ALLER PLUS LOIN AVEC LAZARUS

Programmation avancée

21/05/2015

GVSOFT

Gilles Vasseur

# POO à gogo : la Programmation Orientée Objet

Dans ce chapitre, nous allons aborder certaines notions fondamentales pour exploiter au mieux la puissance de Free Pascal. Non seulement ce dernier est un héritier de la programmation structurée (regroupement du code en fonctions et procédures), mais il a été entièrement pensé pour manipuler au mieux des objets à travers la notion de classe. Sans imposer la Programmation Orientée Objet, Free Pascal (et plus encore Lazarus) invite fortement à souscrire à ses principes.

## Classes et objets

### La programmation orientée objet

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}, {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. On pourrait ainsi compléter la liste d’exemples par les trios suivants : {ordinateur, âge, 20}, {ordinateur, taille, moyen tour}.

Ces trios ne deviennent vraiment intéressants qu’avec des *méthodes* pour les manipuler : créer et supprimer une entité, modifier une valeur, ajouter un attribut, comparer des valeurs, lister les attributs d’une entité donnée, 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 attributs[[1]](#footnote-1).

En matière informatique, la programmation structurée regroupe les instructions au sein de structures appelées *fonctions* et *procédures*, mais les *données* en sont distinctes. La *Programmation Orientée Objet* (POO) se propose de représenter de manière plus rigoureuse et efficace les entités et leurs relations en les encapsulant au sein d’*objets*.

### Les classes

La réunion des attributs et des méthodes permettant leur manipulation dans une même structure est le fondement de la POO et prend le nom de *classe*. Dans un premier temps, vous pouvez considérer une classe comme un enregistrement qui possèderait 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 lorsqu’il s’agirait d’associer ces procédures et ces fonctions à un animal particulier. Vous pourriez par exemple créer un enregistrement représentant l’état de l’animal :

TEtatAnimal = record

FNom: string;

FAFaim: Boolean ;

FASoif: Boolean ;

end;

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. 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, les interactions entre les procédures et les fonctions aussi : une erreur pourra se glisser dans leur intrication et il sera difficile de l’y déceler.

Dans un tel cas de figure, la POO va d’emblée montrer son efficacité. Il suffira de déclarer une classe du type[[2]](#footnote-2) :

TUnAnimal = class

private

fNom: string;

fASoif: Boolean ;

fAFaim: Boolean ;

procedure SetNom;

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 attributs et des méthodes proposés par cette classe.

Cette déclaration se fait 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. Ensuite, il faudra bien sûr définir les comportements (que se passe-t-il dans le programme lorsqu’un animal mange ?) dans la partie *implementation* de l’unité : la seule différence entre la définition d’une méthode et celle d’une procédure (ou d’une fonction) traditionnelle est que son identificateur aura le nom de la classe comme préfixe suivi d’un point :

implementation

// […]

procedure TUnAnimal.Avancer ;

begin

ShowMessage(Nom + ‘ avance…’); // affiche le message adapté

end ;

Ainsi, si vous avez défini trois animaux grâce à trois variables du type TUnAnimal, pour agir sur un animal particulier, il suffira que vous précisiez en préfixe quel animal est concerné par la méthode ou l’attribut voulu :

Nemo.Avancer ;

Rantanplan.Dormir ;

Minette.ASoif := True ;

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 cause. Quand vous utilisez une classe de Lazarus aussi banale mais utile que TStringList, vous n’avez heureusement pas à étudier tout le code qui la compose ni à vérifier s’il a été modifié depuis sa dernière utilisation. Mieux : si son implémentation devait changer, vous ne vous en rendriez sans doute compte qu’à travers ses performances, l’extension de ses fonctionnalités ou la correction de bogues. Pour vous, TStringList resterait une classe dont les méthodes et les propriétés seraient accessibles de la même manière, y compris pour un code ancien.

La dernière 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.

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

### 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 une adresse physique vers lui.

Dans l’exemple précédent, Nemo, Rantanplan et Minette étaient trois variables, donc trois *instances*, pointant vers trois objets de type TUnAnimal (la classe). Autrement dit, une *classe* est un moule et les *objets* sont les entités que l’on obtient à partir de ce moule.

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.

### Construction et destruction

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 le nom traditionnel mais non obligatoire est *Create*, et un destructeur (*destructor*) qui répond au nom lui aussi arbitraire de *Destroy* .

Comme le monde virtuel est parfois aussi impitoyable que le monde réel, dans l’exemple concernant les animaux, vous donneriez naissance aux animaux et libéreriez la place en mémoire qu’ils occupaient quand vous auriez décidé de leur disparition.

