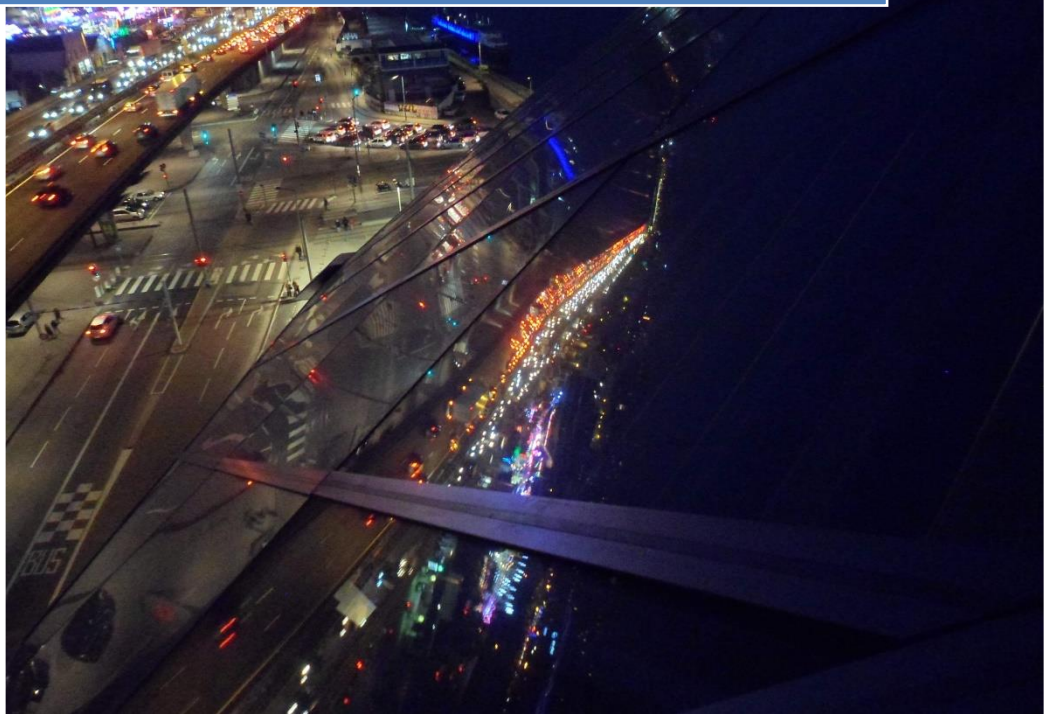


2015

Plus loin avec Free Pascal et Lazarus



Roland Chastain – Gilles Vasseur

07/06/2015

TABLE DES MATIERES

La tour de Babel : traduire une application.....	4
What's the matter ?.....	4
Un programme français... en anglais.....	5
Un peu de bricolage pour traduire.....	7
Une solution plus générale.....	10
Intérêt de ne pas bricoler la traduction	11
Fonctionnement de la solution générale	15
Une quatrième solution	16
De l'anglais au français.....	17
Préparation du programme souche.....	17
fichiers LRT et PO.....	19
Traduction automatique complète.....	23
Encore plus loin : de l'anglais au choix de la langue	24
Générer la liste des langues et choisir la langue.....	24
La mémorisation du choix et le redémarrage de l'application	26
Bilan.....	30
P00 à gogo : la Programmation Orientée Objet.....	31
Classes et objets.....	31
La programmation orientée objet	31
Classes.....	32
Champs, méthodes et propriétés	33
Les objets.....	36
Constructeur.....	38
Destructeur	40
Premiers gains de la P00	41
Principes et techniques de la P00.....	42
Encapsulation.....	42
Notion de portée.....	43
Héritage.....	44
Notion de polymorphisme.....	47
Les opérateurs IS et AS.....	49
Bilan.....	51
P00 à gogo : Les méthodes	52
Ce qu'il faut savoir.....	52
Méthodes statiques.....	52
Méthodes virtuelles	54

Compléments sur <i>inherited</i>	57
... et ce qu'il est utile de savoir	59
Méthodes abstraites	59
Méthodes de classe	61
Variables de classe et méthodes de classe statiques	67
Méthodes de message	69
Bilan	69

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 ?¹

Le programmeur qui souhaite adapter un logiciel à une autre langue pensera 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 ;
- 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. ») ;

¹ Quel est le problème ?

- 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.

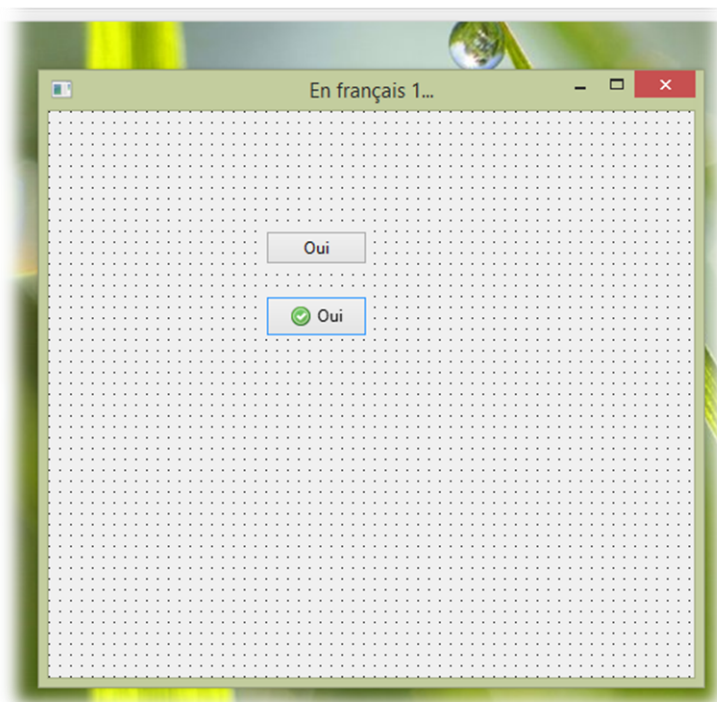
UN PROGRAMME FRANÇAIS... EN ANGLAIS

[Exemple TR_01]

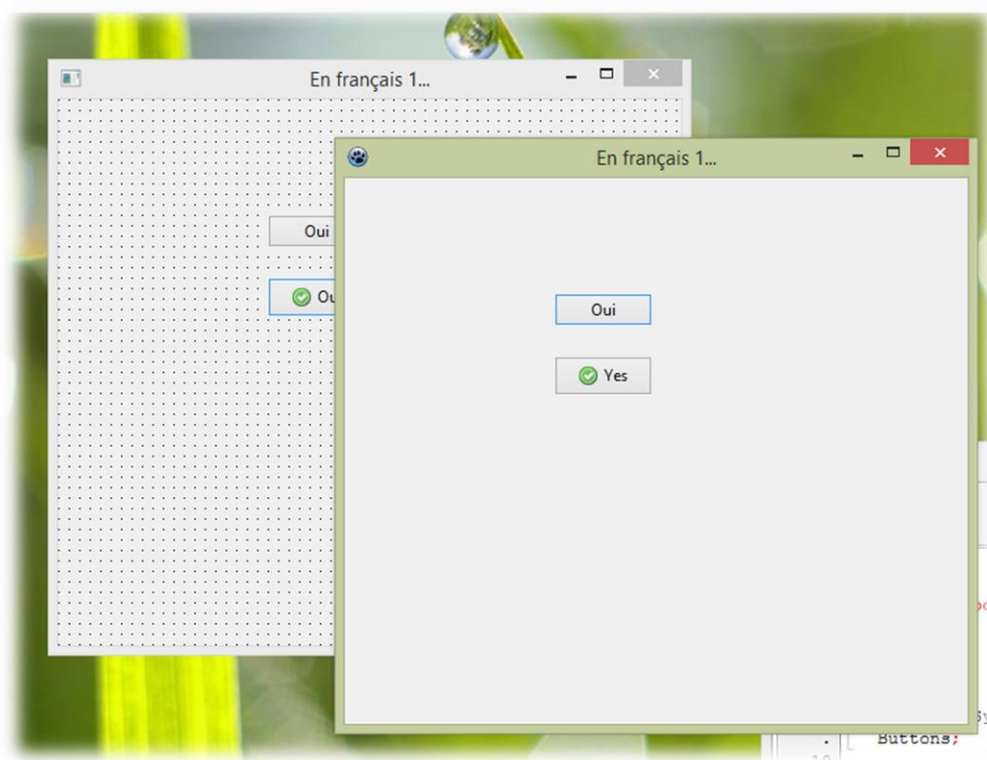
Pour vous en persuader, examinez le comportement d'un programme aussi élémentaire que celui-ci :

- 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.

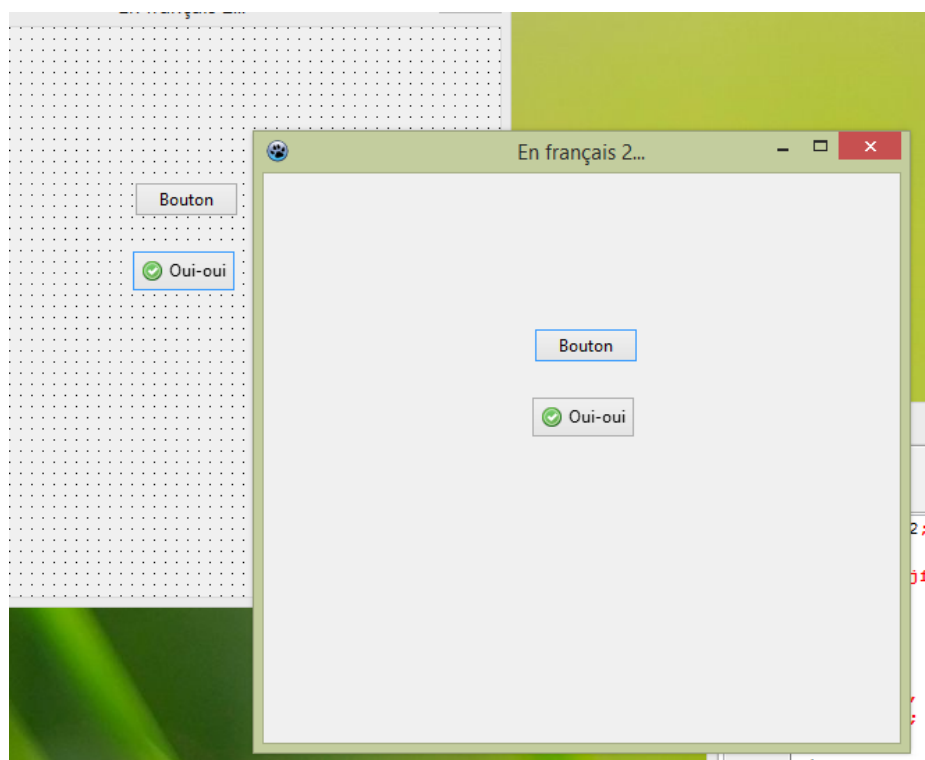
UN PEU DE BRICOLAGE POUR TRADUIRE

[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 :

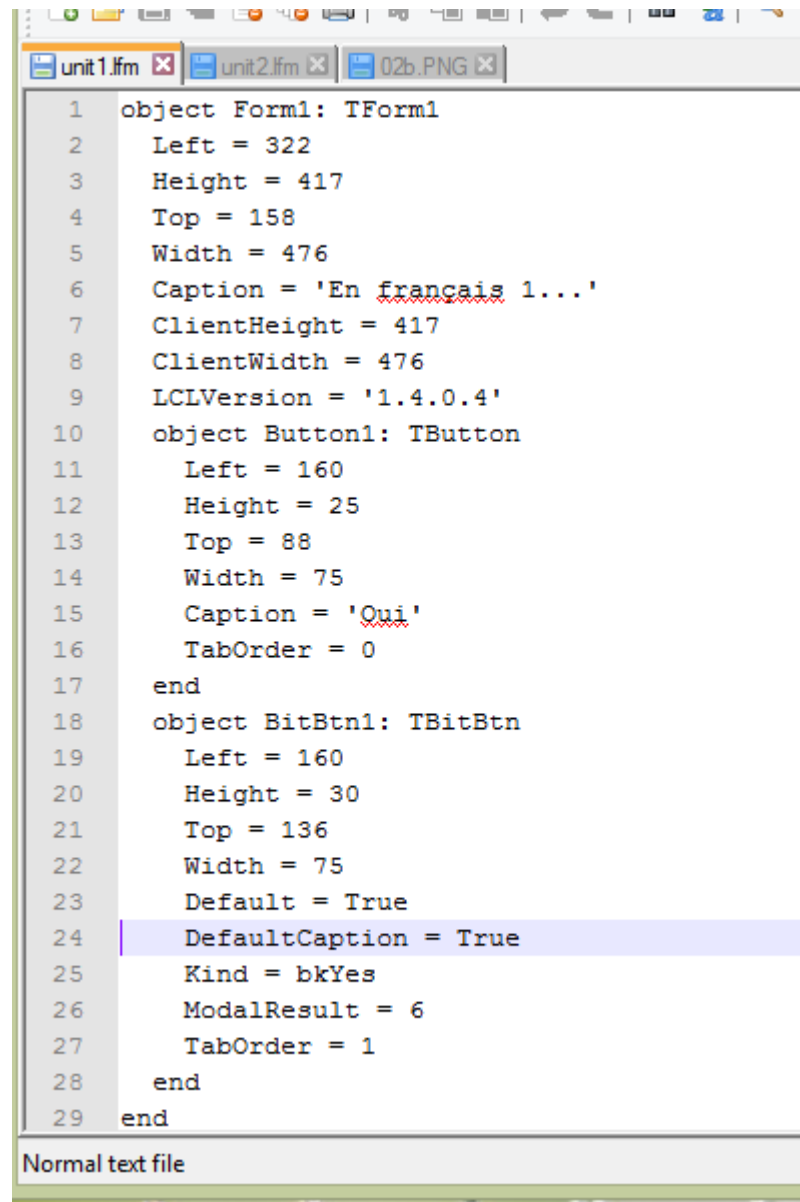
- modifiez la valeur de la légende (*Caption*) en la passant de **&Oui** à **&Oui-oui** ;
- compilez le programme ;
- lancez son exécution.

