The choice may also be driven by questions of code readability.

In the second case, the *libtraductor* macro provides some support to assign multi-language constants to variables, if you follow some simple naming conventions.

The general syntax is:

```
Traductor.load_translation module, const_pattern, binding, var_pattern
```

Let's assume that you have defined a series of multi-language strings in the header of your module *MyModule*.

```
BZTRS_ChangePrecision = ["Precision --> ", "|FR|Pr\écision --> "]
BZTRS_ChangeDegree = ["Control points --> ", "|FR|Pts de contr\ôle --> "]
BZTRS_OpenEnd = ["Open-ended curve", "|FR|Courbe ouverte"]
BZTRS_StartEnd = ["Between Start/End", "|FR|Courbe entre pt d\ébut et fin "]
```

In the initialization part of your module or class, you may want to assign the translated string corresponding to the current language to variables. You would simply write the following statement:

```
Traductor.load_translation MyModule, /BZTRS_/, binding, "@text_"
```

That would actually generate the following statements, in the exact context where you made the call:

```
@text_ChangePrecision = ["Precision --> ", "|FR|Pr\écision --> "]
@text_ChangeDegree = ["Contol points --> ", "|FR|Pts de contr\ôle --> "]
@text_OpenEnd = ["Open-end curve", "|FR|Courbe ouverte"]
@text_StartEnd = ["Between Start/End", "|FR|Courbe entre pt d\ébut et fin "]
```

The first argument is the **module** in which the constants are defined, usually, the module you made the call from, but not always if you group constant definitions in a central, common module.

The second argument is a regular expression to select the constants that should be translated and assigned to variables.

The third argument, `binding`, is necessary so that the variables are given the scope of the place where you invoke `Traductor.load_translation`. Don't bother too much, just pass `binding` and it will work.

The fourth argument is the replacement string. Note that if this third argument is absent, then it is taken as `"@msg_"` by default.

By following some naming conventions, you can spare time in using multi-language strings!

5) More on multi-language string support

Keep leading or trailing spaces

In some occasions you may want to keep leading or trailing spaces, which otherwise would be stripped out automatically by *Traductor*. The method for this is to use a special character `~` in front of the spaces you want to preserve at the beginning of the string (or after the spaces at the end). For instance:

```
a = ["  ~     Hello ~ World      ~ ",
     "|FR|  "   ~    Bonjour ~ monde     ~   "]
puts "<" + Traductor[a] + ">"
```

➔

```
<    Bonjour ~ monde     >
```
if the language is French

```
<     Hello ~ World    >
```
otherwise

Note that the delimiter character `~` is not itself replaced by a space.

Multi-line text

You can freely use the *newline* character `\n` if you have several lines.

For instance

```
a = "   line1\n   line2   \nline3   "
puts "<" + Traductor[a] + ">"
```

➔

```
<line1
   line2
line3>
```
in any language

Note however that *Traductor* will not eliminate the leading and trailing spaces in each line, except leading spaces in the first one and trailing spaces in the last one.

Quoted text with %q and %Q

You can also define your string by using the special quote construction %q (for single quoted strings) and %Q (for double quoted strings, recommended).

```
a = %Q{   line1\nline2 |FR|Ligne1\n ligne2}
```
is valid

Text in mode HERE

Ruby allows defining texts in their original form, by the construction <<HERE … HERE (and actually derived forms). It is supported as well by *Traductor*.

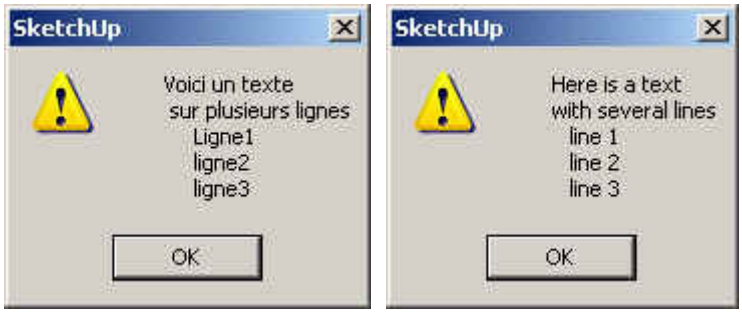
For instance,

```
a = <<HERE
~    Here is a text
   with several lines
         line 1
         line 2
         line 3
   |FR|
~     Voici un texte
      sur plusieurs lignes
           Ligne1
           ligne2
           ligne3
HERE
```

Then the following call….

```
Traductor.messagebox a
```

….will generate in French and in other languages:



6) Good practices for multi-language support

In the definition of multi-language strings, I would recommend **you always have a default string not prefixed with any language code**. For instance if you define your string as `Mystring = "|EN| Welcome |FR| Bienvenue"`, then, if the current language is |DE|, the translation would be `"|EN| Welcome |FR| Bienvenue"` (so not nice), whereas if you define it as `Mystring = "Welcome |FR| Bienvenue"`, you would get `"Welcome"`, whatever the non-French language is, including English.

