

---

## LA TOUR DE BABEL : TRADUIRE UNE APPLICATION

---

**Objectifs** : dans ce chapitre, vous apprendrez à traduire un projet dans une autre langue. Vous découvrirez aussi que la première langue étrangère pour **Lazarus** est... le français.

**Sommaire** : Un programme français... en anglais – de l'anglais au français – encore plus loin : de l'anglais au choix de la langue

**Ressources** : les programmes de test sont présents dans le sous-répertoire *traduction* du répertoire *exemples*.

---

### WHAT'S THE MATTER ?<sup>1</sup>

---

Le programmeur qui souhaite adapter un logiciel à une autre langue imagine peut-être qu'il suffit de traduire les termes à afficher et de présenter cette traduction à l'utilisateur final. Pour lui, la tâche paraît triviale. Et pourtant...

Ce programmeur naïf aura certes prévu une série de messages et pensé à les regrouper dans une unité particulière afin d'éviter de se perdre dans le code source, mais il aura aussi complété des propriétés depuis l'inspecteur d'objet (*Hint* et *Caption* par exemple), fait appel à des unités tierces qui elles-mêmes renvoient des messages (ne serait-ce que ceux de la LCL) et prévu la récupération de données depuis des fichiers ou un clavier. Les chaînes de caractères affichées sont en effet d'origines diverses : elles peuvent aussi bien provenir du code du programme, des fiches créées, des unités utilisées que de conditions extérieures. Dans ce contexte, comment se mettre à l'abri d'oublis, d'erreurs ou d'incohérences ?

De plus, les langues n'entretiennent pas des relations bijectives : les caractères employés, la ponctuation, la syntaxe, les accords (le genre et le nombre), l'emploi des modes et des temps, les habitudes de formulation, même la signification des couleurs sont quelques-uns des aspects qui révèlent qu'une langue renvoie à un système complexe attaché à une culture particulière.

On pourrait ainsi multiplier les exemples de complications :

- l'anglais est une langue compacte si on la compare aux autres langues : il faut en tenir compte pour la largeur des légendes des composants utilisés ;
- les langues à idéogrammes ignorent les abréviations ;
- les majuscules ont un sens particulier en allemand ;

---

<sup>1</sup> Quel est le problème ?

- la notion de pluriel est dépendante de la langue utilisée (Anglais: “*If the length of S is already equal to N, then no characters are added.*” – Français : « *Si la longueur de S est déjà égale à N, aucun caractère n’est ajouté.* ») ;
- le mois d’une date en anglais est donné avant le jour, contrairement au français ;
- ...

Dans le cadre de ce chapitre, afin de ne pas lui donner une ampleur démesurée, ne sera abordée que la traduction du point de vue du programmeur : comment faire pour qu’un logiciel s’adapte au mieux à une autre langue ? Mais il faudra que vous gardiez à l’esprit ce qui précède avant de vous lancer dans l’internationalisation d’un travail !

Si vous êtes tenté d’ignorer ce chapitre en pensant limiter votre production à la langue française, vous êtes invité à en lire au moins la première partie. En effet, **Lazarus** et Free Pascal sont des outils conçus en anglais pour un public anglophone. Les difficultés commencent dès lors qu’un projet envisage d’utiliser une autre langue que la langue de Shakespeare.

---

## UN PROGRAMME FRANÇAIS... EN ANGLAIS

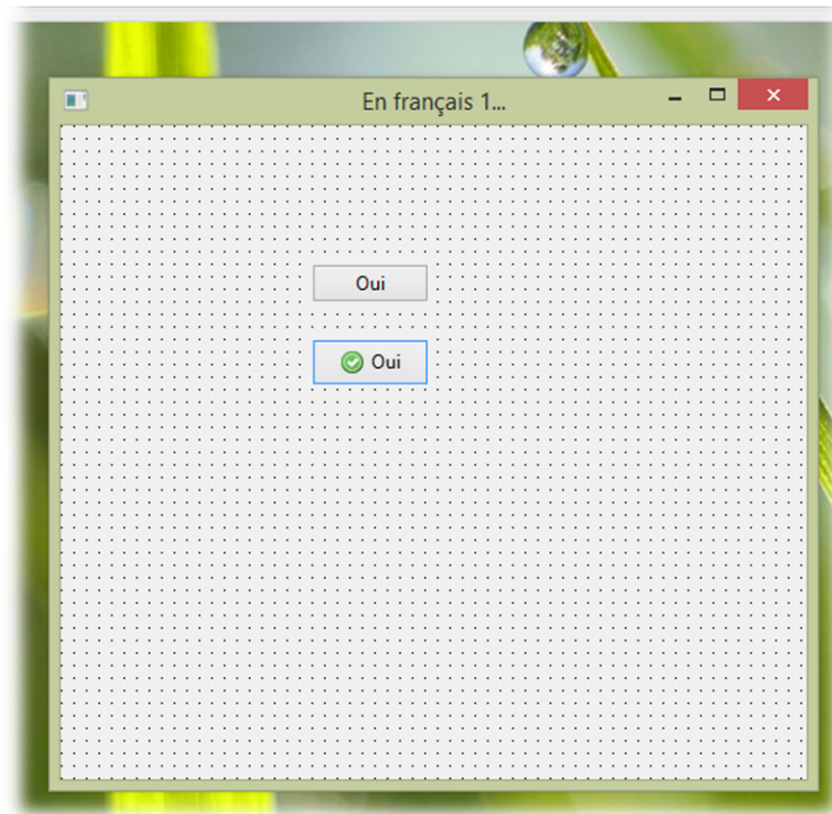
---

### [Exemple TR\_01]

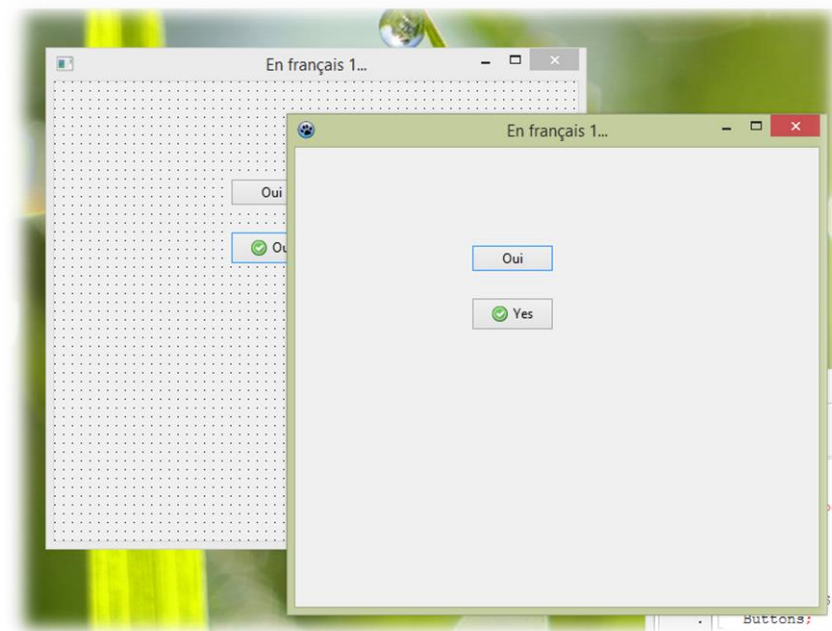
Pour vous en persuader, examinez le comportement d’un programme aussi élémentaire que celui-ci :

- créez un nouveau projet ;
- modifiez la légende de la fiche (*Caption*) en la faisant passer de « *Form1* » à « *En français 1...* » ;
- déposez un bouton *TButton* sur la fiche proposée ;
- modifiez la légende de ce bouton (*Caption*) en la faisant passer de « *Button1* » à « *Bouton* » ;
- déposez un bouton avec glyphe *TBitBtn* sur la même fiche ;
- modifiez sa propriété de type (*Kind*) en la faisant passer de « *bkCustom* » à « *bkYes* ».