Nemo := TUnAnimal.Create ; // création de l’objet

// Ici, le travail avec l’animal créé…

Nemo.Free ; // libération des ressources de l’objet (en interne, *Free* appelle *Destroy*)

Quand vous utilisez des composants dont le propriétaire[[3]](#footnote-3) est défini (ce qui est le cas dès que vous vous servez des fiches et que vous y déposez les composants nécessaires à votre application), c’est ce propriétaire qui est chargé d’allouer et de libérer la mémoire de manière transparente. Vous n’avez par conséquent pas à vous en occuper.

En revanche, lors d’une instanciation dynamique (c’est-à-dire au cours d’un programme, sans utiliser le système automatique évoqué ci-dessus), c’est à vous en tant que programmeur de tout gérer. L’instanciation d’une *TStringList* prendra alors cette forme :

var

MaStringList : TStringList ;

begin

MaStringList := TStringList.Create ; // on crée la liste de chaînes

try // on protège le code de manipulation pour être sûr de préserver les ressources

// ici le traitement voulu…

MaStringList.Add(‘coucou’) ;

finally

// en interne, la méthode *Free* appelle le destructeur *Destroy*

MaStringList.Free ; // les ressources seront toujours libérées

end ;

end ;

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 :

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

Le compilateur ne vous alerterait pas parce qu’il penserait que vous voulez exécuter la méthode *Create* de l’objet *MaStringList[[4]](#footnote-4)*. Le problème est que seules les instructions explicites de cette méthode seraient exécutées, sans le mécanisme de réservation et d’allocation de mémoire.

En revanche, à l’exécution une erreur de violation d’accès serait immédiatement déclenchée, car la mémoire nécessaire à l’objet n’aurait pas été allouée ! C’est à partir de la classe qu’on crée un objet et c’est à la variable dédiée à l’objet qu’on affecte le pointeur vers la structure allouée.

Si oublier de créer l’instance d’une classe et l’utiliser malgré tout aboutira au déclenchement d’une exception (violation d’accès), s’abstenir de libérer l’instance d’une classe produira des *fuites de mémoire*, le système interdisant à d’autres processus d’accéder à des portions de mémoire qu’il pense encore réservées.

Le lecteur attentif est aussi en droit de se demander pourquoi le destructeur à écrire est baptisé *Destroy* et que la méthode utilisée pour la destruction de l’objet est *Free*. En fait, *Free* vérifie que l’objet existe vraiment (c’est-à-dire qu’il pointe sur une adresse) avant d’appeler *Destroy*, évitant ainsi de lever de nouveau une exception pour violation d’accès.

## Principes fondamentaux

### Encapsulation

L’*encapsulation* est le concept fondamental de la POO. Il s’agit de protéger toutes les données au sein d’une classe en interdisant leur manipulation directe : seul l’accès à travers une méthode ou une propriété est autorisé.

Ainsi, aucun objet extérieur à une instance de la classe *TUnAnimal* n’aura la possibilité de connaître l’existence de *fFaim* et donc d’y accéder :

// Erreur : compilation refusée

MonObjet.AFaimAussi := MonAnimal.fAfaim ;

// OK si ATresSoif est une propiété booléenne modifiable de AutreObjet.

AutreObjet.ATresSoif := MonAnimal.AFaim ;

Une *méthode* est une procédure ou une fonction particulière utilisée au sein d’un objet. La façon de l’appeler diffère d’une procédure ou d’une fonction traditionnelle[[5]](#footnote-5) dans la mesure où elle doit à la moindre ambiguïté être préfixée du nom de l’objet qui la convoque, suivi d’un point.

Exemples :

MaListe.Add(‘un nouvel élément’) ;

Medor.Dormir ;

Les objets pouvant aussi contenir d’autres objets, le niveau d’imbrication dépasse fréquemment le premier niveau :

MaListe.Items[2].Text := ‘il fait beau’ ;

Une *propriété* est le moyen d’accéder à un attribut représenté dans la partie cachée de la classe par un *champ*.

Si vous reprenez l’interface de la classe *TUnAnimal* définie plus haut, vous pourriez à présent la commenter ainsi :

TUnAnimal = class // c’est bien une classe

private // indique que ce qui suit n’est pas visible à l’extérieur de la classe

fNom: string; // un attribut représenté par un champ de type chaîne

fASoif : Boolean ; // deux attributs représentés par deux champs booléens

fAFaim : Boolean ;

procedure SetNom;// détermine la valeur d’un attribut *via* une méthode

public // indique que ce qui suit est visible à l’extérieur de la classe

procedure Avancer ;

procedure Manger;

procedure Boire;

procedure Dormir;

// les propriétés permettent d’accéder aux attributs *via* des champs

// ou des méthodes

property ASoif : Boolean read fASoif write fASoif;

property AFaim : Boolean read fAFaim write SetAFaim;

property Nom: string read fNom write SetNom;

end ;

Les propriétés se servent des mots réservés *read* et *write* pour accéder aux *méthodes* et aux *champs*, c’est-à-dire à la représentation des attributs, à l’intérieur de l’interface de la classe.