Second, it is strongly advised to **put all your translatable strings defined as constants in the top section of your module**, so that they can be easily retrieved and possibly translated by someone else, without digging into your code.

If you have other strings constants that must not be translated, make it clear to the reader too.

I also stressed the benefits of **following some naming conventions**, whatever it is, as seen in the previous section.

You can have a look at my own practices in my macro ***beziersplline.rb***

2. Message Box

The *Traductor* macro provides `Traductor.messagebox`, which has an equivalent functionality as the `UI.messagebox` verb, but, as you could expect, with the support of multi-language strings.

The syntax is actually the same as the Sketchup `UI.messagebox` verb, except that you can pass multi-language strings for the `message` and `title` arguments. So it exists in 3 forms, depending on whether you pass or not the second and third arguments, with the same return value convention as for `UI.messagebox` (i.e. the index of the button pressed)

```
Button_index = Traductor.messagebox message                    #just OK button
Button_index = Traductor.messagebox message, nbtype
Button_index = Traductor.messagebox message, nbtype, title
```

…which is actually equivalent to:

```
Button_index = UI.messagebox Traductor[message], nbtype, Traductor[title]
```

A more advanced form is provided to encode variable arguments as %1, %9 in the message:

```
Button_index = Traductor.messagebox_arg message, nbtype, title, *args
```

…which is actually equivalent to:

```
Button_index = UI.messagebox Traductor[message,*args], nbtype, Traductor[title]
```

For instance:

```
a = 23
msg = "Value of %2 is %1 |FR| la valeur de %2 est %1"
Traductor.messagebox_arg msg, MB_OK, "Results|FR|Resultats", a, "a"
```

As a reminder, `nbtype` can take the following values, with indication of the button index:

- **MB_OK** – Contains an OK (1) button
- **MB_OKCANCEL** – Contains OK (1) and CANCEL (2)buttons
- **MB_ABORTRETRYCANCEL** – Contains ABORT (3), RETRY (4), and CANCEL (2) buttons
- **MB_YESNOCANCEL** – Contains YES (6), NO (7), and CANCEL (2) buttons
- **MB_YESNO** – Contains YES (6) and NO (7) buttons
- **MB_RETRYCANCEL** – Contains RETRY (4) and CANCEL (2) buttons
- **MB_MULTILINE** – Contains an OK (1) button. In a MB_MULTILINE message box, the message is displayed as a multi-line message with scrollbars (as needed).

Note that the title argument does not seem to be used by Sketchup in all cases. In my version, it works with the option MB_MULTILINE. For other options, the title is always "Sketchup".

3. Dialog Box

Initially my intent was to adapt Sketchup dialog boxes (verb `UI.inputbox`) to support multi-language strings. However, while coding, I realized I could do a little bit more, at least in 2 areas:

- Structuring the results in a way which is more independent of the dialog box itself
- Validating the output results

1) Overview – How to construct a dialog box

Before you can use a dialog box, you need to design its template, which can be done with the following steps:

Creating the dialog box object:

```
dlg = Traductor::DialogBox.new title
```

Declaring fields, either string, numeric or unit-based numeric, or enumeration list:

```
dlg.field_string symbol, label, default, validation_pattern
dlg.field_numeric symbol, label, default, value_min, value_max
dlg.field_enum symbol, label, default, enum_hash
```

Showing the dialog box:

```
Hash_results = dlg.show hash_initial_values
```

Compared to the traditional Sketchup `UI.Inputbox`, the *Traductor* macro provides a few enhancements:

- Default values and results are formatted as hash array, not as arrays. This means that you are not depending any longer on the order of fields as displayed. This may not appear as an immediate benefit, but it actually gives some independence between the way you prompt the user and the way you collect his inputs.
- Validation is integrated for simple checks, such as min /max values for integer, or pattern matching for strings.
- Enumeration list are based on hash arrays too, meaning that the default value and result is not the string that is shown to the user, which is indeed language dependent, but a symbolic value which is language independent.