Voici l'aspect, à la conception, de votre préparation :



Compilez à présent votre application et lancez son exécution. Voici ce que vous obtiendrez :



Vous serez sans doute tenté de croire que la transformation du « *Oui* » en « *Yes* » pour le composant **TBitBtn** est un bogue de **Lazarus** puisque le composant **TButton** ne semble pas souffrir de la même tare. Cependant, avant de vous ruer sur la rubrique *Bugtracker* du site de **Lazarus**, il est de nouveau conseillé de lire la suite. Ce comportement apparemment aberrant s'explique si l'on comprend comment fonctionne le système de traduction.

---

## UN PEU DE BRICOLAGE POUR TRADUIRE

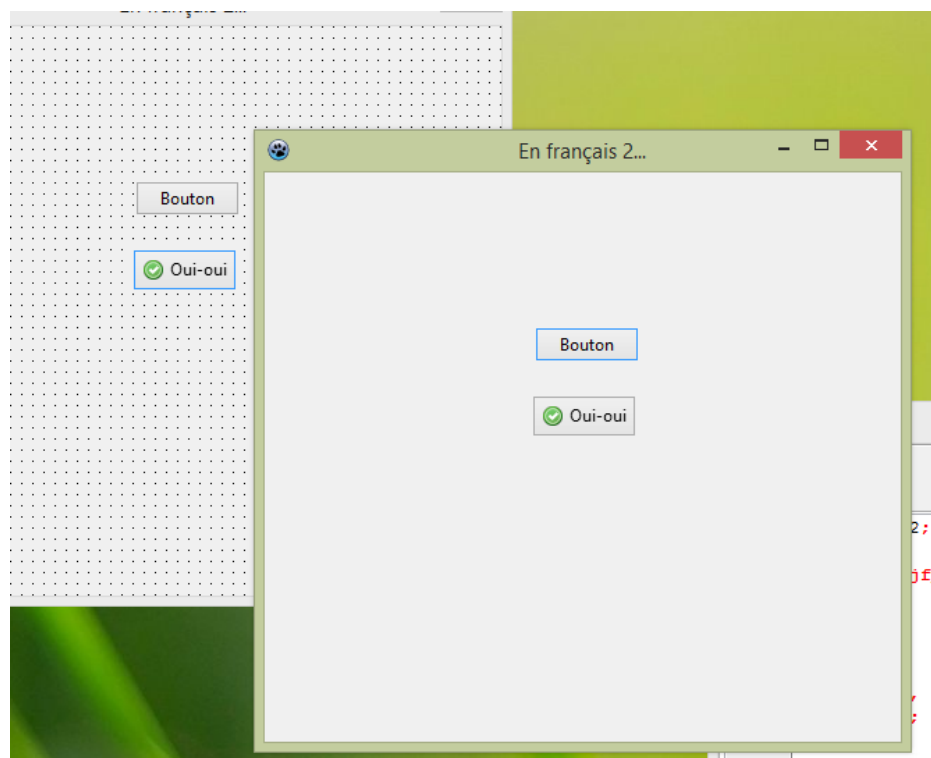
---

### [Exemple TR\_02]

Une première manière de contourner le problème rencontré serait de modifier manuellement la valeur de la propriété en cause, ici **Caption**. En effet, si vous changez cette valeur depuis l'inspecteur d'objet, le comportement correspondra à celui qui était attendu :

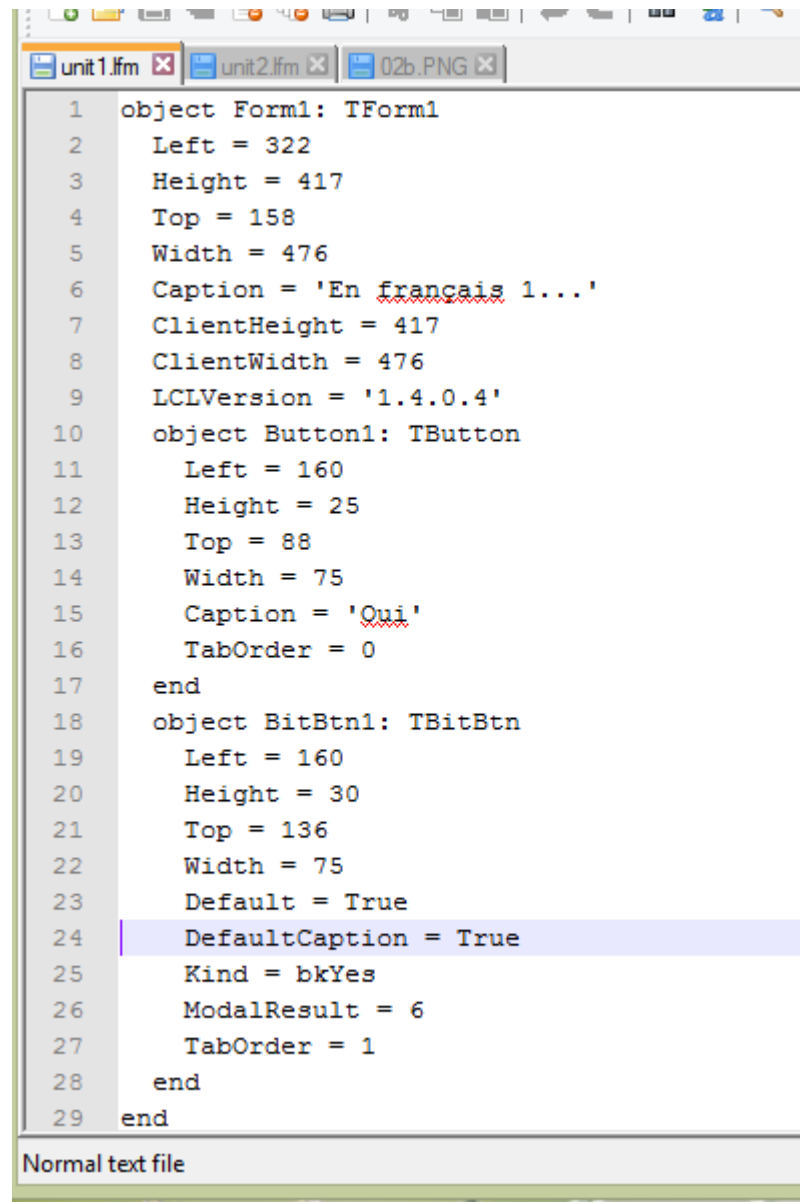
- modifiez la valeur de la légende (**Caption**) en la passant de « *&Oui* » à « *&Oui-oui* » ;
- compilez le programme ;
- lancez son exécution.

Vous obtiendrez cet écran :



Que s'est-il passé ? Pour le comprendre, il faut examiner les fichiers LFM qui contiennent la description des fiches. Comme ce sont de simples fichiers textes, des outils tels que **Notepad++** pour Windows ou **gEdit** pour Linux sont tout à fait adaptés.