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 :

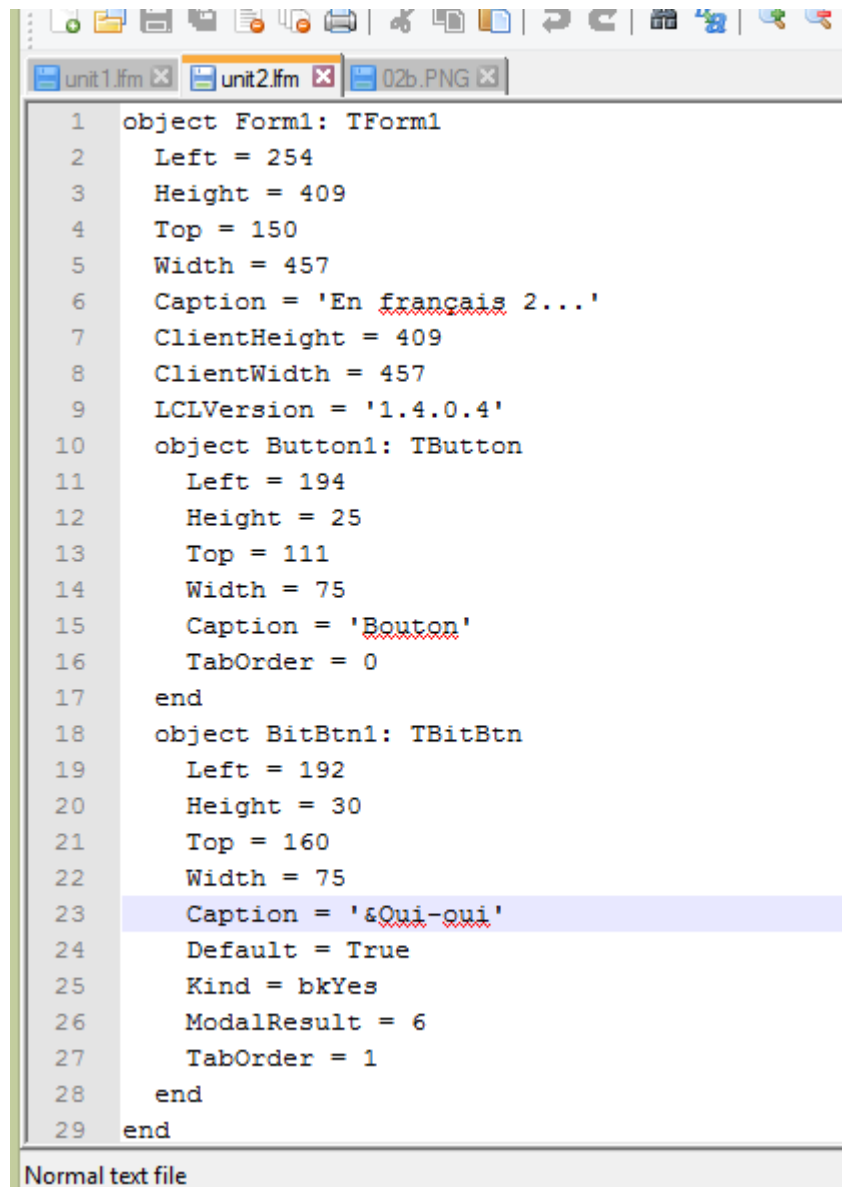


```
1 object Form1: TForm1
2   Left = 322
3   Height = 417
4   Top = 158
5   Width = 476
6   Caption = 'En français 1...'
7   ClientHeight = 417
8   ClientWidth = 476
9   LCLVersion = '1.4.0.4'
10  object Button1: TButton
11    Left = 160
12    Height = 25
13    Top = 88
14    Width = 75
15    Caption = '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

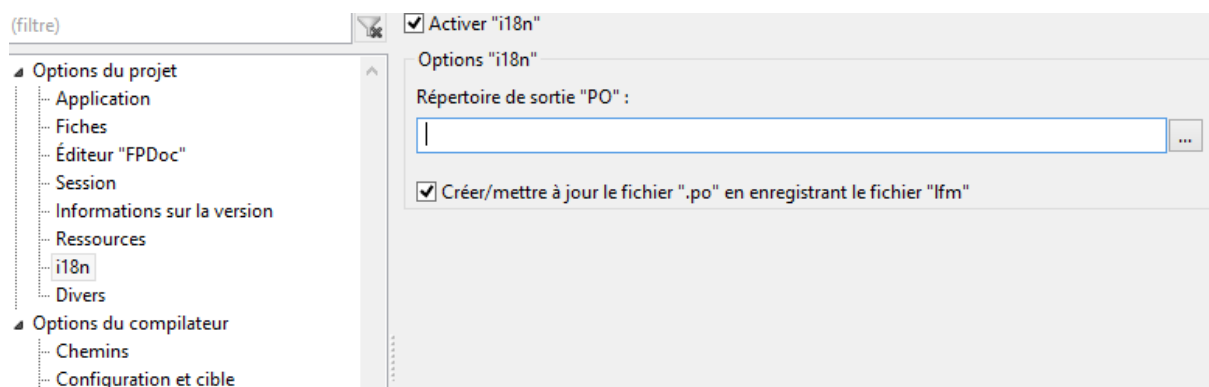
Vous pourriez vous satisfaire des deux premières solutions pour les propriétés accessibles en écriture. Malheureusement, de nombreux messages ne sont pas de ce type : certaines propriétés (comme le nom des couleurs) et la plupart des messages d'erreur sont hors de portée des unités créées.

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'origine² 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**.

² 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 !';

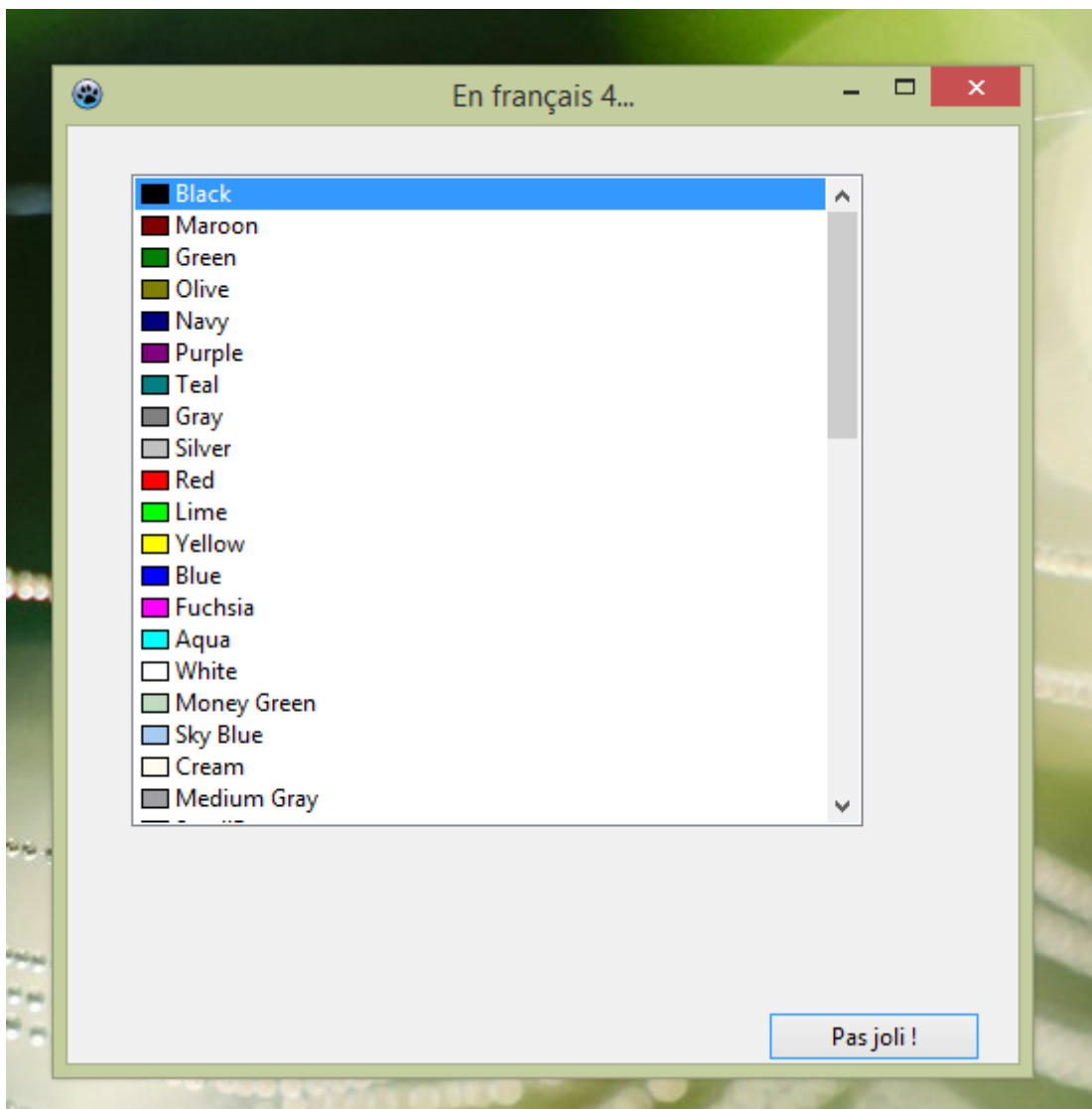
{$R *.lfm}

{ 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;
```

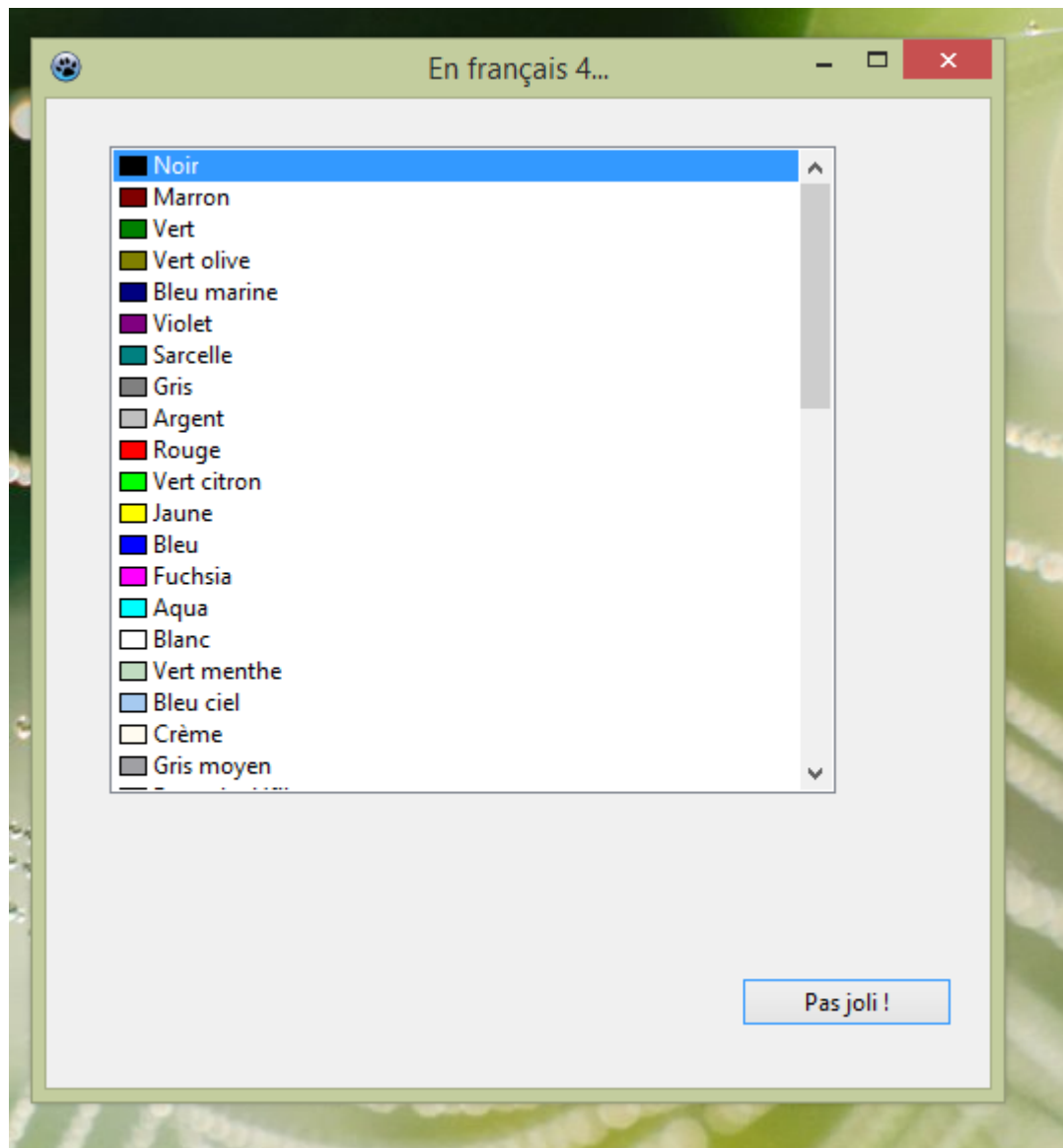
La procédure introduite permet de modifier l’affichage du bouton en fonction de la propriété `Style` de la `TColorListBox`. L’option qui alterne est `cbPrettyNames` : si elle est incluse dans le style, le composant fait appel à la LCL pour afficher le nom en clair des couleurs et non leur codage interne.