Let's review this in more details.

2) Creating the dialog box object

The Traductor macro comes with a class `DialogBox`, which you can instantiate with the `new` method, taking one or more parameters:

```
dlg = Traductor::DialogBox.new title
dlg = Traductor::DialogBox.new title, validation_proc, context
```

In the second form, you can pass your own method `validation_proc` to validate the output of the user, which can be useful if, for instance, the value of some fields depends on the value of other fields. The additional argument `context` will be passed as an argument when calling the validation procedure and contains whatever contextual data you want.

3) Creating fields of the dialog boxes

When you have a Dialog Box object, you can insert fields. Each field is given a **unique symbolic name** (as a string), which will be the index of the initial values you pass when showing the dialog box, as well as the results when the user has pressed OK.

The label of the field is indeed a multi-language string and will appear in the current language along the rules we have seen in previous sections.

For convenience, I have defined 3 categories of fields: **String**, **Numeric** and **Enumeration**.

- ### String fields

  The syntax is

  ```
  dlg.field_string symbol, label, default,
                   valid_pattern=nil, msg_pattern=nil
  ```

  where

  - `Symbol` is the **unique symbolic name** of the field, as a **string**
  - `label` is the **label** appearing in front of the input field, as a **multi-language string**
  - `default` is the **default value** appearing in the field if nothing else is specified, as a **multi-language string**
  - `valid_pattern` is an ***optional*** **multi-language string containing a regular expression** that the input string entered by the user must respect. If nil, then no validation is performed. Note that you need to double the backslashes when you use special characters; ex: `"\\A\\w\\w\\Z"` to force only 2 characters (equivalent to `/\A\w\w\Z/`).
  - `msg_pattern` is an ***optional*** **multi-language string** indicating the **constraint on the string** (as a regular expression is not always that clear to a user!)

  For instance, the following declaration will generate a field named "Name", with a label "Nom:" in French, and "Name:" in other languages, and which must contain `'aa'` in French and `'oo'` in other languages.

  ```
  dlg.field_string "Name", "Name:|FR|Nom:",
                   "enter name |FR| Entrez un nom",
                   "oo |FR| aa",
                   "must contain 'oo'",
                   "|FR| doit contenir un 'aa'"]
  ```

- **Numeric fields**

  Numeric fields can be used to enter numbers, whether integer or floats or Sketchup distances or coordinates. The general syntax is:

  ```
  dlg.field_numeric symbol, label, default, vmin=nil, vmax=nil
  ```

  where:

  - `Symbol` is the **unique symbolic name** of the field, as a string
  - `label` is the **label** appearing in front of the input field, as a multi-language string
  - `default` is the **default numeric value** appearing in the field if nothing else is specified
  - `vmin` is the **lower bound limit** of the value. It is ***optional***. If omitted or nil, then no validation is performed
  - `vmax` is the **upper bound limit** of the value. It is ***optional***. If omitted or nil, then no validation is performed

  For instance, the following declaration will generate a field named *"NbPoints"*, with a label *"Nombre de points:"* in French, and *"Number of Points"* in other languages, and which must be comprised between 3 and 100. The default value is 5.

  ```
  dlg.field_numeric "NbPoints",
                    "Number of points:|FR| Nombre de points:",
                    5, 3, 100
  ```

  For floats, it works the same, with the Ruby rules to define float numbers.

  For Sketchup distance or coordinates, you just need to append the indication of units. In Sketchup, the internal value is always in inches, but you can specify your own. For instance:

  ```
  dlg.field_numeric "Len", "Length:|FR|Longueur", 4.cm, 3.cm, 100.cm
  ```

  The good thing is that Sketchup will automatically convert your specified unit into the current model units.

  Be careful to pass `vmin` and `vmax` in the same form as the default value. For convenience, the limits are indicated after the field label when specified.

- **Enumeration fields (also called combo boxes)**

  In the standard Sketchup `UI.Inputbox` verb, you pass directly the string enumeration as an argument, the string chosen by the user is returned in the results. Obviously, when you are in multi-language environment, this will not be convenient in order to branch your code based on the return value.

  If for instance you need the user to choose between Circle and Polygon, your enumeration parameter will be `"Circle|Polygon"` in English, but `"Cercle|Polygone"` in French. So your code would have to test the return value in both languages.