Dans la première version du programme, on lit ceci :

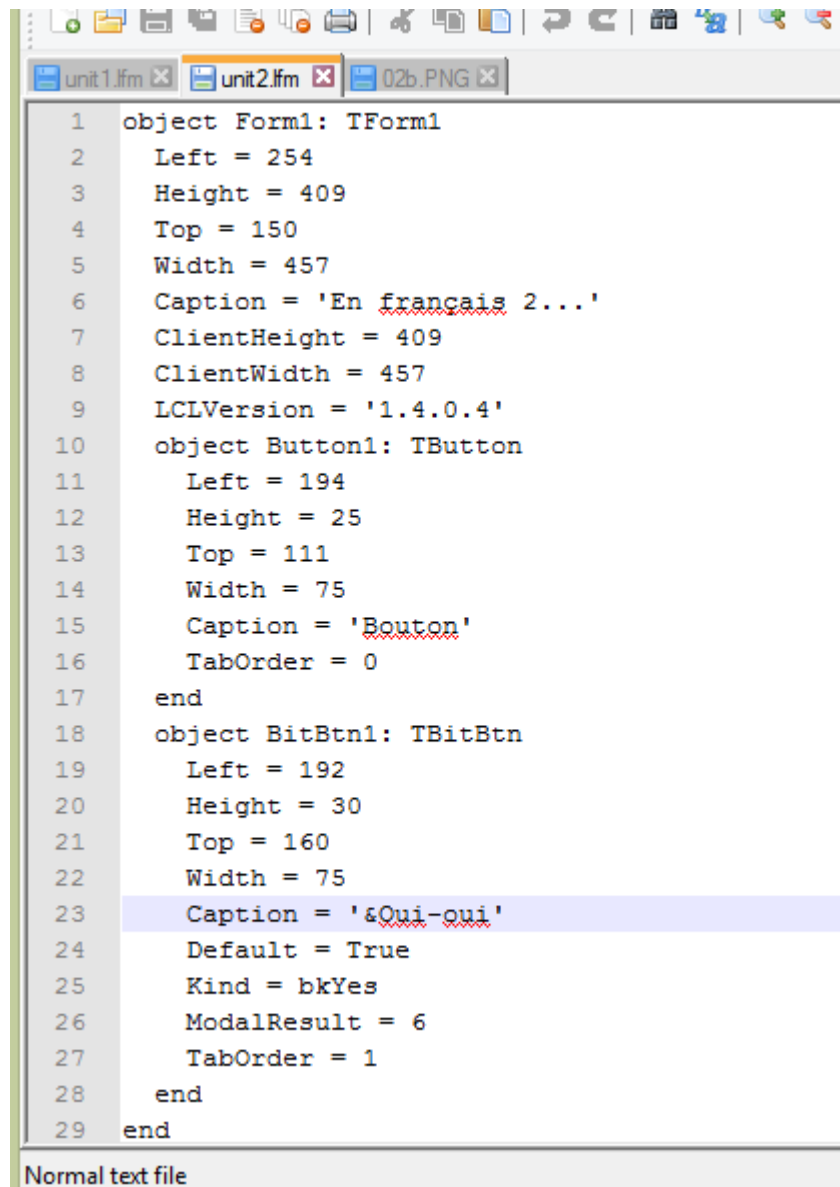


```
1 object Form1: TForm1
2   Left = 322
3   Height = 417
4   Top = 158
5   Width = 476
6   Caption = 'En français 1...'
7   ClientHeight = 417
8   ClientWidth = 476
9   LCLVersion = '1.4.0.4'
10  object Button1: TButton
11    Left = 160
12    Height = 25
13    Top = 88
14    Width = 75
15    Caption = 'Oui'
16    TabOrder = 0
17  end
18  object BitBtn1: TBitBtn
19    Left = 160
20    Height = 30
21    Top = 136
22    Width = 75
23    Default = True
24    DefaultCaption = True
25    Kind = bkYes
26    ModalResult = 6
27    TabOrder = 1
28  end
29 end
```

Normal text file

On voit que le *BitBtn* affiche la légende par défaut : c'est ce qu'indique la ligne « *DefaultCaption = True* ».

L'affichage de la version modifiée donne ceci :



```
1 object Form1: TForm1
2   Left = 254
3   Height = 409
4   Top = 150
5   Width = 457
6   Caption = 'En français 2...'
7   ClientHeight = 409
8   ClientWidth = 457
9   LCLVersion = '1.4.0.4'
10  object Button1: TButton
11    Left = 194
12    Height = 25
13    Top = 111
14    Width = 75
15    Caption = 'Bouton'
16    TabOrder = 0
17  end
18  object BitBtn1: TBitBtn
19    Left = 192
20    Height = 30
21    Top = 160
22    Width = 75
23    Caption = '&Oui-oui'
24    Default = True
25    Kind = bkYes
26    ModalResult = 6
27    TabOrder = 1
28  end
29 end
```

Normal text file

Cette fois-ci, la ligne relevée a disparu, mais une autre ligne a fait son apparition : « *Caption = '&Oui-oui'* ». C'est elle qui assure que le message sera bien traduit à l'exécution.



Un problème secondaire surgit avec cette solution : la chaîne par défaut est finalement la seule qui ne sera jamais affichée ! Dès que vous la proposez, elle est ôtée du fichier LFM.

Mais revenons à notre question initiale : que s'est-il passé ? Afin d'éviter d'encombrer le fichier LFM de données inutiles, *l'EDI n'enregistre que les valeurs des propriétés qui diffèrent de leur valeur par défaut.*

En modifiant le libellé manuellement, vous avez forcé **Lazarus** à stocker la nouvelle valeur dans le fichier LFM qui accompagne la fiche en cause. De même, en inversant la valeur de la propriété *DefaultCaption*, vous avez forcé l'affichage de la propriété *Caption* telle qu'elle apparaît dans l'inspecteur d'objet et non telle qu'elle est enregistrée par défaut dans la LCL. Autrement dit, si vous souhaitez qu'une propriété ait une valeur différente de celle par défaut, assurez-vous que le fichier LFM l'ait correctement enregistrée.



Souvenez-vous surtout que les valeurs par défaut sont celles définies au sein des unités employées, en particulier de la LCL. Ces valeurs sont essentiellement définies dans le constructeur *Create* des classes, en accord avec l'interface qui emploie le mot réservé *default* suivi de la valeur par défaut s'il s'agit de propriétés aux valeurs discrètes.

En fait, avec le composant *TBitBtn*, on aurait obtenu le même affichage en changeant la valeur de *DefaultCaption* de *True* à *False*. Cette seconde solution serait idéale si elle était indépendante du composant utilisé, mais cette propriété n'est présente que pour les descendants de *TCustomBitBtn* !

---

### UNE SOLUTION PLUS GENERALE

---

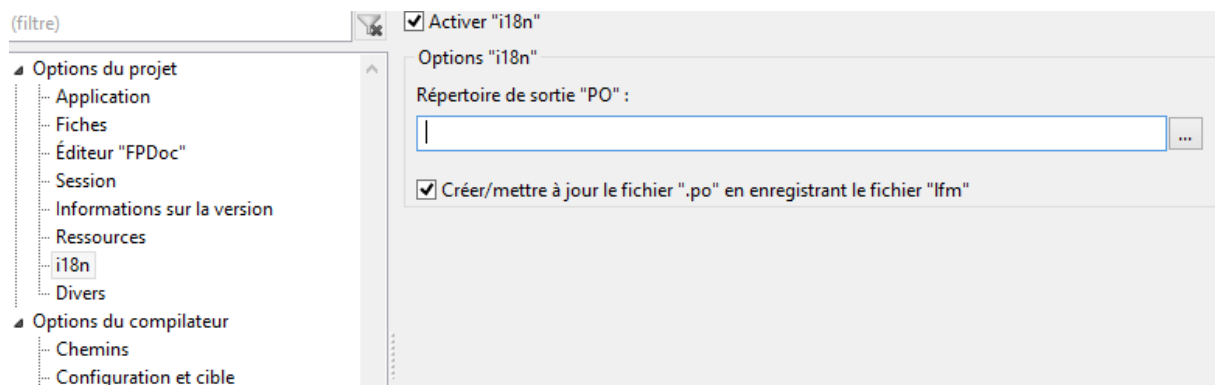
Vous pourriez vous satisfaire des deux premières solutions pour les propriétés accessibles en écriture. Malheureusement, de nombreux messages ne sont pas de ce type : certaines propriétés (comme le nom des couleurs) et la plupart des messages d'erreur sont hors de portée des unités créées.