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**³ dans un sous-répertoire *languages* du répertoire de l’application, les noms de couleurs seront traduits :

³ 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 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

```
{ $R *.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 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 :

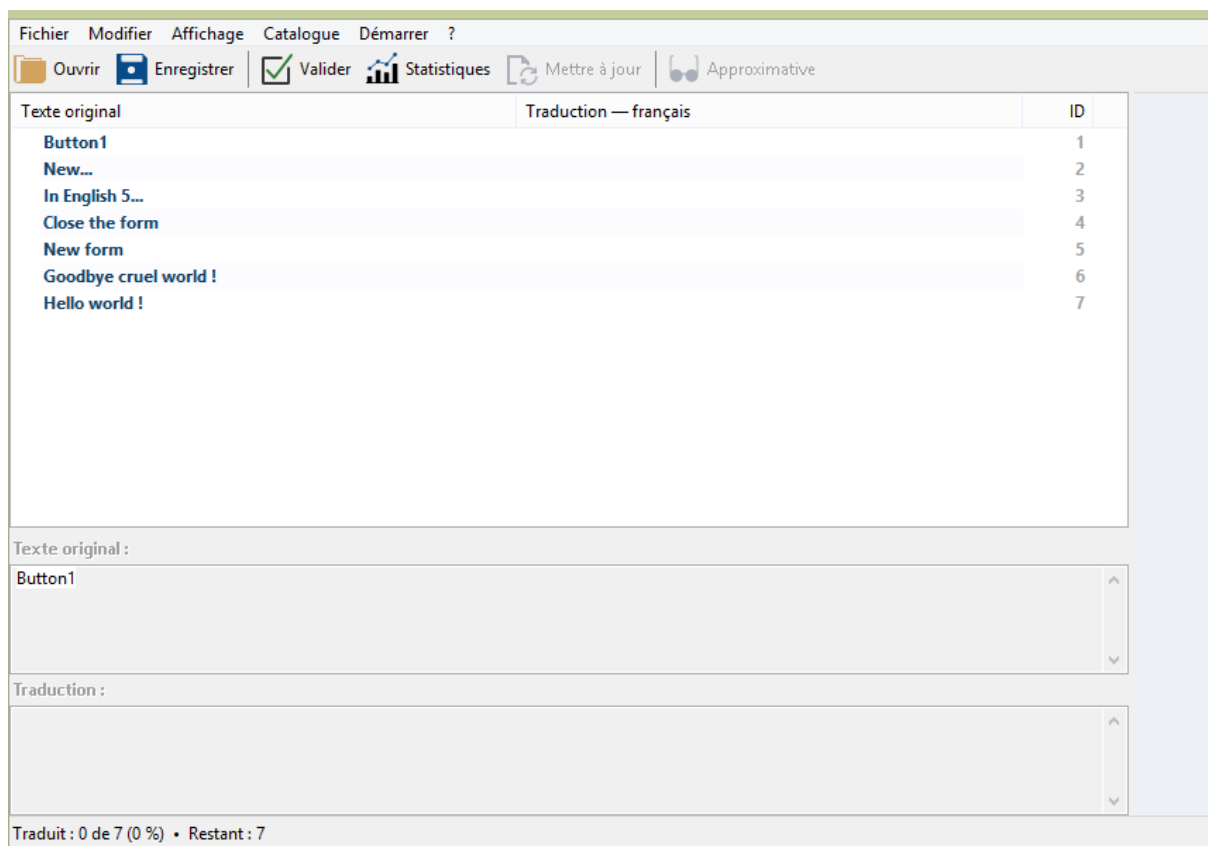
```
msgid ""  
msgstr "Content-Type: text/plain; charset=UTF-8"  
  
#: tform1.button1.caption  
msgid "Button1"  
msgstr ""  
  
#: tform1.button2.caption  
msgid "New..."  
msgstr ""  
  
#: tform1.caption  
msgid "In English 5..."  
msgstr ""  
  
#: tform2.bitbtn1.hint  
msgid "Close the form"  
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 source 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 infixes et présente le fichier sous cette forme :



Proposez la traduction suivante :

Fichier Modifier Affichage Catalogue Démarrer ?

Ouvrir Enregistrer Valider Statistiques Mettre à jour Approximative

Texte original	Traduction — français	ID
Button1	Bouton1	1
New...	Nouveau...	2
In English 5...	En français...	3
Close the form	Fermer la fiche	4
New form	Nouvelle fiche	5
Goodbye cruel world !	Adieu, monde cruel !	6
Hello world !	Bonjour le monde !	7

Texte original :

Hello world !

Traduction :

Bonjour le monde !

Traduit : 7 de 7 (100 %)

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;
  lbLanguages: TListBox;
  pnlData: TPanel;
  procedure btnRestartClick(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure lbLanguagesClick(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 := lbLanguages.Items[lbLanguages.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
  lblLanguage.Caption := R_Language + Process.Language;
  lblDirectory.Caption := R_Directory + Process.FileDir;
  lblFile.Caption := R_File + Process.FileName;
  lblAccess.Caption := R_Access + Process.LanguageFile;
end;

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

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

```

Vous aurez compris que la partie la plus intéressante est comprise dans une nouvelle unité : ***TGVTranslate***. 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
  fFileName: string;
  fFileDir: string;
  fLanguage: 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.

POO A GOGO : LA PROGRAMMATION ORIENTEE OBJET

Objectifs : dans ce chapitre, vous allez aborder certaines notions fondamentales pour exploiter au mieux la puissance de Free Pascal. Non seulement ce dernier est un héritier de la programmation structurée, mais il a été entièrement pensé pour manipuler au mieux des objets à travers le concept de classe. Sans imposer la Programmation Orientée Objet, Free Pascal (et plus encore Lazarus) invite fortement à souscrire à ses principes.

Sommaire : *Classes et objets :* La Programmation Orientée Objet – Classes – Champs, méthodes et propriétés – Les objets – Constructeur – Destructeur – Premiers gains de la POO – *Principes et techniques de la POO :* Encapsulation – Notion de portée – Héritage – Notion de polymorphisme – Les opérateurs *Is* et *As*

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

CLASSES ET OBJETS

LA PROGRAMMATION ORIENTEE OBJET

La *Programmation Structurée* a permis une meilleure lisibilité et par conséquent une maintenance améliorée des programmes en regroupant les instructions au sein de modules appelés *fonctions* et *procédures*. En créant ces éléments plus faciles à comprendre qu'une longue suite d'instructions et de sauts, les projets complexes devenaient maîtrisables.

La *Programmation Orientée Objet (POO)* se propose de représenter de manière encore plus rigoureuse et plus efficace les entités et leurs relations en les encapsulant au sein d'*objets*. Elle renverse d'une certaine façon la perspective en accordant toute leur place aux données alors que la programmation structurée privilégiait les actions.

En matière informatique, décrire le monde qui nous entoure consiste essentiellement à utiliser des trios de données : *entité*, *attribut*, *valeur*. Par exemple : (ordinateur, système d'exploitation, Windows 8.1), (ordinateur, système d'exploitation, Linux Mint 17), (chien, race, caniche), (chien, âge, 5), (chien, taille, petite), (cheveux, densité, rare). L'*entité* décrite est à l'intersection d'*attributs* variés qui servent à caractériser les différentes entités auxquelles ils se rapportent.

Ces trios prennent tout leur sens avec des *méthodes* pour les manipuler : création, insertion, suppression, modification, etc. Plus encore, les structures qui allieront les *attributs* et les *méthodes* pourront interagir afin d'échanger les informations nécessaires à un processus. Il devient ainsi possible de stocker et de manipuler des *entités* en

mémoire, chacune d'entre elles se décrivant par un ensemble *d'attributs* et un ensemble de *méthodes* portant sur ces attributs⁴.

CLASSES

La réunion des attributs et des méthodes permettant leur manipulation dans une même structure est le fondement de la **POO** : cette structure particulière prend le nom de *classe*. Par une première approximation, vous pouvez considérer une classe comme un enregistrement qui posséderait les procédures et les fonctions pour manipuler ses données. Vous pouvez aussi voir une classe comme une boîte noire fournissant un certain nombre de fonctionnalités à propos d'une entité aux attributs bien définis. Peu importe ce qu'il se passe dans cette boîte dans la mesure où elle remplit au mieux les tâches pour lesquelles elle a été conçue.

Imaginez un programme qui créerait des animaux virtuels et qui les animerait. En programmation procédurale classique, vous auriez à coder un certain nombre de fonctions, de procédures et de variables. Ce travail pourrait donner lieu à des déclarations comme celles-ci :

```
var
  V_Nom: string;
  V_AFaim: Boolean;
  V_NombreAnimaux: Integer;
// [...]
procedure Avancer ;
procedure Manger;
procedure Boire;
procedure Dormir;
function ASoif : Boolean ;
function AFaim: Boolean;
function ANom : string ;
procedure SetSoif(Valeur : Boolean) ;
procedure SetFaim(Valeur : Boolean) ;
procedure SetNom(Valeur : string) ;
```

Les difficultés commenceraient avec l'association entre les routines définies et un animal particulier. Vous pourriez par exemple créer un enregistrement représentant l'état d'un animal :

```
TEtatAnimal = record
  FNom: string;
  FAFaim: Boolean ;
  FASoif: Boolean ;
end;
```

⁴ *L'orienté objet* – Bersini Hugues – Eyrolles 2007

Ensuite, il vous faudrait regrouper les enregistrements dans un tableau et chercher des techniques permettant de reconnaître les animaux, de fournir leur état et de décrire leur comportement. Sans doute que certaines de vos routines auraient besoin d'un nouveau paramètre en entrée capable de distinguer l'animal qui fait appel à elles. Avec des variables globales, des tableaux, des boucles et beaucoup de patience, vous devriez vous en tirer. Cependant, si le projet prend de l'ampleur, les variables globales vont s'accumuler tandis que les interactions entre les procédures et les fonctions vont se complexifier : une erreur pourra se glisser dans leur intrication et il sera difficile de l'y déceler.

Dans un tel cas de figure, la **P00** va d'emblée montrer son efficacité. Il vous faudra déclarer une classe⁵ :

```
TAnimal = class
strict private
  fNom: string;
  fASoif: Boolean ;
  fAFaim: Boolean ;
  procedure SetNom(AValue: string);
public
  procedure Avancer;
  procedure Manger;
  procedure Boire;
  procedure Dormir;
published
  property ASoif: Boolean read fASoif write fASoif;
  property AFaim: Boolean read fAFaim write fAFaim;
  property Nom: string read fNom write SetNom;
end ;
```

À chaque fois qu'une variable sera du type de la classe définie, elle disposera à titre privé des *champs*, des *propriétés* et des *méthodes* proposées par cette classe.

CHAMPS, METHODES ET PROPRIETES

Les *champs* (ou *attributs*) décrivent la structure de la classe. *fASoif* est par exemple un champ de type booléen.

Les *méthodes* (procédures et fonctions) décrivent les opérations qui sont applicables grâce à la classe. *Avancer* est par exemple une méthode de *TAnimal*.

Une *propriété* est un moyen d'accéder à un champ : *fNom* est par exemple accessible grâce à la propriété *Nom*. Les propriétés se servent des mots réservés *read* et *write* pour cet accès⁶.

[Exemple PO_01]

⁵ Ne vous inquiétez pas si vous ne maîtrisez pas le contenu de cette structure : son étude se fera bientôt.

⁶ Les propriétés seront étudiées en détail dans le troisième chapitre sur la P00.

Pour avoir accès à la **POO** avec Free Pascal, vous devez activer l'une des trois options suivantes :

- {\$mode objfpc}
- {\$mode delphi}
- {\$mode MacPas}

La première ligne est incluse automatiquement dans le squelette de l'application lorsque vous la créez *via* Lazarus

Afin de préparer votre premier travail en relation avec les classes, procédez comme suit :

- créez une nouvelle application ;
- avec Fichier -> Nouvelle unité, ajoutez une unité à votre projet ;
- enregistrez les squelettes créés automatiquement par Lazarus sous les noms suivants : *project1.lpi* sous *TestPOO01.lpi* – *unit1.pas* sous *main.pas* – *unit2.pas* sous *animal.pas* ;
- dans la partie *interface* de l'unité *animal.pas*, créez une section *type* et entrez le code de définition de la classe *TAnimal* ;
- placez le curseur n'importe où dans la définition de la classe puis pressez simultanément sur **Ctrl-Maj-C** : Lazarus va créer instantanément le squelette de toutes les méthodes à définir.

À ce stade, l'unité devrait ressembler à ceci :

```
unit animal;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type

  { TAnimal }

TAnimal = class
strict private
  fNom: string;
  fASoif: Boolean ;
  fAFaim: Boolean ;
  procedure SetNom(AValue: string);
public
  procedure Avancer;
  procedure Manger;
  procedure Boire;
  procedure Dormir;
```

```

published
  property ASoif: Boolean read fASoif write fASoif;
  property AFaim: Boolean read fAFaim write fAFaim;
  property Nom: string read fNom write SetNom;
end ;

implementation

{ TAnimal }

procedure TAnimal.SetNom(AValue: string);
begin
  if fNom=AValue then Exit;
  fNom:=AValue;
end;

procedure TAnimal.Avancer;
begin

end;

procedure TAnimal.Manger;
begin

end;

procedure TAnimal.Boire;
begin

end;

procedure TAnimal.Dormir;
begin

end;

end.

```



Vous remarquerez qu'une des méthodes est déjà pré-remplie : il s'agit de *SetNom* qui détermine la nouvelle valeur de la propriété nom. Ne vous inquiétez pas de son contenu qui n'est pas utile à votre compréhension à ce stade.

La déclaration d'une classe se fait donc dans une section *type* de la partie *interface* de l'unité. On parle aussi d'*interface* à son propos, c'est-à-dire, dans ce contexte, à la partie visible de la classe. Il faudra bien sûr définir les comportements (que se passe-t-il dans le programme lorsqu'un animal mange ?) dans la partie *implementation* de la même unité. La seule différence entre la définition d'une méthode et celle d'une

procédure ou d'une fonction traditionnelle est que son identificateur porte le nom de la classe comme préfixe, suivi d'un point :