  I propose a slightly different approach by using Hash arrays instead, so that you can give an absolute code to each option, regardless of the string appearing to the user in the current language. In the example above, you would simply define the enumeration as:

  ```
  Enu_hash = { 'C' => "Circle |FR| Cercle",
               'P' => "Polygon |FR| Polygone" }
  ```

  so that what you have to test in the return value is just the code, independent of the language, here 'C' or 'P'.

  The syntax for adding an enumeration field to a dialog box is the following:

  ```
  dlg.field_enum symbol, label, default, enu_hash
  ```

  where

  `enu_hash` is the list of options encoded as a hash array, as seen above.

4) Passing values and getting results

Once the dialog box has been defined, which is normally done once, you can show it to the user, with filling it with some values, and then getting the resulting output.

The standard Sketchup `UI.Inputbox` verb uses Arrays for the input values and the output results, with the array index based on the order of fields. The *LibTraductor* macro uses **hash arrays** instead, where the keys are the symbols of the fields, regardless of their order in the dialog box. I found it more readable in the code, and more flexible for potential evolutions of your code.

So you invoke the dialog box with a call to the class method `show`, which return either `nil` if the user pressed Cancel, or the results as a hash array if the user pressed OK.

For convenience, I provide several forms of invocations, noting that `hash_values` is a hash array containing the input values i.e. those appearing as initial values in the dialog box):

- **Call with one argument**: `hash_results = dlg.show hash_values`

  In this case, `hash_results` is false if the user pressed Cancel, or contains the results if the user pressed OK.

- **Call with two arguments: status =** `dlg.show hash_values, hash_results`

  In this case, `hash_results` contains the results of the dialog box, whatever button was pressed, whereas `status` indicate whether the user pressed Cancel (`nil`) or contains the reference to the results.

- **Call with one argument, overriding inputs with outputs (note the exclamation mark!):**
  ```
  status = dlg.show! hash_values_and_results
  ```
  In this case, `hash_values_and_results` is used both to pass inputs and to receive outputs.

- **Call with NO argument:**
  ```
  status = dlg.show
  ```
  In this case, the initial values of the dialog box are those defined by default, and the dialog box will then always show the previous values validated (so a kind of auto-updating).

Note that, regardless of how the dialog box is called, **the results can always be obtained as**:

- The return value of the `show` method, when user pressed OK
- The attribute `hash_results` of the dialog box object, for instance
  ```
  myresults = dlg.hash_results
  ```

5) Modifying items of a dialog box

Actually the same methods `field_xxx` that were used to create items in a dialog box can serve to modify them. The convention is however slightly different: if an argument is passed `nil`, then the internal value is not altered. Indeed, you must be sure that the dialog item has first been created, otherwise you would create a new one.

A few examples:

- `dlg.field_numeric "NbPoints", nil, nil, 4, 12`
  will only modify the vmin (to 4) and vmax (to 12) of the numeric item with symbolic name *"NbPoints"* , without altering the label and the default

- `dlg.field_string "STR", "new label", nil`
  will only modify the label of the field

- `dlg.field_enum "EE", nil, 'T', new_enu_hash`
  will only modify the list of values and the default

It is possible to **retrieve the internal values of a dialog items**, by using the method `get_item_property`, passing the key and the property name, both as strings (for convenience, I made it case insensitive). For instance,

```
Value_min = dlg.get_item_property "NbPoints", 'vmin'
```

Finally, you can **modify the title and the validation method**:

- `dlg.set_title newtitle`
- `dlg.set_validation_proc newvalproc, newcontext`
  `dlg.set_validation_proc` (with no argument, remove the procedure)