**Lazarus** vient alors à votre rescousse en intégrant un système de traduction complet et automatique.

#### [Exemple TR\_03]

Pour illustrer le mécanisme mis en œuvre, procédez comme suit :

- créez un nouveau projet ;
- dans « *Projet* » → « *Options du projet* » → « *i18n* », cochez « *i18n* » et « *Créer/mettre à jour le fichier « .po » en enregistrant le fichier « .lfm »* » ;



- modifiez la légende de la fiche (*Caption*) en la faisant passer de « *Form1* » à « *En français 3...* » ;

- déposez un bouton avec glyphe (*TBitBtn*) sur la même fiche ;
- modifiez sa propriété de type (*Kind*) en la faisant passer de « *bkCustom* » à « *bkYes* » ;
- enregistrez ce projet dans le répertoire de votre choix sous le nom *TestTranlate03.lpr* ;
- ouvrez depuis le navigateur le répertoire utilisé ;
- créez un sous-répertoire que vous baptiserez *languages* (ce nom a son importance !) ;
- déplacez le fichier *TestTranslate03.po* apparu dans le dossier du projet dans le répertoire *languages* (si ce fichier est introuvable, déplacez légèrement la fiche principale et enregistrez de nouveau le projet) ;
- renommez le fichier *TestTranslate03.po* en *TestTranslate03.fr.po* ;
- copiez le fichier *lclstrconsts.fr.po* depuis son répertoire d'origine<sup>2</sup> jusqu'au répertoire *languages* que vous venez de créer ;
- éditez le fichier du projet *TestTranslate03.lpr* grâce à l'inspecteur de projet (il apparaît lorsqu'on choisit « *Projet* » → « *Inspecteur de projet* » dans le menu principal de l'EDI) ;
- ajoutez l'unité *DefaultTranslator* à la clause *uses* du programme :

```
program project3;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, main,
  { you can add units after this }
  DefaultTranslator; // ← unité ajoutée
```

- compilez et exécutez le programme.

Cette fois-ci, sans avoir rien modifié des propriétés à la conception, le programme traduit correctement la légende du bouton. Vous êtes toutefois en droit de vous dire que les moyens mis en œuvre sont disproportionnés par rapport aux résultats ! Pour vous rassurer, nous allons ci-après expliquer l'intérêt de l'ensemble puis son fonctionnement.

---

### INTERET DE NE PAS BRICOLER LA TRADUCTION

---

Vous avez à présent trois solutions à votre disposition :

- la modification manuelle de certains messages ;
- la modification de certaines propriétés elles-mêmes susceptibles de modifier l'affichage ;
- l'utilisation du système automatique intégré de **Lazarus** via l'unité *DefaultTranslator*.

---

<sup>2</sup> L'emplacement de *lclstrconsts.fr.po* est le sous-répertoire *lcl/languages* du répertoire d'installation de **Lazarus**.



Si les deux premières sont légères, la dernière est de loin celle recommandée, car elle fonctionne automatiquement pour tous les messages des unités du projet. Elle évite par conséquent de parcourir les unités et les fichiers LFM à la recherche de chaînes à traduire, avec le risque d'en oublier ! Enfin, elle est la seule à pouvoir traiter les messages inaccessibles depuis l'EDI.

#### [Exemple TR\_04]

En guise de démonstration, voici un nouveau programme très simple :

- créez un nouveau projet ;
- dans « *Projet* » → « *Options du projet* » → « *i18n* », cochez « *i18n* » et « *Créer/mettre à jour le fichier « .po » en enregistrant le fichier « .lfm »* » ;
- modifiez la légende de la fiche (*Caption*) en la faisant passer de « *Form1* » à « *En français 4...* » ;
- déposez un composant *TColorListBox* sur la fiche principale ;
- déposez un composant *TButton* sur la même fiche ;
- modifiez la légende du bouton (*Caption*) en la faisant passer de « *Button1* » à « *Joli !* » ;
- créez un événement *OnClick* pour le bouton et entrez le code suivant dans la partie *implementation* de la fiche :

```
resourcestring
// chaînes de ressources pour leur future traduction
RS_Pretty = 'Joli !';
RS_NotPretty = 'Pas joli !';

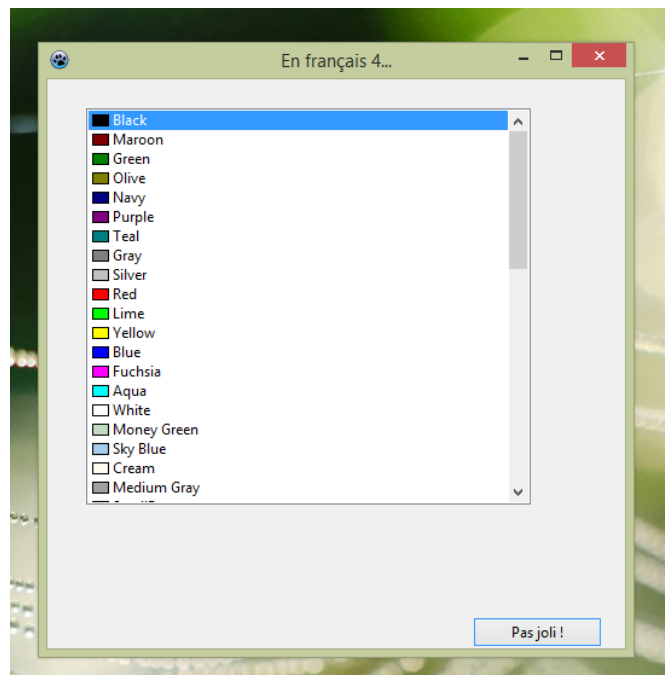
{$R *.lfm}

{ TForm1 }

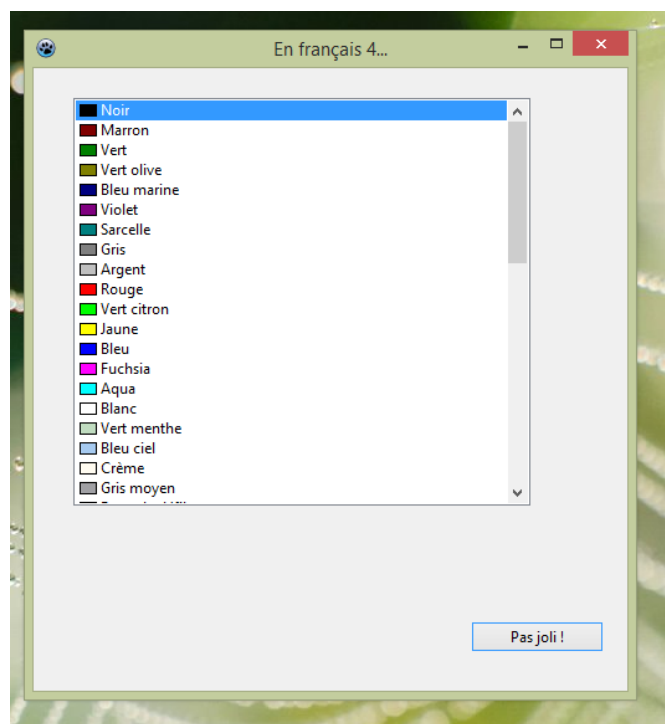
procedure TForm1.Button1Click(Sender: TObject);
begin
  if cbPrettyNames in ColorListBox1.Style then
    begin
      // propriété exclue
      ColorListBox1.Style := ColorListBox1.Style - [cbPrettyNames];
      Button1.Caption := RS_Pretty;
    end
  else
    begin
      // propriété incluse
      ColorListBox1.Style := ColorListBox1.Style + [cbPrettyNames];
      Button1.Caption := RS_NotPretty;
    end;
end;
```

La procédure introduite permet de modifier l'affichage du bouton en fonction de la propriété *Style* de la *TColorListBox*. L'option qui alterne est *cbPrettyNames* : si elle est incluse dans le style, le composant fait appel à la LCL pour afficher le nom en clair des couleurs et non leur codage interne.