```
implementation
// [...]
procedure TAnimal.Avancer ;
begin

end ;
```

Complétez à présent votre programme :

- ajoutez une clause *uses* à la partie *implementation* de l'unité *animal.pas* ;
- complétez cette clause par *Dialogs* afin de permettre l'accès aux boîtes de dialogue ;
- insérez dans chaque squelette de méthode (sauf *SetNom*) une ligne du genre : *MessageDlg(Nom + ' mange...', mtInformation, mbOK, 0)*; en adaptant bien entendu le verbe à l'intitulé de la méthode.

Vous aurez compris que les méthodes complétées afficheront chacune un message comprenant le nom de l'animal tel que défini par sa propriété *Nom* suivi d'un verbe indiquant l'action en cours.

Reste à apprendre à utiliser cette classe qui n'est jusqu'à présent qu'une belle boîte sans vie.

LES OBJETS

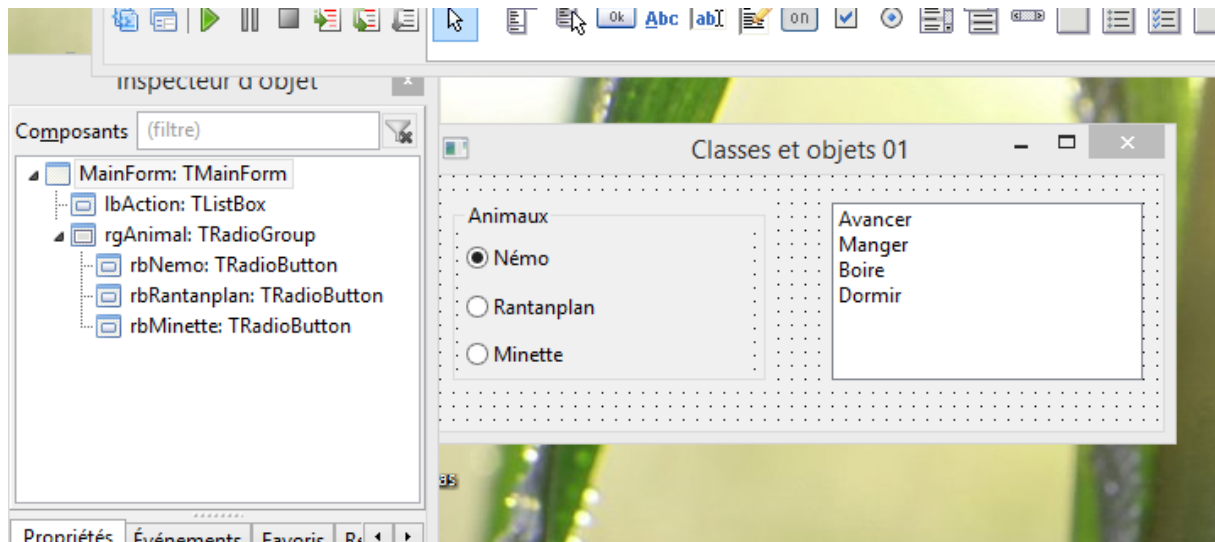
Contrairement à la *classe* qui est une structure abstraite, l'*objet* est la concrétisation de cette classe : on parlera d'*instanciation* pour l'action qui consiste essentiellement à allouer de la mémoire pour l'objet et à renvoyer un pointeur vers l'adresse de son implémentation. L'objet lui-même est une *instance* d'une classe.

Dans l'exemple en cours de rédaction, *Nemo*, *Rantanplan* et *Minette* pourront être trois variables, donc trois *instances*, pointant vers trois objets de type *TAnimal* (la classe). Autrement dit, une *classe* est un moule et les *objets* sont les entités réelles que l'on obtient à partir de ce moule⁷.

Pour avancer dans la réalisation de votre programme d'exemple, procédez comme suit :

- ajoutez cinq composants à votre fiche principale (*TListBox*, *TRadioGroup* comprenant trois *TRadioButton*), en les plaçant et les renommant selon le modèle suivant :

⁷ Vous verrez souvent le terme *objet* employé dans le sens de *classe*. S'il s'agit bien d'un abus de langage, sachez qu'il ne porte pas à conséquence dans la plupart des cas.



- cliquez sur la propriété *Items* du composant *TListBox* et complétez la liste qui apparaît toujours selon le modèle précédent : Avancer, Manger, Boire et Dormir ;
- dans la clause *uses* de la partie *interface* de *MainForm.pas*, ajoutez *animal* afin que cette unité soit connue à l'intérieur de la fiche principale :

```
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  ExtCtrls,
  animal; // unité de la nouvelle classe
```

- dans la partie *private* de l'interface de la fiche *TMainForm*, définissez quatre variables de type *TAnimal* : *Nemo*, *Rantanplan*, *Minette* et *UnAnimal* :

```
private
  { private declarations }
  Nemo, Rantanplan, Minette, UnAnimal : TAnimal;
public
  { public declarations }
end;
```

Vous aurez ainsi déclaré trois animaux grâce à trois variables du type *TAnimal*. Pour agir sur un animal particulier, il suffira que vous affectiez une de ces variables à la quatrième (*UnAnimal*) pour que l'animal concerné par vos instructions soit celui choisi :

```
UnAnimal := Rantanplan ;8
```

La façon d'appeler une méthode diffère de celle d'une routine traditionnelle dans la mesure où elle doit à la moindre ambiguïté être préfixée du nom de l'objet qui la convoque, suivi d'un point :

⁸ Pour les (très) curieux : cette affectation est possible, car ces variables sont des pointeurs vers les objets définis.

```
UnAnimal := Nemo ;  
UnAnimal .Avancer ; // Nemo sera concerné  
UnAnimal .Dormir ;  
UnAnimal .ASoif := False ;
```

C'est ce que vous allez implémenter en créant les gestionnaires *OnClick* des composants *lbAction*, *rbMinette*, *rbNemo* et *rbRantanplan*.

Pour ce faire :

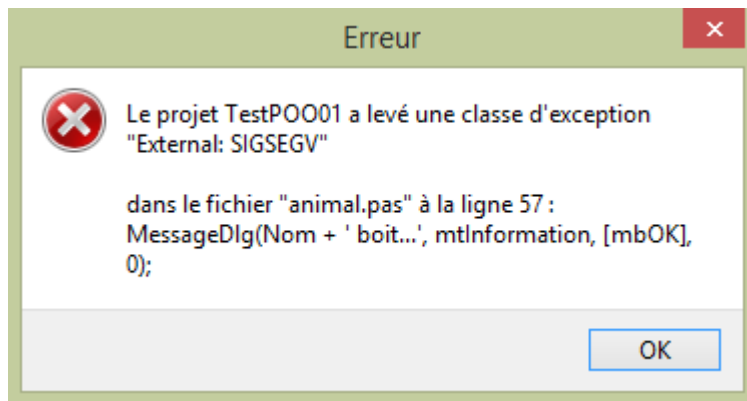
- cliquez tour à tour sur chacun des composants voulus de telle manière que Lazarus crée pour vous le squelette des méthodes ;
- complétez le corps des méthodes comme suit :

```
procedure TMainForm.lbActionClick(Sender: TObject);  
// *** choix d'une action ***  
begin  
  case lbAction.ItemIndex of // élément choisi dans TListBox  
    0: UnAnimal.Avancer;  
    1: UnAnimal.Manger;  
    2: UnAnimal.Boire;  
    3: UnAnimal.Dormir;  
  end;  
end;  
  
procedure TMainForm.rbMinetteClick(Sender: TObject);  
// *** l'animal est Minette ***  
begin  
  UnAnimal := Minette;  
end;  
  
procedure TMainForm.rbNemoClick(Sender: TObject);  
// *** l'animal est Némó ***  
begin  
  UnAnimal := Nemo;  
end;  
  
procedure TMainForm.rbRantanplanClick(Sender: TObject);  
// *** l'animal est Rantanplan ***  
begin  
  UnAnimal := Rantanplan;  
end;
```

CONSTRUCTEUR

Si vous lancez l'exécution de votre programme à ce stade, la compilation se déroulera normalement et vous pourrez agir sur les boutons radio sans problème.

Cependant, tout clic sur une action à réaliser par l'animal sélectionné provoquera une erreur fatale :



Dans son jargon, Lazarus vous prévient que le programme a rencontré une erreur de type **External: SIGSEGV**. Ce type d'erreur survient quand vous tentez d'accéder à une portion de mémoire qui ne vous est pas réservée.

L'explication de l'erreur est simple : l'objet en tant qu'instance d'une classe occupe de la mémoire, aussi est-il nécessaire de l'allouer et de la libérer. On utilise à cette fin un constructeur (*constructor*) dont celui par défaut est *Create*, et un destructeur (*destructor*) qui répond toujours au nom de *Destroy*.

Comme le monde virtuel est parfois aussi impitoyable que le monde réel, vous donnerez naissance aux animaux et libérerez la place en mémoire qu'ils occupaient quand vous aurez décidé de leur disparition. Autrement dit, il est de votre responsabilité de tout gérer⁹. L'instanciation de *TAnimal* prendra alors cette forme :

```
Nemo := TAnimal.Create ; // création de l'objet
// Ici, le travail avec l'animal créé...
```

La ligne qui crée l'objet est à examiner avec soin. L'objet n'existant pas avant sa création (un monde impitoyable est malgré tout rationnel), vous ne pourriez pas écrire directement une ligne comme :

```
MonAnimal.Create ; // je crois créer, mais je ne crée rien !
```

Le compilateur ne vous alerterait pas parce qu'il penserait que vous voulez faire appel à la méthode *Create* de l'objet *MonAnimal*¹⁰, ce qui est tout à fait légitime à la conception (et à l'exécution si l'objet est déjà créé). Le problème est que vous essayeriez d'exécuter une méthode à partir d'un objet *MonAnimal* qui n'existe pas encore puisque non créé... Une erreur de violation d'accès serait immédiatement déclenchée à l'exécution, car la mémoire nécessaire à l'objet n'aurait pas été allouée !

⁹ Vous verrez par la suite que cette obligation ne s'applique pas pour un objet dont le propriétaire est défini.

¹⁰ Il est possible d'appeler autant de fois que vous le désirez la méthode *Create*, même s'il est rare d'avoir à le faire.



C'est toujours en mentionnant le nom de la classe (ici, *TAnimal*) qu'on crée un objet.

Que vous utilisiez *Create* ou un constructeur spécifique, le fonctionnement de l'instanciation est toujours le même :

- le compilateur réserve de la place pour la variable du type de la classe à instancier : c'est un pointeur ;
- le constructeur de la classe appelle *getmem* pour réserver sur le tas la place à tout l'objet, initialise cette zone de mémoire et renvoie un pointeur vers elle.



La zone mémoire occupée par l'objet étant mise à zéro dès sa création, il n'est pas nécessaire d'initialiser les champs et les propriétés si vous souhaitez qu'ils prennent leur valeur nulle par défaut : chaîne vide, nombre à zéro... Pour rappel, la valeur nulle d'un pointeur est fournie par la constante *nil*.

À partir du moment où un objet a été créé, une variable appelée *Self* est définie implicitement pour chaque méthode de cet objet. Cette variable renvoie une référence aux données de l'objet. Son utilisation la plus fréquente est de servir de paramètre à une méthode ou à une routine qui a besoin de cette référence¹¹.

DESTRUCTEUR

Si l'oubli de créer l'instance d'une classe et son utilisation forcée provoquent une erreur fatale, s'abstenir de libérer l'instance d'une classe *via* un destructeur produira des *fuites de mémoire*, le système interdisant à d'autres processus d'accéder à des portions de mémoire qu'il pense encore réservées. Tout objet créé doit être détruit à la fin de son utilisation :

```
Nemo.Free ; // libération des ressources de l'objet
```



À propos de destructeur, le lecteur attentif est en droit de se demander pourquoi il est baptisé *Destroy* alors que la méthode utilisée pour la destruction de l'objet est *Free*. En fait, *Free* vérifie que l'objet existe avant d'appeler *Destroy*, évitant ainsi de lever de nouveau une exception pour violation d'accès. Ainsi, on définit la méthode *Destroy*, mais on appelle toujours la méthode *Free*.

Vous pouvez à présent terminer votre premier programme mettant en œuvre des classes créées par vos soins :

- définissez le gestionnaire *OnCreate* de la fiche principale :

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
```

¹¹ Pour des exemples d'utilisation de *Self*, voir le chapitre sur la LCL.

```
// on crée les instances et on donne un nom à chaque animal créé
Nemo := TAnimal.Create;
Nemo.Nom := 'Némo';
Rantanplan := TAnimal.Create;
Rantanplan.Nom := 'Rantanplan';
Minette := TAnimal.Create;
Minette.Nom := 'Minette';
// objet par défaut
UnAnimal := Nemo;
end;
```

- de la même manière, définissez le gestionnaire *OnDestroy* de cette fiche :

```
procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
// on libère toutes les ressources
Minette.Free;
Rantanplan.Free;
Nemo.Free;
end;
```

Vous pouvez enfin tester votre application et constater que les animaux sont reconnus ainsi que les actions à effectuer.

PREMIERS GAINS DE LA POO

Que gagne-t-on à utiliser ce mécanisme apparemment plus lourd que le précédent ?

- en premier lieu, le programmeur disposera de briques pour la conception de ses propres créations. C'est exactement ce que vous faites quand vous utilisez un composant de Lazarus. Ces briques préfabriquées font évidemment gagner beaucoup de temps.
- qui plus est, dans la mesure où la manière dont telle ou telle fonctionnalité est réalisée est indifférente, la modification de l'intérieur de la boîte n'influera en rien les autres programmes qui utiliseront la classe en cause¹². Dans l'exemple pour les animaux, vous pourriez fort bien décider que la méthode *Dormir* émette un bip : vous n'auriez qu'une ligne à ajouter au sein de cette méthode pour que tous les animaux bénéficient de ce nouveau comportement ;
- enfin, les données et les méthodes étant regroupées pour résoudre un micro-problème, la lisibilité et la maintenance de votre application s'en trouveront grandement facilitées. Circuler dans un projet de bonne dimension reviendra à examiner les interactions entre les briques dont

¹² Cette remarque ne vaut évidemment que si la classe a été bien conçue dès le départ ! En particulier, si l'ajout de nouvelles fonctionnalités est toujours possible, en supprimer interdirait toute réelle rétrocompatibilité.

il est constitué ou à étudier une brique particulière, au lieu de se perdre dans les méandres des bouts de codes entrecroisés.

Vous allez voir ci-après que les gains sont bien supérieurs encore. À partir du petit exemple produit, vous pouvez déjà pressentir la puissance de la POO : imaginez avec quelle facilité vous pourriez ajouter un nouvel animal ! De plus, n'êtes-vous pas étonné par ces méthodes *Create* et *Destroy* surgies de nulle part ? D'où viennent-elles ? Sachant qu'en Pascal tout se déclare, comment se fait-il qu'on puisse les utiliser sans apparemment avoir eu à les définir ?

PRINCIPES ET TECHNIQUES DE LA POO

ENCAPSULATION

Si vous reprenez l'interface de la classe *TAnimal*, fort de vos nouvelles connaissances, vous pourriez la commenter ainsi :

```
TAnimal = class // c'est bien une classe
strict private // indique que ce qui suit n'est pas visible à l'extérieur de la classe
  fNom: string; // un champ de type chaîne
  fASoif : Boolean ; // deux champs booléens
  fAFaim : Boolean ;
  procedure SetNom(AValue : string); // détermine la valeur d'un champ via une
méthode
  public // indique que ce qui suit est visible à l'extérieur de la classe
  procedure Avancer ; // des méthodes...
  procedure Manger;
  procedure Boire;
  procedure Dormir;
  // les propriétés permettent d'accéder aux champs
  // et/ou des méthodes manipulant ces champs
  property ASoif : Boolean read fASoif write fASoif;
  property AFaim : Boolean read fAFaim write SetAFaim;
  property Nom: string read fNom write SetNom;
end ;
```

L'*encapsulation* est le concept fondamental de la **POO**. Il s'agit de protéger toutes les données au sein d'une classe : en général, même si Free Pascal laisse la liberté d'une manipulation directe, seul l'accès à travers une méthode ou une propriété est autorisé.