6) Validation procedure

You have the choice between relying on

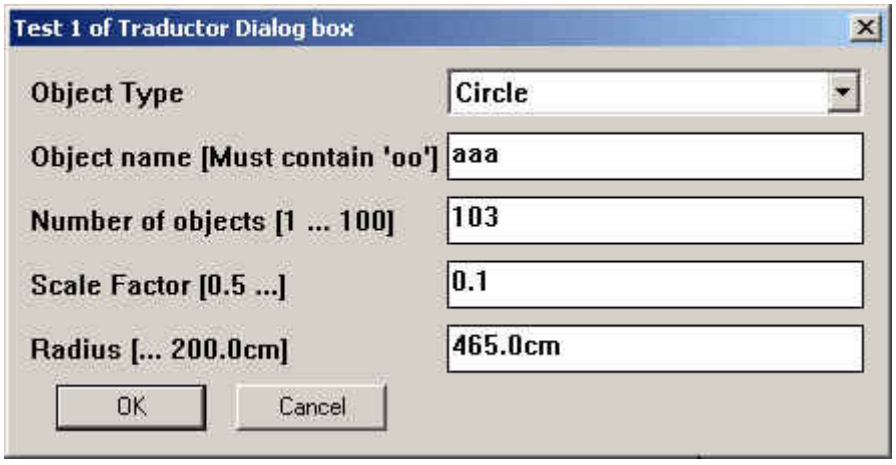
- Either the built-in validation mechanism provided by Traductor
- Or your own validation procedure that you specify when creating the dialog box

Note that there is anyway a validation provided by Sketchup itself, on which external programming has no control. This deals with validation of data type. For instance, if you enter a string in a field supposed to be numeric, you'll get a message like this one:

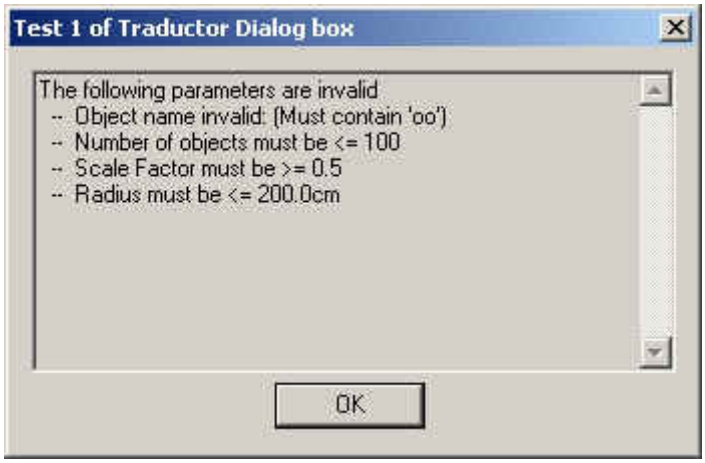
Built-in validation mechanism:

The principle is that when the user presses OK, the method will check all parameters entered by the user against their possible constraints (min, max for numeric, pattern for string) and display a single message box warning the user with problems. The user is then taken back to the dialog box, until he enters correct values or press Cancel.

For instance, if you entered some invalid values in the following dialog box….



You will get the following error message



Your own validation procedure

If you pass your own validation procedure, then the built-in validation is bypassed and instead all current values are passed to the validation procedure, which has the form:

```
status = my_own_validation_proc dlg, context=nil
```

The validation procedure can consult the results in `dlg.hash_results`.

It must return `true` if it considers the results are valid, or `false` otherwise. In the latter case, it is responsible to prompt the user for warning or additional actions.

Note that the validation procedure can alter the value of `dlg.hash_results`, either to reset some fields to correct values (if it returns `false`), or to change the results (if it returns `true`). It can also alter any property of the dialog items, as seen above.

For instance, your validation procedure can ask for confirmation on some value, and validate if the user does confirm.

As an example, let's assume you design your dialog box as:

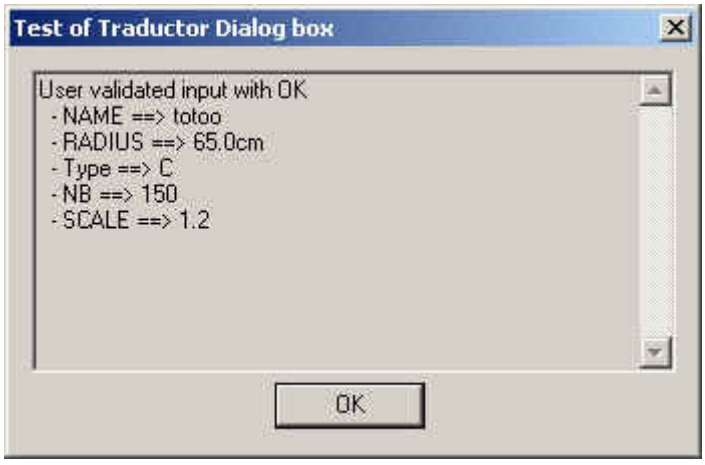
```
@dlg2 = Traductor::DialogBox.new STR_Title,
                                 'LibTraductor_Test.valid2proc', 150
@dlg2.field_enum "Type", STR_Type, 'C', STR_EnumType
@dlg2.field_string "NAME", STR_Name, "totoo", STR_Pattern
@dlg2.field_numeric "NB", STR_Number, 3
@dlg2.field_numeric "SCALE", STR_ScaleFactor, 1.2, 0.5, nil
@dlg2.field_numeric "RADIUS", STR_Radius, 65.cm, nil, 200.cm
```