À cette étape, si vous lancez l'exécution du programme et que vous cliquez sur le bouton, vous obtiendrez des noms en anglais :



En ajoutant la même unité *DefaultTranslator* à la clause *uses* du programme principal et les fichiers PO<sup>3</sup> dans un sous-répertoire *languages* du répertoire de l'application, les noms de couleurs seront traduits :



<sup>3</sup> N'oubliez pas de renommer *TestTranslate04.po* en *TestTranslate04.fr.po* et de copier le fichier *lclstrconsts.fr.po* dans ce répertoire si vous voulez que la LCL soit traduite !

En revanche, le codage des couleurs n'est pas affecté par la traduction : par exemple, *clBlue* affiche « *Blue* » en anglais et « *Bleu* » en français. Si vous cliquez de nouveau sur le bouton, les codes seront affichés tout simplement. Ce fonctionnement est bien celui désiré : pour l'utilisateur final, seul le nom des couleurs importe ; pour le programmeur, c'est celui des codes associés à ces couleurs.



Ce qu'il faut retenir de cet exemple très simple, c'est que des propriétés inaccessibles directement depuis l'EDI, comme ici le nom des couleurs, peuvent être traduites grâce à un mécanisme automatique.

### [Exemple TR\_05]

Par la même occasion, les messages d'exception le sont aussi. Pour vous en assurer, dans le même projet, complétez le code de la procédure *Button1Click* ainsi :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I, J: Integer;
begin
  I := 2;
  J := I - I;
  Button1.Caption:= IntToStr(I div J); // oups...
  // le reste ne change pas...
  if cbPrettyNames in ColorListBox1.Style then
  {...}
```

Lors du clic sur le bouton, une exception va être levée, car vous tenterez de diviser un nombre par 0. Avec l'unité *DefaultTranslator*, le message sera affiché en français. Si vous retirez l'unité de la clause *uses* du programme principal, le message sera en anglais. La solution adoptée pour la traduction est par conséquent très puissante : tout ce qui est du ressort de la LCL est traduit !

---

## FONCTIONNEMENT DE LA SOLUTION GENERALE

---

Comme le monde de l'informatique est étranger à la magie, l'apparent miracle de la traduction du texte a évidemment une explication rationnelle.

Le fait d'activer l'option « *i18n* » d'un projet indique au compilateur **Free Pascal** qu'il va devoir s'occuper de l'internationalisation du projet. « *i18n* » n'est qu'une abréviation d'*internationalization* : le « *i* » du début, les 18 lettres du mot et le « *n* » de la fin. En cochant la création et la mise à jour de fichiers PO à l'enregistrement des fichiers LFM, vous forcez **Lazarus** à produire des fichiers de ressources particuliers LRT pour chaque fiche lors de son enregistrement. Au cours de la compilation, **Lazarus** va rassembler ces fichiers de ressources en un seul fichier qui portera le nom du projet avec le suffixe PO. Ce fichier final contiendra toutes les chaînes à traduire définies par le projet. Nous détaillerons son contenu quand nous aborderons les traductions multilingues.

L'étape suivante consiste à inclure *DefaultTranslator* dans la clause *uses* du programme principal. Cette unité est rudimentaire, car elle se contente d'utiliser une autre unité (*LCLTranslator*) et d'exécuter dans sa section *initialization* une simple ligne :

```
SetDefaultLang("", "", false);
```

Cette procédure travaille pour l'essentiel ainsi :

- elle recherche un éventuel fichier PO portant le nom du projet adapté à la langue du système (pour nous, le français) : *projet4.fr.po* ;
- en cas de réussite, elle convertit les chaînes qu'il contient ;
- en cas de nouveau succès, elle recherche la version adaptée du fichier *lclstrconsts.po* (dans notre cas *lclstrconsts.fr.po*) pour convertir toutes ses chaînes.

Le premier travail s'effectue grâce à la fonction *FindLocaleFileName* de l'unité *LCLTranslator*.



Cette fonction cherche le fichier PO adapté à partir d'une série de répertoires standards et dans cet ordre : *languages* (celui que nous avons utilisé), *locale*, *locale\LC\_Messages* (ou *locale/LC\_Messages* pour les systèmes Unix) et */usr/share/locale/* (systèmes Unix seulement).

La recherche s'effectue à partir de deux infixes : pour le français, il s'agit de « *fr* » et de « *fr\_FR* ». La seconde version est dite étendue et la première réduite. Il s'agit de nuances et de particularités entre des dialectes suivant le pays où est parlée la langue. Ainsi, le français peut-il être celui de France, mais aussi celui du Québec, de Belgique, du Bénin, du Burundi... La version réduite est traitée prioritairement.

La conversion des chaînes est effectuée grâce à la fonction *TranslateResourceStrings* dont le rôle est de balayer toutes les chaînes d'origine afin de les transformer selon le contenu du fichier PO.

Ce n'est qu'après un traitement réussi que la LCL est traduite elle aussi par la même fonction *TranslateResourceStrings*. Voilà pourquoi nous avons besoin de créer un fichier PO propre à notre fiche pour obtenir une traduction correcte de la chaîne « *&Yes* » qui est définie et utilisée par la LCL.

---

## UNE QUATRIEME SOLUTION

---

### [Exemple TR\_06]

Il ressort de cette analyse qu'il existe une quatrième façon de traiter notre problème : en forçant la traduction de la LCL grâce à une portion de code, nous n'aurons plus besoin de créer un fichier PO supplémentaire.

En revanche, le code en sera un peu alourdi : il faudra modifier le corps du programme en contraignant ce dernier à une traduction explicite à partir du fichier *lclstrconsts.fr.po*, le tout en exploitant deux nouvelles unités (*gettext* et *translations*).

```

program TestTranslate06;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, main,
  { you can add units after this }
  sysutils, // une unité ajoutée pour PathDelim
  gettext, translations; // deux unités ajoutées

{$R *.res}

procedure LCLTranslate;
var
  PODirectory, Lang, FallbackLang: String;
begin
  Lang := ''; // langue d'origine
  FallbackLang := ''; // langue d'origine étendue
  PODirectory := '.' + PathDelim + 'languages' + PathDelim; // répertoire de travail
  GetLanguageIDs(Lang, FallbackLang); // récupération des descriptifs de la langue
  TranslateUnitResourceStrings('LCLStrConsts',
    PODirectory + 'lclstrconsts.fr.po', Lang, FallbackLang); // traduction
end;

begin
  RequireDerivedFormResource := True;
  LCLTranslate; // on ordonne la traduction
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

```



On notera que le délimiteur pour les chemins d'accès aux fichiers est traité grâce à la constante *PathDelim* définie en fonction du système d'exploitation en cours par *sysutils*. En évitant de coder ce délimiteur en dur, on étend la portabilité du code.

Avec cette méthode, il est inutile d'activer l'option « i18n ». Le répertoire du fichier PO est fourni par la procédure *LCLTranslate* à partir de la variable *PODirectory*. En revanche, seule la LCL est traduite par ce biais : la traduction d'autres unités exige de compléter le code ou de revenir à la traduction *via* « i18n » qui est au bout du compte bien plus simple à mettre en œuvre.

## DE L'ANGLAIS AU FRANÇAIS

## PREPARATION DU PROGRAMME SOUCHE

**[Exemple TR\_07]**

L'étape suivante va un peu compliquer le programme à traduire. Vous allez construire un projet plus ambitieux avec deux fiches et des textes à traduire.

- créez un nouveau projet ;
- dans « *Projet* » → « *Options du projet* » → « *i18n* », « cochez i18n et Créer/mettre à jour le fichier « .po » en enregistrant le fichier « .lfm » » ;
- modifiez la légende de la fiche (*Caption*) en la faisant passer de « *Form1* » à « *In English 5...* » ;
- ajoutez un bouton *TButton* à la fiche ;
- passez sa propriété *AutoSize* de *False* à *True* afin que la taille du bouton s'adapte automatiquement à celle de sa légende ;
- créez un gestionnaire d'événement *OnClick* pour ce bouton ;
- tapez le code suivant pour ce gestionnaire :

**implementation**