Ainsi, aucun objet extérieur à une instance de la classe *TAnimal* ne connaîtra l'existence de *fAFaim* et donc ne pourra y accéder :

```
// Erreur : compilation refusée
MonObjet.AFaimAussi := MonAnimal.fAfaim ;
// OK si ATresSoif est une propriété booléenne modifiable de AutreObjet.
AutreObjet.ATresSoif := MonAnimal.AFaim ;
```

Paradoxalement, cette contrainte est une bénédiction pour le programmeur qui peut pressentir la fiabilité de la classe qu'il utilise à la bonne encapsulation des données. Peut-être le traitement à l'intérieur de la classe changera-t-il, mais restera cette interface qui rend inutile la compréhension de la mécanique interne.

NOTION DE PORTEE

Le niveau d'encapsulation est déterminé par la *portée* du champ, de la propriété ou de la méthode. La *portée* répond à la question : qui est autorisé à voir cet élément et donc à l'utiliser ?

Lazarus définit six niveaux de portée :

- *strict private* : l'élément n'est visible (donc utilisable) que par un autre élément de la même classe ;
- *private* : l'élément n'est visible que par un élément présent dans la même unité ;
- *strict protected* : l'élément n'est utilisable que par un descendant de la classe (donc une classe dérivée) présent dans l'unité ou dans une autre unité que celle de la classe ;
- *protected* : l'élément n'est utilisable que par un descendant de la classe (donc une classe dérivée), qu'il soit dans l'unité de la classe ou dans une autre unité y faisant référence, ou par une autre classe présente dans l'unité de la classe ;
- *public* : l'élément est accessible partout et par tous ;
- *published* : l'élément est accessible partout et par tous, et comprend des informations particulières lui permettant de s'afficher dans l'inspecteur d'objet de Lazarus.

Ces sections sont toutes facultatives : en l'absence de précision, les éléments de l'interface sont de type *public*.

Le niveau d'encapsulation repose sur une règle bien admise qui est de ne montrer que ce qui est strictement nécessaire. Par conséquent, choisissez la plupart du temps le niveau d'encapsulation le plus élevé possible pour chaque élément. L'expérience vous aidera à faire les bons choix : l'erreur sera donc souvent formatrice, bien plus que l'immobilisme !

Souvenez-vous tout d'abord que vous produisez des boîtes noires dans lesquelles l'utilisateur introduira des données pour en récupérer d'autres ou pour provoquer certains comportements comme un affichage, une impression, etc. Si vous autorisez la modification du cœur de votre classe et que vous la modifiez à votre tour, n'ayant *a priori* aucune idée du contexte de l'utilisation de votre classe, vous êtes assuré de perturber les programmes qui l'auront utilisée.

Aidez-vous ensuite de ces quelques repères :

- généralement, une section *strict private* abrite des champs et des méthodes qui servent d'outils de base. L'utilisateur de votre classe n'aura jamais besoin de se servir d'eux.

- une section *private* permet à d'autres classes de la même unité de partager des informations. Elle est très fréquente pour des raisons historiques : la section *strict private* est apparue tardivement.
- les variantes de *protected* permettent surtout des redéfinitions de méthodes¹³.
- la section *public* est la portée par défaut, qui n'a pas besoin de se faire connaître puisqu'elle s'offre à la première sollicitation venue !
- enfin, *published* sera un outil précieux lors de l'intégration de composants dans la palette de Lazarus.



Remarquez que la visibilité la plus élevée (*public* ou *published*) est toujours moins permissive qu'une variable globale : l'accès aux données ne peut s'effectuer qu'en spécifiant l'objet auquel elles appartiennent. Autrement dit, une forme de contrôle existe toujours à travers cette limitation intentionnelle. C'est dans le même esprit que les variables globales doivent être très peu nombreuses : visibles sans contrôle dans tout le programme, elles sont souvent sources d'erreurs parfois difficiles à détecter et à corriger.

HERITAGE

Jusqu'à présent, les classes vous ont sans doute semblé de simples enregistrements (*record*) aux capacités étendues : en plus de proposer une structure de données, elles fournissent les méthodes pour travailler sur ces données. Cependant, la notion de classe est bien plus puissante que ce qu'apporte l'encapsulation : il est aussi possible de dériver des sous-classes d'une classe existante qui hériteront de toutes les fonctionnalités de leur parent. Ce mécanisme s'appelle l'*héritage*.

Autrement dit, non seulement la classe dérivée saura exécuter un certain nombre de tâches qui lui sont propres, mais elle saura aussi, sans aucune ligne de code supplémentaire à écrire, exécuter toutes les tâches de son ancêtre.



Vous noterez qu'une classe donnée ne peut avoir qu'un unique parent, mais autant de descendants que nécessaire. L'ensemble forme une arborescence à la manière d'un arbre généalogique.

Encore plus fort : cet *héritage* se propage de génération en génération, la nouvelle classe héritant de son parent, de l'ancêtre de son parent, la chaîne ne s'interrompant qu'à la classe souche. Avec Lazarus, cette classe souche est toujours *TObject* qui définit les comportements élémentaires que partagent toutes les classes.

Ainsi, la déclaration de *TAnimal* qui commençait par la ligne *TAnimal = class* est une forme elliptique de *TAnimal = class(TObject)* qui rend explicite la parenté des deux classes.

¹³ Voir le chapitre suivant.



En particulier, vous trouverez dans *TObject* la solution au problème posé par l'apparente absence de définition de *Create* et de *Destroy* dans la classe *TAnimal* : c'est *TObject* qui les définit !

[Exemple PO_02]

Si vous manipulez la classe *TAnimal*, vous pourriez avoir à travailler avec un ensemble de chiens et envisager alors de créer un descendant *TChien* aux propriétés et méthodes étendues.

En voici une définition possible que vous allez introduire dans l'unité *animal.pas*, juste en-dessous de la classe *TAnimal* :

```
TChien = class(TAnimal)
strict private
  fBatard : Boolean ;
  procedure SetBatard(AValue: Boolean);
public
  procedure Aboyer;
  procedure RemuerDeLaQueue;
  property Batard: Boolean read fBatard write SetBatard;
end;
```

La première ligne indique que la nouvelle classe descend de la classe *TAnimal*. Les autres lignes ajoutent des fonctionnalités (*Aboyer* et *RemuerDeLaQueue*) ou déclarent de nouvelles propriétés (*Batard*). La puissance de l'héritage s'exprimera par le fait qu'un objet de type *TChien* disposera des éléments que déclare sa classe, mais aussi de tout ce que proposent *TAnimal* et *TObject*, dans la limite de la portée qu'elles définissent.

Comme pour la préparation de sa classe ancêtre, placez le curseur sur une ligne quelconque de l'interface de la classe *TChien* et pressez **Ctrl-Maj-C**. Aussitôt, Lazarus produit les squelettes nécessaires aux définitions des nouvelles méthodes :

```
property Nom: string read fNom write SetNom;
end ; // fin de la déclaration de TAnimal

{ TChien }

TChien = class(TAnimal)
strict private
  fBatard : Boolean ;
  procedure SetBatard(AValue: Boolean);
public
  procedure Aboyer;
  procedure RemuerDeLaQueue;
  property Batard: Boolean read fBatard write SetBatard;
end;
```

implementation

uses

Dialogs; // pour les boîtes de dialogue

{ TChien }

procedure TChien.SetBatard(AValue: Boolean);
begin

end;

procedure TChien.Aboyer;
begin

end;

procedure TChien.RemuerDeLaQueue;
begin

end;

{ TAnimal }

procedure TAnimal.SetNom(AValue: string); // [...]

D'ores et déjà, les lignes de code suivantes seront compilées et exécutées sans souci :

```
Medor := TChien.Create ; // on crée le chien Medor
Medor.Aboyer ; // la méthode Aboyer est exécutée
Medor.Batard := True ; // Medor n'est pas un chien de race
Medor.Manger ; // il a hérité de son parent la capacité Manger
Medor.Free ; // on libère la mémoire allouée
```

Comme les nouvelles méthodes ne font rien en l'état, complétez-les ainsi :

{ TChien }

procedure TChien.SetBatard(AValue: Boolean);
begin

fBatard := AValue;
end;

procedure TChien.Aboyer;
begin
 MessageDlg(Nom + ' aboie...', mtInformation, [mbOK], 0);
end;

```

procedure TChien.RemuerDeLaQueue;
begin
  MessageDlg(Nom + ' remue de la queue...', mtInformation, [mbOK], 0);
end;

```

De même, modifiez légèrement l'unité *main.pas* afin qu'elle prenne en compte cette nouvelle classe avec l'objet *Rantanplan* :

```

[...]
procedure rbRantanplanClick(Sender: TObject);
private
  { private declarations }
  Nemo, Minette, UnAnimal : TAnimal;
  Rantanplan: TChien; // <= changement
public
  { public declarations }
end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  // on crée les instances et on donne un nom à l'animal créé
  Nemo := TAnimal.Create;
  Nemo.Nom := 'Némo';
  Rantanplan := TChien.Create; // <= changement
  Rantanplan.Nom := 'Rantanplan';
  Minette := TAnimal.Create;

```

NOTION DE POLYMORPHISME

Lancez votre programme et essayez différents choix. Vous remarquerez que ce programme et celui qui n'avait pas défini *TChien* se comportent exactement de la même manière.

Que notre nouvelle application ne prenne pas en compte les nouvelles caractéristiques de la classe *TChien* n'a rien de surprenant puisque nous ne lui avons pas demandé de le faire, mais que notre *Rantanplan* se comporte comme un *TAnimal* peut paraître déroutant.

Par exemple, vous n'avez pas changé l'affectation de *Rantanplan* à *UnAnimal* qui est de type *TAnimal* :

```
procedure TMainForm.rbRantanplanClick(Sender: TObject);  
// *** l'animal est Rantanplan ***  
begin  
    UnAnimal := Rantanplan;  
end;
```

De même, si vous reprenez la partie de code qui correspond à un choix dans *TListBox*, vous constaterez qu'elle traite correctement le cas où *UnAnimal* est un *TChien* :

```
procedure TMainForm.lbActionClick(Sender: TObject);  
// *** choix d'une action ***  
begin  
    case lbAction.ItemIndex of  
        0: UnAnimal.Avancer;  
        1: UnAnimal.Manger;  
        2: UnAnimal.Boire;  
        3: UnAnimal.Dormir;  
    end;  
end;
```

La réponse à ce comportement étrange tient au fait que tout objet de type *TChien* est aussi de type *TAnimal*. En héritant de toutes les propriétés et méthodes publiques de son ancêtre, une classe peut légitimement occuper sa place si elle le souhaite : *Rantanplan* est donc un objet *TChien* ou un objet *TAnimal* ou, bien sûr, un objet *TObject*. C'est ce qu'on appelle le *polymorphisme* qui est une conséquence directe de l'héritage : un objet d'une classe donnée peut prendre la forme de tous ses ancêtres.

Grâce au polymorphisme, l'affectation suivante est correcte :

```
UnAnimal := Rantanplan ;
```

L'objet *Rantanplan* remplit toutes les conditions pour satisfaire la variable *UnAnimal* : en tant que descendant de *TAnimal*, il possède toutes les propriétés et méthodes à même de compléter ce qu'attend *UnAnimal*.

La réciproque n'est pas vraie et l'affectation suivante déclenchera dès la compilation une erreur, avec un message « types incompatibles » :

```
Rantanplan := UnAnimal ;
```

En effet, *UnAnimal* est incapable de renseigner les trois apports de la classe *TChien* : les méthodes *Aboyer*, *RemuerDeLaQueue* et la propriété *Batard* resteraient indéterminées.



Pour les curieux : certains d'entre vous auront remarqué que de nombreux gestionnaires d'événements comme *OnClick* comprennent un paramètre *Sender* de type *TObject*. Comme *TObject* est l'ancêtre de toutes les classes, grâce au polymorphisme, n'importe quel objet est accepté en paramètre. Ces gestionnaires s'adaptent donc à tous les objets qui pourraient faire appel à eux ! Élégant, non ?

LES OPERATEURS IS ET AS

Évidemment, il serait intéressant d'exploiter les nouvelles caractéristiques de la classe *TChien*. Mais comment faire puisque notre objet de type *TChien* est pris pour un objet de type *TAnimal* ?

Il existe heureusement deux opérateurs qui permettent facilement de préciser ce qui est attendu :

- *Is* vérifie qu'un objet est bien du type d'une classe déterminée. Il renvoie une valeur booléenne (*True* ou *False*) ;
- *As* force un objet à prendre la forme d'une classe déterminée. Si cette transformation (appelée *transtypage*) est impossible du fait de l'incompatibilité des types, une erreur est déclenchée.