With the validation procedure as:

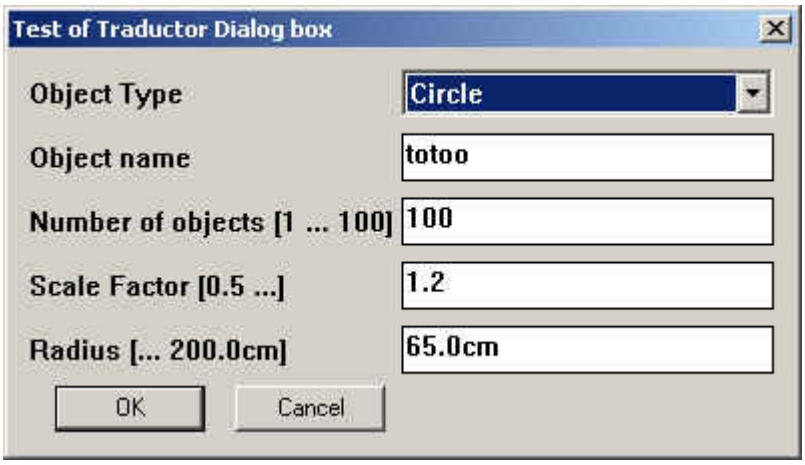
```
def LibTraductor_Test.valid2proc dlg, context=nil
  val = dlg.hash_results['NB']
  if (val > context)
    status = UI.messagebox (Traductor.s(MSG_CONFIRM2,context), MB_YESNO)
    if (status == 6)
      dlg.hash_results['NB'] = context          #cap value to context
      return true
    else
      dlg.hash_results['NB'] = 100              #reset value to maximum 100
      return false
    end
  end
  true
end
```

When it displays and if you enter the value 200 in the Field 'NB' (Number of objects), then your validation procedure will prompt the user for confirmation:

If you press 'YES', then the validation will be OK, but the returned value will be capped at 150:



If you press 'NO', you will take the user back to the dialog box, but with the value now set to 100 (the values of the other field being unchanged).



This is indeed trickier to program than to rely on the built-in validation procedure, but this can give you some flexibility.

Note that an alternative method is simply to rely first on whatever you can check with the built-in validation procedure, and then, when the user press OK, do your own validation and possibly redisplay the dialog box if there is still a problem.

7) An example of using dialog boxes

I put hereunder a full example of how to code a dialog box illustrating each type of fields. This is taken from the test script ***LibTraductor_test.rb*** which goes along with the macro.

As a first step, you **define the constants** in the header of your module

```
STR_Title = ["Test 1 of Traductor Dialog box",
             "|FR| Premier test des boites de dialogue Traductor"]
STR_Type = ["Object Type", "|FR| Definition de l'objet"]
STR_EnumType = { 'R' => "Rectangle |FR| Rectangle",
                 'T' => "Triangle |FR| Triangle",
                        'C' => "Circle |FR| Cercle" }
STR_Number = ["Number of objects", "|FR| Nombre d'objets"]
STR_ScaleFactor = ["Scale Factor", "|FR| Facteur d'echelle"]
STR_Radius = ["Radius", "|FR| Rayon"]
STR_Name = ["Object name", "|FR| Nom de l'objet"]
STR_Pattern = "/o/ |FR| /a/"
MSG_CANCEL = "User pressed cancel |FR|Sortie de dialogue avec ANNULER"
MSG_OK = "User validated input with OK |FR| Validation des entrees avec OK"
```

Then you **design your dialog box**

```
unless @dlg
   @dlg = Traductor::DialogBox.new STR_Title
   @dlg.field_string "NAME", STR_Name, "toto"
   @dlg.field_enum ("Type", STR_Type, 'C', STR_EnumType)
   @dlg.field_numeric ("NB", STR_Number, 3, 1, 100)
   @dlg.field_numeric ("SCALE", STR_ScaleFactor, 1.2, 0.5, nil)
   @dlg.field_numeric ("RADIUS", STR_Radius, 65.cm, nil, 200.cm)
end
```