```
{SR *.lfm}
```

```
{ TForm1 }
```

**resourcestring**

```
RS_Hello = 'Hello world!';
```

```
RS_Bye = 'Goodbye cruel world!';
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
// *** inversion de la légende du bouton 1 ***
```

```
begin
```

```
  if Button1.Caption = RS_Hello then
```

```
    Button1.Caption := RS_Bye
```

```
  else
```

```
    Button1.Caption := RS_Hello;
```

```
end;
```



Remarquez que les chaînes ne sont elles aussi pas saisies en dur, c'est-à-dire qu'elles sont isolées dans une section particulière (*resourcestring*) qui indique qu'il s'agit de ressources qui feront l'objet d'un stockage particulier. Sans ce dernier, les traductions ne s'effectueraient pas, les libellés des constantes de ressources servant d'index au traducteur.

- créez aussi un gestionnaire d'événement *OnCreate* pour la fiche afin qu'une légende s'affiche correctement dès le lancement de l'application :

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
// *** création de l'application ***
```

```
begin
  Button1.Caption := RS_Hello;
end;
```

- ajoutez un second bouton **TButton** à cette fiche ;
- modifiez sa légende (**Caption**) de « *Button2* » à « *New...* » ;
- créez un gestionnaire d'événement **OnClick** pour ce bouton, mais laissez-le vide pour le moment ;
- cliquez sur « *Nouvelle fiche* » du menu « *Fichier* » ;
- modifiez la légende (**Caption**) de cette nouvelle fiche de « *Form2* » à « *New form* » ;
- ajoutez un composant **TBitBtn** à cette fiche ;
- modifiez sa propriété **Kind** de « *bkCustom* » à « *bkClose* » ;
- ajoutez du texte à la propriété **Hint** de ce bouton : « *Close the form* » ;
- passez sa propriété **ShowHint** de **False** à **True** afin de permettre l'affichage à l'exécution d'une bulle d'aide associée à ce bouton ;
- dans « *Projet* » → « *Options du projet* » → « *Fiches* », passez la fiche « *Form2* » de la colonne « *créer les fiches automatiquement* » à la colonne « *fiches disponibles* » avant de valider ce choix en cliquant sur « *OK* » ;
- dans la partie **implementation** de la première fiche **Form1**, ajoutez une clause **uses** afin que la seconde fiche soit connue de la première :

```
uses
  unit2 ;
```

- retournez au gestionnaire **OnClick** du second bouton de la première fiche (**Form1**) et entrez le code suivant :

```
procedure TForm1.Button2Click(Sender: TObject);
// *** ouverture d'une nouvelle fiche ***
var
  MyForm: TForm2;
begin
  MyForm := TForm2.Create(Self); // on crée la fiche
  try
    MyForm.ShowModal; // on la montre (seule active)
  finally
    MyForm.Close; // on libère la fiche
  end;
end;
```

- enregistrez le projet sous le nom **TestTranslate07.lpr** ;
- compilez et lancez l'application.

Vous disposez à présent d'une application un peu plus complexe que les précédentes, avec deux fiches dont une qui permet de faire surgir la seconde sous forme modale.

En dehors de sa relative complexité, cette application présente aussi la particularité d'être en anglais. L'objectif va évidemment consister à la traduire le plus simplement possible en français.

---

### FICHIERS LRT ET PO

---

Une première méthode consisterait à reprendre toutes les chaînes entrées et de les traduire. Si vous la choisissez, c'est que vous n'avez pas lu ce qui précédait !

La méthode la plus efficace va passer par la création d'un dossier *languages* dans lequel vous allez copier l'habituel fichier *lclstrconsts.fr.po* pour la traduction de la LCL, mais aussi le fraîchement créé *project5.po*.

Comme vous avez activé l'option « *i18n* » et l'enregistrement avec les fichiers LFM, **Lazarus** a créé automatiquement autant de fichiers LRT que d'unités et un unique fichier PO qui regroupe l'ensemble des chaînes à traduire.

En utilisant votre éditeur préféré, vous vous apercevrez que le fichier *unit1.lrt*, contient des paires de valeurs :

```
TFORM1.CAPTION=In English 5...  
TFORM1.BUTTON1.CAPTION=Button1  
TFORM1.BUTTON2.CAPTION=New...
```

Le fichier *unit2.lrt* est construit selon le même modèle :

```
TFORM2.CAPTION=New form  
TFORM2.BITBTN1.HINT=Close the form
```

De son côté, le contenu de *TestTranslate05.po*, un peu plus complexe, reprend les mêmes informations réparties sur trois lignes, accompagnées d'un en-tête et des chaînes de ressources incluses dans le code source :

```
msgid ""  
msgstr "Content-Type: text/plain; charset=UTF-8"  
  
#: tform1.button1.caption  
msgid "Button1"  
msgstr ""  
  
#: tform1.button2.caption  
msgid "New..."  
msgstr ""  
  
#: tform1.caption  
msgid "In English 5..."  
msgstr ""  
  
#: tform2.bitbtn1.hint  
msgid "Close the form"  
msgstr ""  
  
#: tform2.caption
```



```
msgid "New form"
msgstr ""

#: unit1.rs_bye
msgid "Goodbye cruel world !"
msgstr ""

#: unit1.rs_hello
msgid "Hello world !"
msgstr ""
```

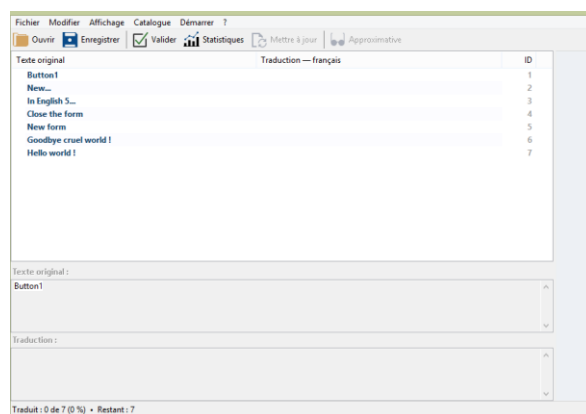
L'en-tête précise que le type de caractères utilisé est « *UTF-8* » pour la prise en compte des jeux de caractères différents suivant les langues. Cet en-tête contiendra plus tard un identificateur de la langue de traduction. Quant aux triplets de valeurs, ils comportent tous une troisième ligne réduite au code « *msgstr* » (pour *message string*) suivi d'une chaîne vide `""`. C'est cette chaîne vide qui contiendra la traduction désirée. Enfin, la première ligne de ces triplets correspond à un repère dans le code source ou dans le fichier LFM.

Même si la traduction peut se faire manuellement, l'utilisation d'outils spécialisés dans le traitement des fichiers PO est vivement recommandée : non seulement ils évitent bien des erreurs, mais ils fournissent aussi des outils d'édition et souvent des propositions de traduction qui s'appuient sur vos traductions et celles présentes sur Internet.