Par conséquent, vous pourriez écrire ceci avec *is* :

```
If (Rantanplan is TChien) then // ce serait vrai
  Result := 'Il s'agit d'un chien'
else
  Result := 'Ce n'est pas un chien.' ;
// [...]
Result := (Minette is TChien); // faux
Result := (Nemo is TObject); // vrai
```

Et ceci avec *as*:

```
(Rantanplan as TChien).Aboyer ; // inutile mais correct
Rantanplan.Aboyer // équivalent du précédent
(Nemo as TChien).Dormir ; // erreur : Nemo n'est pas de type TChien
(UnAnimal as TChien).Manger ; // correct pour Rantanplan mais pas pour les
autres
```

Le déclenchement possible d'une erreur avec *as* conduit à l'accompagner la plupart du temps d'un test préalable avec *is* :

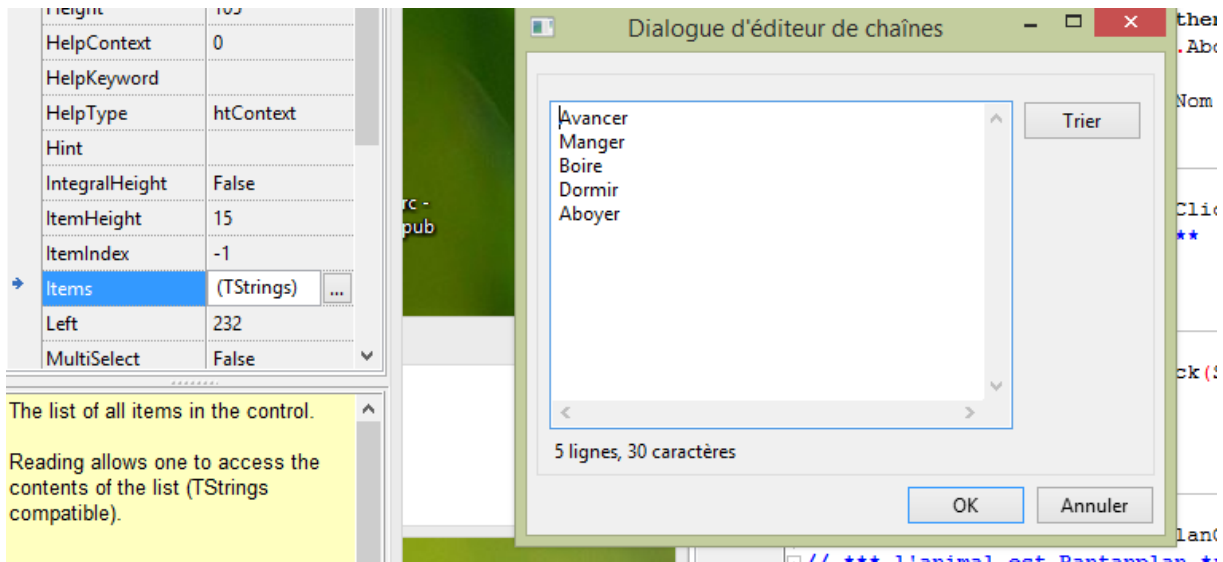
```
if (UnAnimal is TChien) then // l'objet est-il du type voulu ?
```

```
(UnAnimal as TChien).Aboyer; // si oui, transtypage avant d'exécuter la méthode
```

[Exemple PO_03]

Pour ce qui est du projet en cours, reprenez le programme et modifiez-le ainsi :

- ajoutez **Aboyer** à la liste des actions possibles dans le composant *lbAction* de type *TListBox* :



- modifiez la méthode *OnClick* de *lbAction* dans l'unité *main.pas* :

```
procedure TMainForm.lbActionClick(Sender: TObject);
// *** choix d'une action ***
begin
  case lbAction.ItemIndex of
    0: UnAnimal.Avancer;
    1: UnAnimal.Manger;
    2: UnAnimal.Boire;
    3: UnAnimal.Dormir;
    4: if UnAnimal is TChien then
        (UnAnimal as TChien).Aboyer
      else
        MessageDlg(UnAnimal.Nom + ' ne sait pas aboyer...', mtError, [mbOK], 0);
  end;
end;
```

La traduction en langage humain de cette modification est presque évidente : si l'objet *UnAnimal* est du type *TChien* alors forcer cet animal à prendre la forme d'un chien et à aboyer, sinon signaler que cet animal ne sait pas aboyer.

Dans ce chapitre, vous aurez appris à :

- ✓ comprendre ce qu'est la Programmation Orientée Objet à travers les notions d'encapsulation, de portée, d'héritage, de polymorphisme et de transtypage ;
- ✓ définir et utiliser les classes, les objets, les constructeurs, les destructeurs, les champs, les méthodes ;
- ✓ définir les propriétés.

POO A GOGO : LES METHODES

Objectifs : dans ce chapitre, vous allez consolider vos connaissances concernant la Programmation Orientée Objet en étudiant tour à tour les différents types de méthodes.

Sommaire : Méthodes statiques – Méthodes virtuelles – Compléments sur *inherited* – Méthodes abstraites – Méthodes de classe – Méthodes de classe statiques – Méthodes de message

Ressources : les programmes de test sont présents dans le sous-répertoire *poo2* du répertoire *exemples*.

CE QU'IL FAUT SAVOIR...

METHODES STATIQUES

Les méthodes *statiques* sont celles définies par défaut dans une classe. Elles se comportent comme des procédures ou des fonctions ordinaires à ceci près qu'elles ont besoin d'un objet pour être invoquées. Elles sont dites statiques parce que le compilateur crée les liens nécessaires dès la compilation : elles sont ainsi d'un accès particulièrement rapide, mais manquent de souplesse.

Une méthode statique peut être remplacée dans les classes qui en héritent. Pour cela, il suffit qu'elle soit accessible à la classe enfant : soit, bien que privée, elle est présente dans la même unité, soit elle est d'une visibilité supérieure et accessible partout.

Par exemple, en ce qui concerne la méthode *Manger* définie dans le parent *TAnimal*, vous estimerez à juste titre qu'elle a besoin d'être adaptée au régime d'un carnivore. Afin de la redéfinir, il suffirait de l'inclure à nouveau dans l'interface puis de coder son comportement actualisé.

[Exemple PO-04]

Reprenez le programme sur les animaux et modifiez-le selon le modèle suivant :

- ajoutez la méthode *Manger* à l'interface de la classe *TChien* :

```
TChien = class(TAnimal)
strict private
  fBatard : Boolean ;
  procedure SetBatard;
public
  procedure Manger; // <= la méthode est redéfinie
  procedure Aboyer;
```

```

procedure RemuerDeLaQueue;
property Batard: Boolean read fBatard write SetBatard;
end;

```

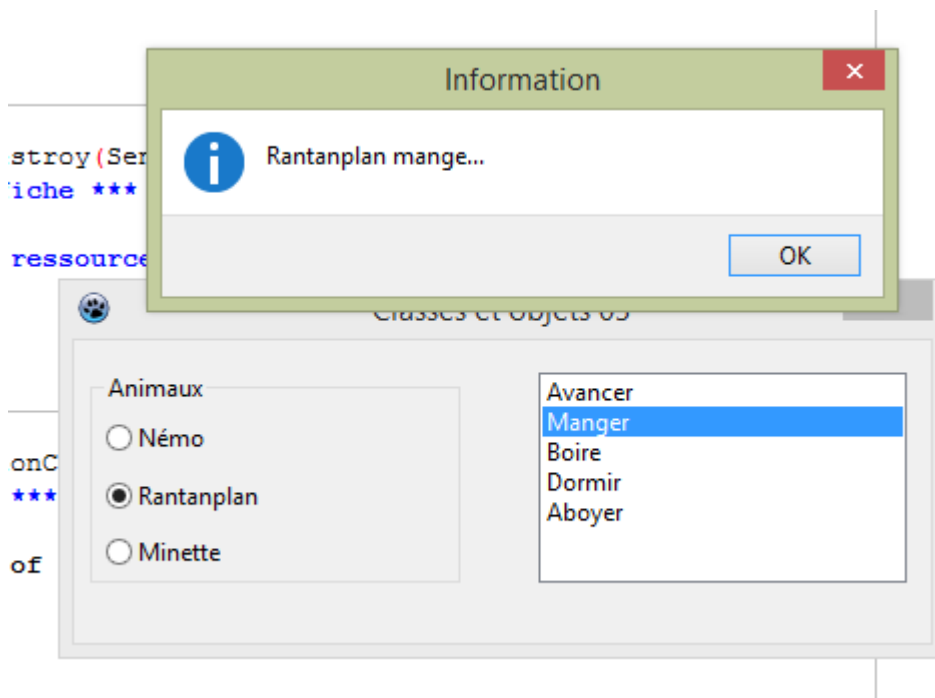
- pressez simultanément **Ctrl-Maj-C** pour demander à Lazarus de générer le squelette de la nouvelle méthode ;
- complétez ce squelette en vous servant du modèle suivant :

```

procedure TChien.Manger;
begin
  MessageDlg(Nom + ' mange de la viande...', mtInformation, [mbOK], 0);
end;

```

À l'exécution, si vous choisissez Rantanplan comme animal et que vous cliquez sur Manger, vous avez la surprise de voir que vos modifications semblent ne pas être prises en compte :



L'explication est à chercher dans le gestionnaire *OnClick* du composant *lbAction* :

```

procedure TMainForm.lbActionClick(Sender: TObject);
// *** choix d'une action ***
begin
  case lbAction.ItemIndex of
    0: UnAnimal.Avancer;
    1: UnAnimal.Manger; // <= ligne qui pose problème
    2: UnAnimal.Boire;
    3: UnAnimal.Dormir;
    4: if UnAnimal is TChien then // [...]

```

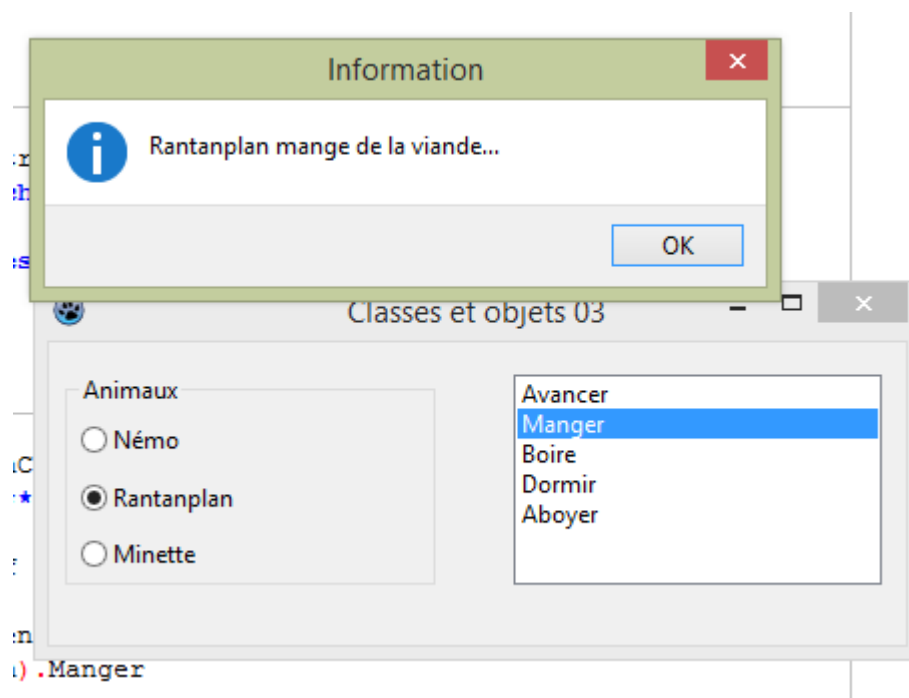
En effet, en écrivant *UnAnimal.Manger*, vous demandez à un animal de manger et non à un chien ! Vous obtenez logiquement ce que sait faire tout animal, à savoir manger, et non la spécialisation de ce que fait un chien carnivore.

Dès lors que votre classe *TChien* a redéfini le comportement de son parent, il faut modifier la ligne qui pose problème :

```
1 : if UnAnimal is TChien then
    (UnAnimal as TChien).Manger
else
    UnAnimal.Manger ;
```

Par ces lignes, vous forcez l'animal à prendre la forme d'un chien si c'est un chien qui est impliqué dans l'action : en termes plus abstraits, vous testez *UnAnimal* pour savoir s'il n'est pas du type *TChien* avant de le forcer à prendre cette forme et d'exécuter la méthode *Manger* adaptée.

À présent, vous obtenez bien le message qui correspond au régime alimentaire de Rantanplan :



Vous aurez noté que la redéfinition d'une méthode statique provoque le *remplacement* de la méthode de l'ancêtre. Mais comment modifier cette méthode de telle sorte qu'elle conserve les fonctionnalités de son ancêtre tout en en acquérant d'autres ?

METHODES VIRTUELLES

Une *méthode virtuelle* permet d'hériter du comportement de celle d'un parent tout en autorisant si nécessaire des compléments.

Contrairement aux méthodes statiques dont le compilateur connaît directement les adresses, les méthodes virtuelles sont accessibles en interne *via* une table¹⁴ d'exécution qui permet de retrouver les adresses de chacune des méthodes dont il a hérité et de celles qu'il a définies lui-même.

[Exemple PO-05]

Pour le programmeur, la déclaration d'une telle méthode se fait par l'ajout du mot *virtual* après sa déclaration. Il est aussi possible d'utiliser *dynamic* qui est strictement équivalent pour Free Pascal, mais qui a un sens légèrement différent avec Delphi¹⁵.

Vous allez modifier votre définition de la classe *TAnimal* en rendant virtuelle sa méthode *Manger* :

- reprenez le code source de l'unité *animal.pas* ;
- dans l'interface de *TAnimal*, ajoutez *virtual* après la déclaration de *Manger* :

```
public
  procedure Avancer;
  procedure Manger; virtual; // <= voici l'ajout
  procedure Boire;
```

Si vous exécutez le programme, son comportement ne change en rien du précédent. En revanche, lors de la compilation, Free Pascal aura émis un message d'avertissement : « une méthode héritée est cachée par TChien.Manger ». En effet, votre classe *TChien* qui n'a pas été modifiée redéfinit sans vergogne la méthode *Manger* de son parent : au lieu de la compléter, elle l'écrase comme une vulgaire méthode statique.

L'intérêt de la méthode virtuelle *Manger* est précisément que les descendants de *TAnimal* vont pouvoir la redéfinir à leur convenance. Pour cela, ils utiliseront l'identificateur *override* à la fin de la déclaration de la méthode redéfinie :

- modifiez l'interface de la classe *TChien* en ajoutant *override* après la définition de sa méthode *Manger* :

```
public
  procedure Manger; override; // <= ligne changée
  procedure Aboier;
  procedure RemuerDeLaQueue;
```

- recompilez le projet pour constater que l'avertissement a disparu ;

¹⁴ Cette table porte le nom de VMT : *Virtual Method Table*.

¹⁵ L'appel en Delphi des méthodes marquées comme *dynamic* diffère de l'appel des méthodes *virtual* : elles sont accessibles grâce à une table DMT plus compacte qu'une VMT mais plus lente d'accès. Les méthodes *dynamic* ont perdu de leur intérêt depuis l'adressage en au moins 32 bits.