Finally you **invoke the dialog box**

- A first method is to rely on defaults, and just call the method **show** with no argument. Then the next time you will call it, it will show the values entered by the user the previous time.

```
if (@dlg.show)
   puts "User validated with OK --> #{@dlg.hash_results}"
else
   puts "User pressed cancel"
end
```

- A second method is to keep a separate control over the input and the output. You defined the input in a hash array, with all or a subset of the values (the rest will be taken as default):

```
   hash_values = {'Type' => 'R'}
   if (@dlg.show hash_values)
      puts "User validated with OK --> #{@dlg.hash_results}"
   else
      puts "User pressed cancel"
   end
```

The variable `hash_values` is not touched, so that the next time you will call the dialog box with the same call `@dlg.show hash_values`, it will show the initial values, not the latest validated

- A third method is to use the same variable, here `hash_array`, to pass the input and get the output:

```
hash_array = {'Type' => 'R'}
if (@dlg.show! hash_array)
    puts "User validated with OK --> #{@hash_array}"
else
    puts "User pressed cancel"
end
```

Note that if the variable `hash_array` contains other keys that those used by the dialog box, they will be preserved. This is a way to carry additional parameters in the same variable.

More generally, you can play with the test macro ***LibTraductor_test.rb*** to check the different behaviors of the dialog box interface.

4. Hash Arrays

Hash Arrays are a very convenient way to represent parameters with an approach [key, value]. We have seen the benefits about Dialog boxes. However, you cannot directly store a hash array as an attribute of a Sketchup entity via the Entity.set_attribute verb, because it is a structured, not a flat string.

LibTraductor provides 3 methods for helping this area[2]:

- `hash_marshal`: Encode a hash array into a character string (i.e. marshaling)
- `hash_unmarshal`: Decode a string into a hash array
- `hash_pretty`: produce a nice string which can be used to display your hash array

1) Marshaling Hash Arrays

```
str = Traductor.hash_marshal (hash_array)
```

The encoding is based on a double line feed '`\t\t`' between each pair (key, value) and then a double tab i.e. '`\t\t`' between the key and the value. In addition, types are identified after the value by a triple tilde '`~~~`' followed by the type of the variable, either Integer, Float or Length. So be careful not to have a double line feed, a double tab or a triple tilde in the keys and values of your hash array. If `hash_array` is nil, the method returns the empty string "".

For instance if `hsh = {"a" => 23.0cm, "b" => 43.5, "c" => "hello"}` then,

```
Traductor.hash_marshal hsh
→ "a\t\t9.05511811023622~~~ \n\nb\t\t43.5~~~f\n\nc\t\thello"
```

Note that the distance `23.0cm` has been encoded as inches, `9.05511811023622`.

2) Un-marshaling Hash Arrays

```
hsh = Traductor.hash_unmarshal (str)
```

This of course works if the string has been previously marshaled by *Traductor*. Note that the Sketchup units are preserved in the marshal and un-marshal operations. If `str` is nil, the method returns an empty hash array {}.

3) Pretty Print of Hash Arrays

```
str = Traductor.hash_pretty (hash_array, leadstr=nil)
```

This method prints the pairs (key, value) one by line, with an optional lead string. For instance, with the previous hash array.

`Traductor.hash_pretty hsh, "  --  "` will generate

```
--  a => 23.0cm
--  b => 43.5
--  c => hello
```

[2] The Ruby methods `Marshal::dump` and `Marshal::load` do not work either with Sketchup entity attributes as the string is encoded in binary form.

4) Usage with Sketchup entities

This main benefit of the Traductor marshaling is to be able to store hash arrays as attributes of Sketchup entities.

For instance…

```
entity.set_attribute "dicoskp", "param", Traductor.hash_marshal hsh
```

….and then

```
hsh = Traductor.hash_unmarshal (entity.get_attribute "dicoskp", "param")
```

5. Tracing Code

You may want to log some information to the Ruby console, for instance in case of programming error, or just as a permanent trace.

In order to help you make a difference between the method puts, and also to support multi-language, *Traductor* includes 2 methods, which are actually doing the same thing but have different names for readability of your code:

```
def Traductor.log_error (msg, *args)
   puts Traductor[msg, *args]
end

def Traductor.log_info (msg, *args)
   puts Traductor[msg, *args]
end
```

These two methods supports token replacement (i.e. %1, …, %9)