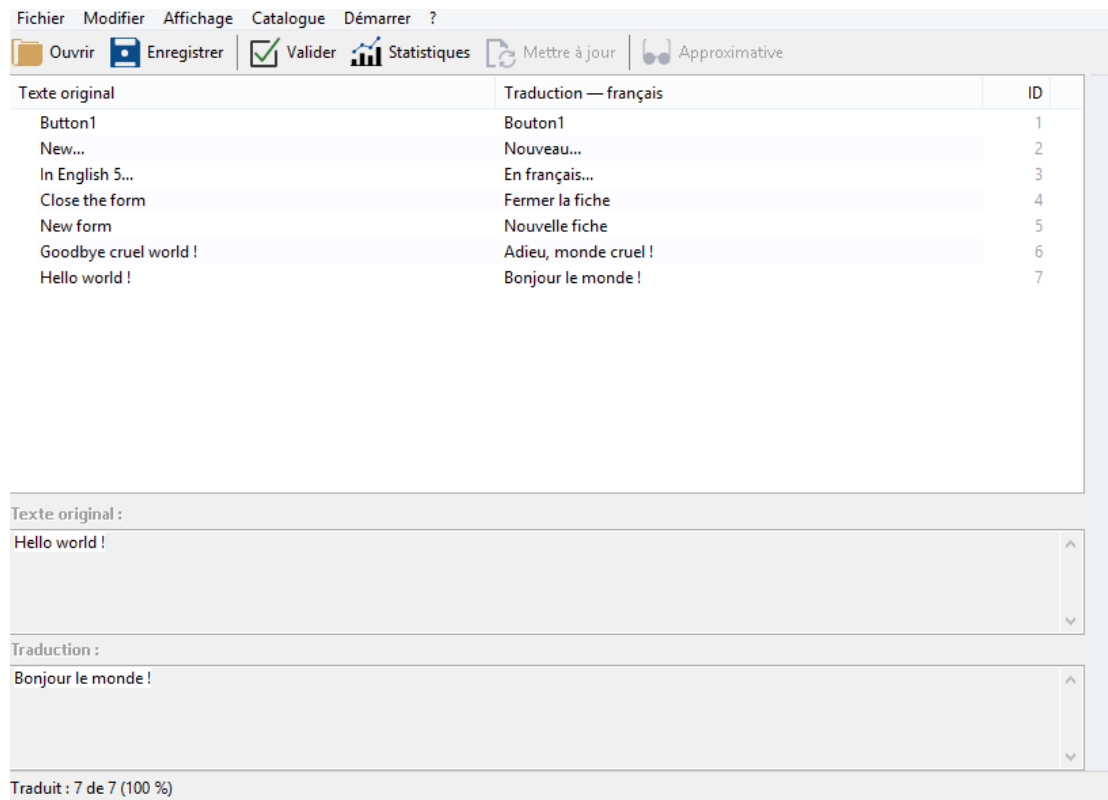
Pour Windows et Linux, un éditeur comme **poEdit** (gratuit dans sa version standard) est bien adapté. Il en existe d'autres parmi lesquels vous trouverez certainement celui qui vous convient le mieux.

Avant de traduire, le fichier souche doit être préservé : faites-en une copie dans le répertoire *languages* et rebaptisez-le *TestTranslate.fr.po*. L'infixe « *fr* » est celui qui indique qu'il s'agit de la traduction française : sans lui, il faudra préciser la langue de traduction.

**poEdit** reconnaît immédiatement cet infixe et présente le fichier sous cette forme :



Proposez la traduction suivante :



Après avoir enregistré votre travail de traduction, vous pouvez éditer le fichier modifié :

```
msgid ""
msgstr ""
"Content-Type: text/plain; charset=UTF-8\n"
"Project-Id-Version: \n"
"POT-Creation-Date: \n"
"PO-Revision-Date: \n"
"Last-Translator: \n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Transfer-Encoding: 8bit\n"
"Language: fr\n"
"X-Generator: Poedit 1.7.5\n"

#: tform1.button1.caption
msgid "Button1"
msgstr "Bouton1"

#: tform1.button2.caption
msgid "New..."
msgstr "Nouveau..."

#: tform1.caption
msgid "In English 5..."
msgstr "En français..."
```

```
#: tform2.bitbtn1.hint
msgid "Close the form"
msgstr "Fermer la fiche"

#: tform2.caption
msgid "New form"
msgstr "Nouvelle fiche"

#: unit1.rs_bye
msgid "Goodbye cruel world !"
msgstr "Adieu, monde cruel !"

#: unit1.rs_hello
msgid "Hello world !"
msgstr "Bonjour le monde !"
```

En dehors de l'en-tête qui a pris de l'ampleur afin de préciser si nécessaire la langue de traduction, l'identité du traducteur et/ou de son équipe, les dates de création et de modification et l'outil de traduction utilisé, vous remarquerez surtout que les troisièmes lignes déjà mentionnées de chaque triplet contiennent à présent la traduction proposée pour la chaîne originale correspondante.

---

### TRADUCTION AUTOMATIQUE COMPLETE

---

L'unité *DefaultTranslator* dispose de tout ce qui lui est nécessaire pour travailler :

- un répertoire *languages* pour y chercher les fichiers de traduction ;
- des fichiers PO qui contiennent les repères des chaînes à modifier ainsi que les couples de chaînes langue d'origine/langue de traduction.



Vous avez là l'explication de l'absence de traduction des chaînes codées en dur : il manque à l'unité les moyens de savoir où les situer sans ambiguïté.

En ajoutant tout simplement le nom de cette unité dans la clause `uses` du programme principal, vous obtenez... un programme en français !

```
program TestTranslate07;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, Unit1, Unit2,
  DefaultTranslator; // en français !

{$R *.res}

begin
```

```
RequireDerivedFormResource := True;  
Application.Initialize;  
Application.CreateForm(TForm1, Form1);  
Application.Run;  
end.
```

---

## ENCORE PLUS LOIN : DE L'ANGLAIS AU CHOIX DE LA LANGUE

---

Un degré de complexité sera franchi si vous souhaitez laisser le choix de la langue à l'utilisateur de votre programme. Partant d'une série de fichiers PO présents dans un répertoire donné, il s'agira de :

- générer la liste des langues disponibles ;
- proposer cette liste afin que l'utilisateur fasse son choix ;
- définir et mémoriser la langue en fonction de ce choix ;
- relancer le logiciel pour la prise en compte de cette langue.

---

## GENERER LA LISTE DES LANGUES ET CHOISIR LA LANGUE

---

Pour simplifier, certains aspects du problème seront traités sous leur forme la plus naïve : le programme saura d'emblée quels fichiers de traduction seront présents dans un répertoire donné et l'utilisateur en choisira un grâce à un contrôle de type *TListBox*.

### [Exemple TR\_08]

Sans surprise, l'unité principale du programme ressemblera à ceci :

```
unit main;  
  
interface  
  
uses  
Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,  
ExtCtrls,  
GVTranslate; // unité de la gestion des traductions  
  
type  
  
  { TMainForm }  
  
TMainForm = class(TForm)  
  btnRestart: TButton;  
  lblLanguage: TLabel;  
  lblDirectory: TLabel;  
  lblFile: TLabel;  
  lblAccess: TLabel;  
  lblLanguages: TListBox;  
  pnlData: TPanel;  
  procedure btnRestartClick(Sender: TObject);  
  procedure FormCreate(Sender: TObject);  
  procedure FormDestroy(Sender: TObject);
```

```
    procedure IbLanguagesClick(Sender: TObject);
    private
        Process: TGVTranslate; // traducteur
    end;

    var
        MainForm: TMainForm;

    implementation

    {$R *.lfm}

    resourcestring
        R_Language = 'Language: ';
        R_Directory = 'Directory: ';
        R_File = 'File: ';
        R_Access = 'Access: ';

    { TMainForm }

    procedure TMainForm.btnRestartClick(Sender: TObject);
    // *** bouton pour redémarrer le programme ***
    begin
        // choix enregistré
        Process.Language := IbLanguages.Items[IbLanguages.ItemIndex];
        // on redémarre
        Process.Restart;
    end;

    procedure TMainForm.FormCreate(Sender: TObject);
    // *** création de la fiche ***
    begin
        Process := TGVTranslate.Create; // nouveau traducteur créé
        // mise à jour des légendes des étiquettes
        IbLanguage.Caption := R_Language + Process.Language;
        IbDirectory.Caption := R_Directory + Process.FileDir;
        IbFile.Caption := R_File + Process.FileName;
        IbAccess.Caption := R_Access + Process.LanguageFile;
    end;

    procedure TMainForm.FormDestroy(Sender: TObject);
    // *** destruction de la fiche ***
    begin
        Process.Free; // traducteur libéré
    end;

    procedure TMainForm.IbLanguagesClick(Sender: TObject);
    // *** clic sur la liste de choix ***
    begin
        // on active le bouton si un choix a été fait
        btnRestart.Enabled := (IbLanguages.ItemIndex <> - 1);
    end;

end.
```