- modifiez la méthode *Manger* pour qu'elle bénéficie de la méthode de son ancêtre :

```
procedure TChien.Manger;
begin
  inherited Manger; // on hérite de la méthode du parent
  MessageDlg('... mais principalement de la viande...', mtInformation, [mbOK], 0);
end;
```

On a introduit un mot réservé qui fait appel à la méthode du parent : *inherited*. Si vous lancez l'exécution du programme, vous constatez que choisir Rantanplan puis Manger provoque l'affichage de deux boîtes de dialogue successives : la première qui provient de *TAnimal* grâce à *inherited* précise que Rantanplan mange tandis que la seconde qui provient directement de *TChien* précise que la viande est son principal aliment¹⁶. Grâce à la table interne construite pour les méthodes virtuelles, le programme a été aiguillé correctement entre les versions de *Manger*.

Il est bien sûr possible de laisser tel quel le comportement d'une méthode virtuelle tout comme il est possible de modifier une méthode virtuelle que le parent aura ignoré et donc de remonter dans la généalogie. Très souvent, on définit une classe générale qui se spécialise avec ses descendants, sans avoir à tout prévoir avec l'ancêtre le plus générique et tout à redéfinir avec la classe la plus spécialisée¹⁷.



La méthode virtuelle aura toujours la même forme, depuis l'ancêtre le plus ancien jusqu'au descendant le plus profond : même nombre de paramètres, du même nom, dans le même ordre et du même type.

Reste une possibilité assez rare mais parfois utile : vous avez vu que redéfinir complètement une méthode virtuelle par une méthode statique provoquait un avertissement du compilateur. Il est possible d'imposer ce changement au compilateur en lui disant en quelque sorte que cet écrasement est voulu. Pour cela, faites suivre la redéfinition de votre méthode virtuelle par le mot réservé *reintroduce* :

```
// méthode du parent
TAnimal = class
// [...]
procedure Manger ; virtual ; // la méthode est virtuelle
// [...]
// méthode du descendant
TAutreAnimal = class(TAnimal)
procedure Manger ; reintroduce ; // la méthode virtuelle est écrasée
```

¹⁶ On a là une illustration du polymorphisme : un objet de type *TChien* est vraiment un objet de type *TAnimal*, mais qui possède ses propres caractéristiques.

¹⁷ En fait, le mécanisme est si intéressant qu'il suffit de jeter un coup d'œil à la LCL pour voir qu'il est omniprésent !

À présent, la méthode *Manger* est redevenue statique et tout appel à elle fera référence à sa version redéfinie.

Étant donné la puissance et la souplesse des méthodes virtuelles, vous vous demanderez peut-être pourquoi elles ne sont pas employées systématiquement : c'est que leur appel est plus lent que celui des méthodes statiques et que la table des méthodes consomme de la mémoire supplémentaire. En fait, utilisez la virtualité dès qu'une des classes qui descendrait de votre classe serait susceptible de spécialiser ou de compléter certaines de ses méthodes. C'est ce que vous avez fait avec la méthode *Manger* : elle renvoie à un comportement général, mais sera probablement précisée par les descendants de *TAnimal*.

Pour résumer :

- on ajoute *virtual* à la fin de la ligne qui définit une première fois une méthode virtuelle ;
- *dynamic* est strictement équivalent à *virtual* (mais a un sens différent avec Delphi) ;
- on ajoute *override* à la fin de la ligne qui redéfinit une méthode virtuelle dans un de ses descendants ;
- on utilise *inherited* à l'intérieur de la méthode virtuelle redéfinie pour hériter du comportement de son parent ;
- on utilise éventuellement *reintroduce* à la fin de la ligne pour écraser l'ancienne méthode au lieu d'en hériter.

COMPLEMENTS SUR *INHERITED*

D'un point de vue syntaxique, *inherited* est souvent employé seul dans la mesure où il n'y a pas d'ambiguïté quant à la méthode héritée. Les deux formulations suivantes sont par conséquent équivalentes :

```
procedure TChien.Manger;
begin
  inherited Manger; // on hérite de la méthode de l'ancêtre
// ou
  inherited ; // équivalent
[...]
```

La place de *inherited* au sein d'une méthode a son importance : si l'on veut modifier le comportement du parent, il est très souvent nécessaire d'appeler en premier lieu *inherited* puis d'apporter les modifications. Lors d'un travail de nettoyage du code, il est au contraire souvent indispensable de nettoyer ce qui est local à la classe enfant avant de laisser le parent faire le reste du travail.

Ainsi, *Create* et *Destroy* sont toutes les deux des méthodes virtuelles. Leur virtualité s'explique facilement, car la construction et la destruction d'un objet varieront sans doute suivant la classe qui les invoquera.

Lorsque vous redéfinirez *Create*, il est fort probable que vous ayez à procéder ainsi :

```
constructor Create ;  
begin  
    inherited Create ; // on hérite  
    // ensuite votre travail d'initialisation  
    // [...]  
end;
```

Il faut en effet vous dire que vous ne connaissez pas toujours exactement les actions exécutées par tous les ancêtres de votre classe : êtes-vous sûr qu'aucun d'entre eux ne modifiera pour ses propres besoins une propriété que vous voulez initialiser à votre manière ? Dans ce cas, les *Create* hérités annuleraient votre travail !

Pour *Destroy*, le contraire s'applique : vous risquez par exemple de vouloir libérer des ressources qui auront déjà été libérées par un ancêtre de votre classe et par conséquent de provoquer une erreur. La forme habituelle du destructeur *Destroy* hérité sera donc :

```
destructor Destroy ;  
begin  
    // votre travail de nettoyage  
    // [...]  
    inherited Create ; // on hérite ensuite !  
end;
```

Par ailleurs, *inherited* peut être appelé à tout moment dans le code de définition de la classe. Il est parfaitement légal d'avoir une méthode statique ou virtuelle dont le code serait ceci :

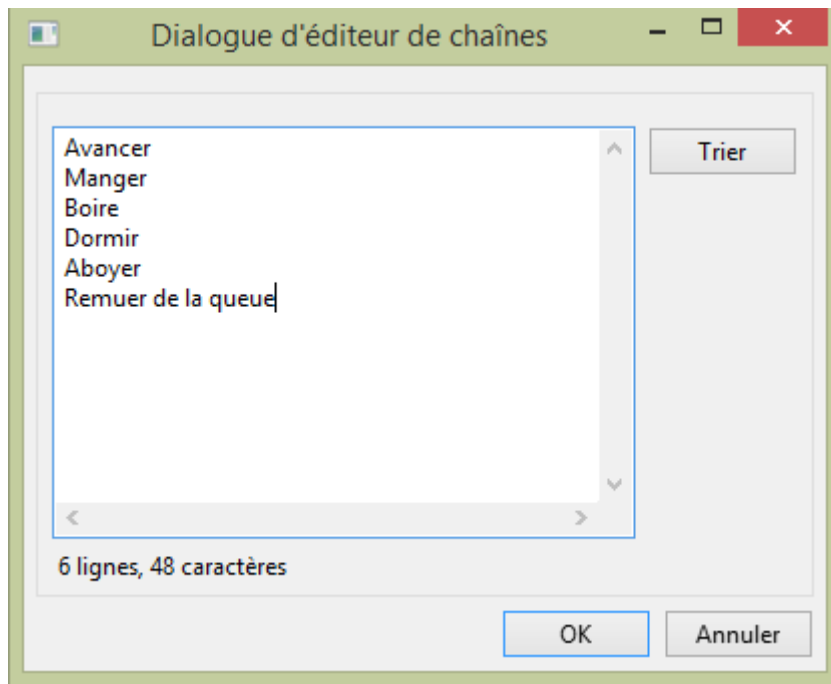
```
procedure TChien.RemuerDeLaQueue;  
begin  
    inherited Manger; // <= Manger vient de TAnimal !  
    MessageDlg('C'est pourquoi il remue de la queue...', mtInformation, [mbOK], 0);  
end;
```

La méthode héritée est celle qui affiche simplement le nom de l'animal en précisant qu'il mange. On explique ensuite la conséquence dans une nouvelle boîte de dialogue : on exprimerait ainsi le fait que le chien mange et qu'il en est très satisfait !

[Exemple PO-06]

Pour obtenir ce résultat, procédez ainsi :

- ajoutez « Remuer de la queue » à la liste des actions possibles de *lbAction* :



- ajoutez les lignes suivantes à l'événement *OnClick* du même composant :

```

else
    MessageDlg(UnAnimal.Nom + ' ne sait pas aboyer...', mtError, [mbOK], 0);
5: if UnAnimal is TChien then // <= nouvelle portion de code
    (UnAnimal as TChien).RemuerDeLaQueue
else
    MessageDlg(UnAnimal.Nom + ' ne sait pas remuer de la queue...', mtError, [mbOK],
0);
end;

```

- dans *animal.pas*, remplacez le code de la méthode *RemuerLaQueue* par le code proposé ci-avant.

À l'exécution, vous avez bien les messages adaptés qui s'affichent. Vous vérifiez une nouvelle fois que le polymorphisme en tant que conséquence de l'héritage permet à un objet de type *TChien* de prendre la forme d'un *TChien* ou d'un *TAnimal* suivant le contexte.

... ET CE QU'IL EST UTILE DE SAVOIR

METHODES ABSTRAITES

Il peut être utile dans une classe qui servira de moule à d'autres classes plus spécialisées de déclarer une méthode qui sera nécessaire, mais sans savoir à ce stade comment l'implémenter. Il s'agit d'une sorte de squelette de classe dont les descendants auront tous un comportement analogue. Dans ce cas, plutôt que de laisser cette méthode vide, ce qui n'imposerait pas de redéfinition et risquerait de déstabiliser l'utilisateur face à un code qui ne produirait aucun effet, on déclarera cette méthode avec *abstract*. Appelée à être vraiment définie, elle sera par ailleurs toujours une méthode virtuelle.

Simplement, faute d'implémentation, on prendra bien garde de ne pas utiliser *inherited* lors d'un héritage direct : une erreur serait bien évidemment déclenchée.

Examinez par exemple la classe *TStrings*. Cette dernière est chargée de gérer à son niveau fondamental une liste de chaînes et ce sont ses descendants qui implémenteront les méthodes qui assureront le traitement réel des chaînes manipulées¹⁸.

Voici un court extrait de son interface :

```
protected
  procedure DefineProperties(Filer: TFile); override;
  procedure Error(const Msg: string; Data: Integer);
  // [...]
  function Get(Index: Integer): string; virtual; abstract; // attention : deux
qualifiants
  function GetCapacity: Integer; virtual;
  function GetCount: Integer; virtual; abstract; // idem
```

On y reconnaît une méthode statique (*Error*), une méthode virtuelle redéfinie (*DefineProperties*) et une méthode virtuelle simple (*GetCapacity*). Nouveauté : les méthodes *Get* et *GetCount* sont marquées par le mot-clé *abstract* qui indique que *TStrings* ne propose pas d'implémentations pour ces méthodes parce qu'elles n'auraient aucun sens à son niveau.

Les descendants de *TStrings* procéderont à cette implémentation tandis que *TStrings*, en tant qu'ancêtre, sera d'une grande polyvalence. En effet, si vous ne pourrez jamais travailler avec cette seule classe puisqu'un objet de ce type déclencherait des erreurs à chaque tentative (même interne) d'utilisation d'une des méthodes abstraites, l'instancier permettra à n'importe quel descendant de prendre sa forme.

Comparez :

```
procedure Afficher(Sts: TStringList);
var
  LItem: String; // variable locale pour récupérer les chaînes une à une
begin
  for LItem in Sts do // on balaie la liste
    writeln(LItem); // et on affiche l'élément en cours
end;

et :
```

```
procedure Afficher(Sts: TStrings); // <= seul changement
var
  LItem: String;
begin
  for LItem in Sts do
    writeln(LItem);
```

¹⁸ Une des classes les plus utilisées, *TStringList*, a pour ancêtre *TStrings*.

```
end;
```

La première procédure affichera n'importe quelle liste de chaînes provenant d'un objet de type *TStringList*. La seconde acceptera tous les descendants de *TStrings*, y compris *TStringList* et est par conséquent bien plus polyvalente.

Au passage, vous aurez encore vu une manifestation de la puissance du polymorphisme : bien qu'en partie abstraite, *TStrings* pourra être utile puisqu'une classe qui descendra d'elle prendra sa forme en comblant ses lacunes !

METHODES DE CLASSE

Free Pascal offre aussi la possibilité de définir des *méthodes de classe*. Avec elles, on ne s'intéresse plus à la préparation de l'instanciation, mais à la manipulation directe de la classe. Dans d'autres domaines, on parlerait de métadonnées. Il est par conséquent inutile d'instancier une classe pour accéder à ces méthodes particulières, même si on peut aussi y accéder depuis un objet.

[Exemple PO-07]

La déclaration d'une méthode de classe se fait en plaçant le mot-clé *class* avant de préciser s'il s'agit d'une procédure ou d'une fonction. Par exemple, vous pourriez décider de déclarer une fonction qui renverrait le copyright associé à votre programme sur les animaux :

- ouvrez l'unité *animal.pas* et modifiez ainsi la déclaration de la classe *TAnimal* :

```
{ TAnimal }

TAnimal = class
private
  fNom: string;
  fASoif: Boolean ;
  fAFaim: Boolean ;
  procedure SetNom(AValue: string);
public
  procedure Avancer;
  procedure Manger; virtual;
  procedure Boire;
  procedure Dormir;
  class function Copyright: string; // <= modification !
```

- pressez **Ctrl-Maj-C** pour créer le squelette de la nouvelle fonction que vous complèterez ainsi :

```
class function TAnimal.Copyright: string;
begin
  Result := 'Roland Chastain - Gilles Vasseur 2015';
end;
```

- observez l'en-tête de cette fonction qui reprend *class* y compris dans sa définition ;
- dans l'unité main.pas, complétez le gestionnaire de création de la fiche *OnCreate* :

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
    // on crée les instances et on donne un nom à l'animal créé
    Nemo := TAnimal.Create;
    Nemo.Nom := 'Némo';
    Rantanplan := TChien.Create;
    Rantanplan.Nom := 'Rantanplan';
    Minette := TAnimal.Create;
    Minette.Nom := 'Minette';
    MainForm.Caption := MainForm.Caption + ' - ' + TAnimal.Copyright; // <= nouveau !
    // objet par défaut
    UnAnimal := Nemo;
end;
```

En lançant le programme, vous obtiendrez un nouveau titre pour votre fiche principale, agrégeant l'ancienne dénomination et le résultat de la fonction *Copyright*. L'important est de remarquer que l'appel a pu s'effectuer sans instancier *TAnimal*.

Bien sûr, vous auriez pu vous servir d'un descendant de *TAnimal* : *TChien* ferait aussi bien l'affaire puisque cette classe aura hérité *Copyright* de son ancêtre. De même, vous auriez tout aussi bien pu vous servir d'une instance d'une de ces classes : *Rantanplan*, *Nemo* ou *Minette*. Les méthodes de classe obéissent en effet aux mêmes règles de portée et d'héritage que les méthodes ordinaires. Elles peuvent être virtuelles et donc redéfinies.

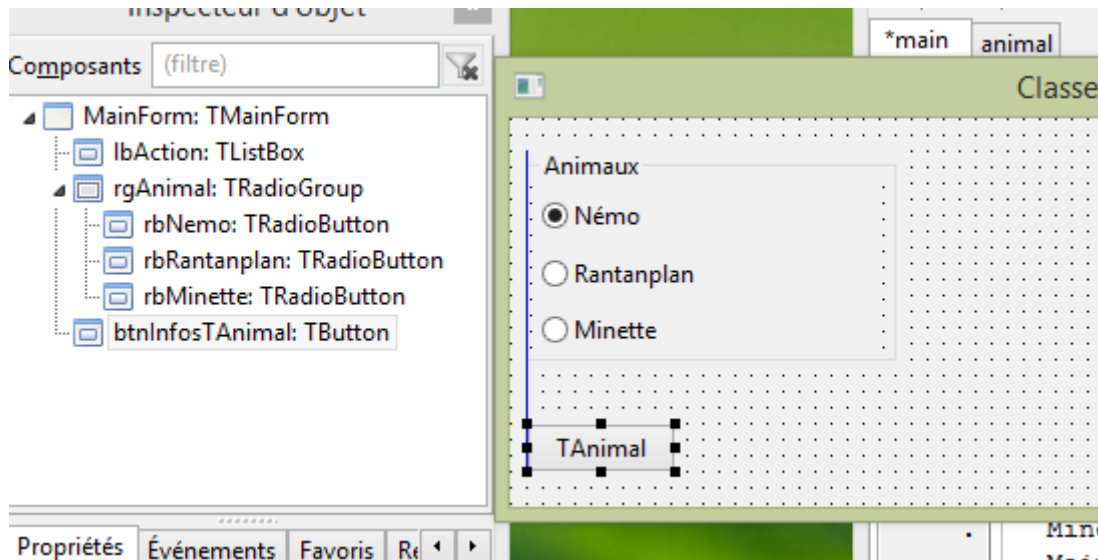
Leurs limites découlent de leur définition même : comme elles sont indépendantes de l'instanciation, elles ne peuvent pas avoir accès aux champs, propriétés et méthodes ordinaires de la classe à laquelle elles appartiennent. De plus, depuis une méthode de classe, *Self* pointe non pas vers l'instance de la classe mais vers la table des méthodes virtuelles qu'il est alors possible d'examiner.

Leur utilité est manifeste si l'on désire obtenir des informations à propos d'une classe et non des instances qui seront créées à partir d'elle.

[Exemple PO-08]

Afin de tester des applications possibles des méthodes de classe, reprenez le projet en cours :

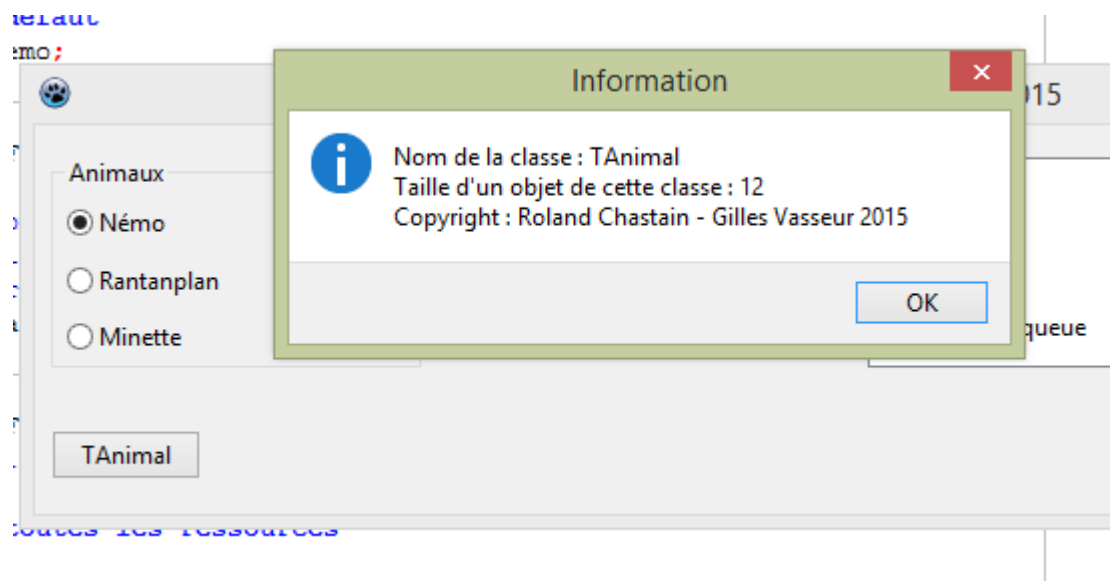
- ajoutez un bouton à la fiche principale, renommez-le *btnInfosTAnimal* et changez sa légende en *TAnimal* ;



- créez un gestionnaire *OnClick* pour ce bouton et complétez-le ainsi :

```
procedure TMainForm.btnInfosTAnimalClick(Sender: TObject);
begin
  MessageDlg('Nom de la classe : ' + TAnimal.ClassName +
    #13#10'Taille d'un objet de cette classe : ' + IntToStr(TAnimal.InstanceSize) +
    #13#10'Copyright : ' + TAnimal.Copyright
    , mtInformation, [mbOK], 0);
end;
```

En cliquant à l'exécution sur le bouton, vous afficherez ainsi le nom de la classe, la taille en octets d'un objet de cette classe et le copyright que vous avez défini précédemment :



Mais où les méthodes de classe *ClassName* et *InstanceSize* ont-elles été déclarées ? Elles proviennent de l'ancêtre *TObject* qui les définit par conséquent pour toutes les classes. Vous pourrez donc vous amuser à remplacer dans ce cas *TAnimal* par n'importe

quelle autre classe accessible depuis votre code : *TChien*, bien sûr, mais aussi *TForm*, *TButton*, *TListBox*... C'est ainsi que vous verrez qu'un objet de type *TChien* occupe 16 octets en mémoire alors qu'un objet de type *TForm* en occupe 1124...

Une application immédiate de ces méthodes de classe résidera dans l'observation de la généalogie des classes. Pour cela, vous utiliserez une méthode de classe nommée *ClassParent* qui fournit un pointeur vers la classe parente de la classe actuelle. Vous remonterez dans les générations jusqu'à ce que ce pointeur soit à *nil*, c'est-à-dire jusqu'à ce qu'il ne pointe sur rien.

[Exemple PO-09]

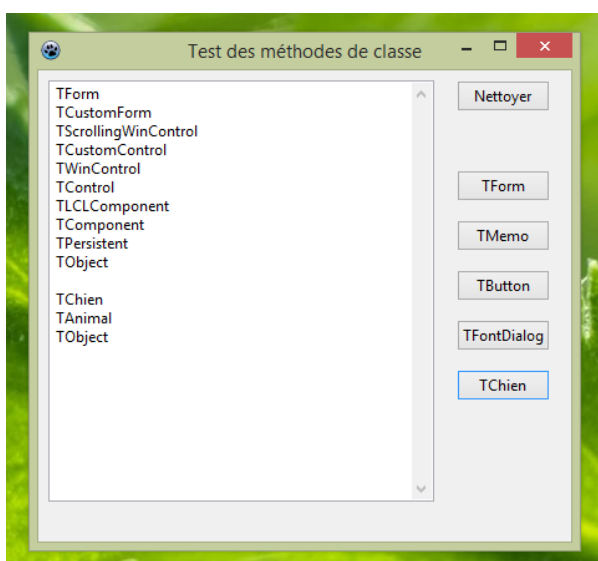
En utilisant un composant *TMemo* nommé *mmoDisplay*, la méthode d'exploration pourra ressembler à ceci :

```
procedure TMainForm.Display(AClass: TClass);
begin
  repeat
    mmoDisplay.Lines.Add(AClass.ClassName);
    AClass := AClass.ClassParent;
  until AClass = nil;
  mmoDisplay.Lines.Add("");
end;
```

Vous remarquerez que le paramètre qu'elle prend est de type *TClass* : il ne s'agit par conséquent pas d'un objet (comme dans le cas du *Sender* par exemple des gestionnaires d'événements), mais bien d'une classe.

Le mécanisme de cette méthode est simple : on affiche le nom de la classe en cours, on affecte la classe parent au paramètre et on boucle tant que cette classe existe, c'est-à-dire n'est pas égale à *nil*.

Voici un affichage obtenu par ce programme :



Vous constaterez entre autres que la classe *TForm* est à une profondeur de neuf héritages de *TObject* alors que la classe *TChien* est au second niveau (ce qui correspond aux définitions utilisées dans l'unité *animal.pas*). Comme affirmé plus haut, toutes ces classes proviennent *in fine* de *TObject*.

Voici le listing complet de cet exemple :

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  Buttons;

type

  { TMainForm }

  TMainForm = class(TForm)
    btnForm: TButton; // boutons pour les tests
    btnClear: TButton;
    btnMemo: TButton;
    btnButton: TButton;
    btnFontDialog: TButton;
    btnChien: TButton;
    mmoDisplay: TMemo; // memo pour l'affichage
    procedure btnButtonClick(Sender: TObject);
    procedure btnChienClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnFontDialogClick(Sender: TObject);
    procedure btnFormClick(Sender: TObject);
    procedure btnMemoClick(Sender: TObject);
  private
    { private declarations }
    procedure Display(AClass: TClass); // affichage
  public
    { public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

uses
```

```

    animal; // unité pour le traitement des animaux

{ TMainForm }

procedure TMainForm.btnFormClick(Sender: TObject);
// *** généalogie de TForm ***
begin
    Display(TForm);
end;

procedure TMainForm.btnMemoClick(Sender: TObject);
// *** généalogie de TMemo ***
begin
    Display(TMemo);
end;

procedure TMainForm.btnClearClick(Sender: TObject);
// *** effacement du mémo ***
begin
    mmoDisplay.Lines.Clear;
end;

procedure TMainForm.btnFontDialogClick(Sender: TObject);
// *** généalogie de TFontDialog ***
begin
    Display(TFontDialog);
end;

procedure TMainForm.btnButtonClick(Sender: TObject);
// *** généalogie de TButton ***
begin
    Display(TButton);
end;

procedure TMainForm.btnChienClick(Sender: TObject);
// *** généalogie de TChien ***
begin
    Display(TChien);
end;

procedure TMainForm.Display(AClass: TClass);
// *** reconstitution de la généalogie ***
begin
    repeat
        mmoDisplay.Lines.Add(AClass.ClassName); // classe en cours
        AClass := AClass.ClassParent; // on change de classe pour la classe parent
    until AClass = nil; // on boucle tant que la classe existe
    mmoDisplay.Lines.Add(""); // ligne vide pour séparation
end;

```

end.

VARIABLES DE CLASSE ET METHODES DE CLASSE STATIQUES

En ajoutant le mot réservé *static* à la fin de la déclaration d'une méthode classe, vous obtenez une méthode de classe statique. Une méthode de ce type se comporte comme une procédure ou une fonction ordinaire. Leur utilisation permet de les nommer avec le préfixe de la classe et non celui de l'unité où elles ont été définies : la lisibilité en est meilleure et les conflits de noms sont limités. Contrairement aux méthodes statiques ordinaires, les méthodes de classe statiques ne peuvent ni accéder à *Self* ni être déclarées virtuelles.

Les méthodes de classe statiques n'ont pas d'accès aux membres d'une instance : par exemple, si vous tentez à partir d'elles de faire appel à une méthode ordinaire ou d'accéder à un champ ordinaire, le compilateur déclenchera une erreur. En revanche, vous pouvez déclarer des *variables de classe* et des *méthodes de classe statiques* qui seront manipulables à volonté entre elles

[Exemple PO-10]

Le programme d'exemple proposé se contente de prendre une chaîne d'une zone d'édition, de la mettre en majuscules et de l'afficher dans un composant de type *TMemo*. Au lieu de faire appel à une variable, une procédure et une fonction simples, leurs équivalents variable et méthodes de classe statiques sont utilisés.

En voici le listing complet :

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls;

type
  { TMyClass }

  TMyClass = class
  private
    class var fMyValue: string; // variable de classe
  public
    class procedure SetMyValue(const AValue: string); static; // méthodes de
classe
    class function GetMyValue: string; static;
  end;
```

```

{ TMainForm }

TMainForm = class(TForm)
  btnClear: TButton; // nettoyage de l'affichage
  btnOK: TButton; // changement de valeur
  edtSetVar: TEdit; // entrée de la valeur
  mmoDisplay: TMemo; // affichage de la valeur
  procedure btnClearClick(Sender: TObject);
  procedure btnOKClick(Sender: TObject);
  procedure edtSetVarExit(Sender: TObject);
private
  { private declarations }
public
  { public declarations }
end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.btnClearClick(Sender: TObject);
// *** nettoyage de la zone d'affichage ***
begin
  mmoDisplay.Lines.Clear;
end;

procedure TMainForm.btnOKClick(Sender: TObject);
// *** affichage de la valeur ***
begin
  mmoDisplay.Lines.Add(TMyClass.GetMyValue); // on affiche
end;

procedure TMainForm.edtSetVarExit(Sender: TObject);
// *** nouvelle valeur ***
begin
  TMyClass.SetMyValue(edtSetVar.Text); // on affecte à la variable de classe
end;

{ TMyClass }

class procedure TMyClass.SetMyValue(const AValue: string);
// *** la valeur est mise à jour ***
begin

```

```
fMyValue := Upcase(AValue); // en majuscules
end;

class function TMyClass.GetMyValue: string;
// *** récupération de la valeur ***
begin
    Result := fMyValue;
end;

end.
```



Le mot réservé *class* doit être présent à la fois lors de la déclaration et au moment de la définition de la méthode.

METHODES DE MESSAGE

BILAN