Vous aurez compris que la partie la plus intéressante est comprise dans une nouvelle unité : *GVTranslate*. C'est elle qui a en charge l'accès aux fichiers de langue, mais aussi le redémarrage de l'application après l'enregistrement des changements.

---

### LA MEMORISATION DU CHOIX ET LE REDEMARRAGE DE L'APPLICATION

---

Une première difficulté réside dans le fait qu'une application en cours d'exécution ne peut pas se modifier elle-même : il faut lancer un nouveau processus depuis celui en cours d'exécution avant de mettre fin à ce dernier. Deux autres difficultés tiennent à ce que les fichiers de traduction sont à identifier par leur extension et qu'ils ne sont pas forcément dans le répertoire de l'application.

La classe *TGVTranslate* a pour mission de résoudre ces problèmes :

```
{ TGVTranslate }

TGVTranslate = class
strict private
  FileName: string;
  FileDir: string;
  Language: string;
  function GetLanguageFile: string;
  procedure SetFileName(const AValue: string);
  procedure SetFileDir(const AValue: string);
  procedure SetLanguage(const AValue: string);
  procedure Translate;
public
  constructor Create;
  procedure Restart;
  property Language: string read fLanguage write SetLanguage;
  property FileName: string read fFileName write SetFileName;
  property FileDir: string read fFileDir write SetFileDir;
  property LanguageFile: string read GetLanguageFile;
end;
```

Si l'essentiel des fonctionnalités de cette classe renvoie au problème d'identification des fichiers, la méthode *Restart* s'occupe de faire redémarrer l'application. Pour cela, elle fait appel à une unité fournie par **Lazarus** : *UTF8Process*.

Voici le listing commenté de cette méthode :

```
procedure TGVTranslate.Restart;
// *** redémarrage de l'application ***
var
  Exe: TProcessUTF8;
begin
  Exe := TProcessUTF8.Create(nil); // processus créé
  try
    Exe.Executable := Application.ExeName; // il porte le nom de l'application
    // ajout des paramètres
    Exe.Parameters.Add(Language); // langue en paramètre
    Exe.Parameters.Add(FileDir); // répertoire
    Exe.Parameters.Add(FileName); // nom de fichier
```

```

Exe.Execute; // on démarre la nouvelle application
finally
  Exe.Free; // processus libéré
  Application.Terminate; // l'application en cours est terminée
end;
end;

```

L'application est par conséquent relancée avec trois paramètres sur la ligne de commande : la langue désirée, le chemin à suivre relatif au répertoire de l'application et le nom du fichier sans son extension.



Cette procédure est facilement réutilisable dans d'autres contextes.

Les méthodes en charge des propriétés sont assez simples si ce n'est qu'elles prévoient de leur donner des valeurs par défaut si elles étaient indéterminées :

```

const
  C_DefaultDir = 'languages';
  C_PoExtension = 'po';
  C_DefaultLanguage = 'en';

resourcestring
  RS_FallBackLanguage = 'auto';

{ TGVTranslate }

procedure TGVTranslate.SetLanguage(const AValue: string);
// *** détermine la langue pour la traduction ***
var
  LDummyLang: string;
begin
  if AValue = RS_FallBackLanguage then // langue de la machine ?
  begin
    LDummyLang := '';
    GetLanguageIDs(LDummyLang, fLanguage); // on retrouve son identifiant
  end
  else
    fLanguage := AValue; // nouvelle valeur
  end;

constructor TGVTranslate.Create;
// *** création ***
begin
  inherited Create;
  if Application.ParamCount > 0 then // au moins un paramètre ?
    Language := Application.Params[1] // c'est l'identifiant de la langue
  else
    Language := C_DefaultLanguage; // langue par défaut
  if Application.ParamCount > 1 then // au moins deux paramètres ?
    FileDir := Application.Params[2] // c'est le répertoire des fichiers
  else
    FileDir := ''; // répertoire par défaut
  if Application.ParamCount > 2 then // au moins trois paramètres ?
    FileName := Application.Params[3] // c'est le nom du fichier
  else

```

```

    FileName := ''; // fichier par défaut
    Translate;
end;

procedure TGVTranslate.SetFileName(const AValue: string);
// *** détermine le nom du fichier ***
begin
    if AValue <> '' then // pas valeur par défaut ?
        // à partir de l'extraction du nom du fichier
        fFileName := ExtractFileName(AValue)
    else
        // à partir du nom du programme
        fFileName := ExtractFileNameOnly(Application.ExeName);
    end;

function TGVTranslate.GetLanguageFile: string;
// *** construit et renvoie le chemin complet du fichier de traduction ***
begin
    Result := '.' + PathDelim + FileDir + PathDelim + FileName + '.' +
        Language + '.' + C_POExtension;
end;

procedure TGVTranslate.SetFileDir(const AValue: string);
// *** détermine le répertoire où sont les fichiers de traduction ***
begin
    fFileDir := AValue; // valeur affectée
    if fFileDir <> '' then // pas la valeur par défaut ?
        fFileDir := ExtractFilePath(fFileDir); // on récupère le chemin
    if fFileDir = '' then // chemin vide ?
        fFileDir := C_DefaultDir; // répertoire par défaut
    end;

```



On notera qu'en cohérence avec **Lazarus**, la langue par défaut est l'anglais et que les fichiers de traduction sont attendus par défaut dans le sous-répertoire *languages*.

Enfin, une ultime méthode procède à la traduction elle-même :

```

procedure TGVTranslate.Translate;
// *** traduction ***
var
    LF: string;
begin
    if Language = C_DefaultLanguage then // l'anglais n'a pas besoin d'être traité
        Exit;
    LF := LanguageFile; // fichier de traduction
    if FileExistsUTF8(LF) then // existe-t-il ?
        SetDefaultLang(Language, FileDir) // on traduit
    else
        Language := C_DefaultLanguage; // langue par défaut si erreur
        // accès au fichier de traduction de la LCL
        LF := '.' + PathDelim + FileDir + PathDelim + 'lclstrconsts' + '.' +
            Language + '.' + C_PoExtension;
    if FileExistsUTF8(LF) then // existe-t-il ?
        Translations.TranslateUnitResourceStrings('LCLStrConsts', LF); // on traduit

```



```
end;
```



L'appel à *Translate* se fait au cours même de la création de l'objet de type *TGVTranslate*. Il est primordial que cette création soit réalisée avant l'affichage des fenêtres du projet : une place privilégiée sera au tout début du gestionnaire *OnCreate* de la fiche principale.

---

## BILAN

---

Dans ce chapitre, vous aurez appris à :

- ✓ franciser un programme, y compris lors de l'affichage de messages d'erreurs ;
- ✓ paramétrer les options du compilateur pour enclencher le processus de traduction ;
- ✓ manipuler les fichiers PO ;
- ✓ laisser à l'utilisateur le choix de la langue qu'il préfère.