Notez que *read* et *write* peuvent tous les deux renvoyer à une méthode ou à un champ et qu’un des deux au moins doit être présent. Dans ce dernier cas, c’est généralement *read* qui existe, indiquant que la propriété est en lecture seule.

D’autre part, le degré 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 toujours le niveau d’encapsulation le plus élevé possible pour chaque élément.

Souvenez-vous en effet 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, vous êtes assuré de perturber les programmes qui l’auront utilisée puisque vous n’avez *a priori* aucune idée du contexte de cette utilisation.

Remarquez que l’accessiblité 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, car sources de confusion et d’erreurs parfois difficiles à détecter et à corriger.

Généralement, une section *strict private* abrite des champs et des méthodes qui servent d’outils de base. Une section *private* permet à d’autres classes de la même unité de partager des informations : elle est très fréquente pour des raisons historiques (lorsque *strict private* n’existait pas). Les variantes de *protected* permettent surtout des redéfinitions de méthodes[[6]](#footnote-6). *public* est la portée par défaut, qui n’a pas besoin de se faire connaître puisqu’elle s’offre à la première sollicitation venue. *published* sera un outil précieux lors de l’intégration de composants dans la palette de Lazarus.

L’expérience est une aide précieuse ici comme dans de nombreux domaines pour faire les bons choix : l’erreur sera donc souvent formatrice, bien plus que l’immobilisme !

### Héritage

Jusqu’à présent, les classes vous ont sans doute semblé de simples enregistrements 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 la classe souche. 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.

Encore plus fort : cet *héritage* se propage de génération en génération, la nouvelle classe héritant de son ancêtre, de l’ancêtre de son ancêtre, la chaîne ne s’interrompant qu’à la classe souche.

Mais pourquoi vouloir hériter ? C’est que, par une analogie anthropomorphique, l’héritage renforce la puissance d’une classe ! Si vous manipuliez la classe *TUnAnimal*, vous pourriez avoir à travailler avec un ensemble de chiens et envisager alors de créer un descendant *TUnChien* aux propriétés et méthodes étendues.

En voici une définition possible :

TUnChien = class(TUnAnimal)

strict private

fBatard : Boolean ;

procedure SetBatard;

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 *TUnAnimal*. Les autres lignes ajoutent des fonctionnalités (*Aboyer* et *RemuerDeLaQueue*) ou définissent de nouvelles propriétés (*Batard*). Ce qui exprimera la puissance de l’héritage, c’est qu’un objet de type *TUnChien* disposera de plus de tout ce que sait faire un objet de type *TUnAnimal*.

Les lignes de code suivantes seront par conséquent compilées et exécutées sans souci :

Medor := TUnChien.Create ; // on crée le chien Medor

try // on protège la section

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 ancêtre la capacité de savoir exploiter Manger

finally // le code suivant sera toujours exécuté

Medor.Free ; // on libère la mémoire allouée

end ;

Il s’agit bien ici des fonctionnalités de la classe, donc de sa partie publique. La classe *TUnChien* n’aura accès qu’à ce qu’aura décidé son ancêtre *TUnAnimal*, en fonction de la portée de tel ou tel élément.

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

Vous pouvez aussi souhaiter modifier le comportement d’une méthode reçue en héritage. Pour remplacer celle héritée, il suffit qu’elle soit accessible à la classe enfant : soit elle est privée mais présente dans la même unité, soit elle est d’une visibilité supérieure et accessible y compris dans une autre unité.

Par exemple, en ce qui concerne la méthode *Manger* définie dans l’ancêtre *TUnAnimal*, 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 nouveau comportement :

TUnChien = class(TUnAnimal)

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;

La nouvelle méthode remplacera entièrement[[7]](#footnote-7) celle que contenait *TUnAnimal*.

### Polymorphisme

Avant de passer à la réalisation d’un projet mettant en œuvre ces notions, il vous faut en aborder une dernière : le *polymorphisme[[8]](#footnote-8)*. Les objets qui appartiennent à la même arborescence sont interchangeables : ils partagent au moins un ancêtre commun et peuvent par conséquent être affectés à une variable qui serait du type de cet ancêtre.

Ainsi, une affectation telle que celle qui suit est autorisée :

MonAnimal := MonChien ;

Cette affectation signifie que l’objet *MonAnimal* possède à présent toutes les propriétés de *MonChien*.

La réciproque n’est pas autorisée, car *TUnChien* étant plus riche que *TUnAnimal*, la valeur de certaines propriétés resterait indéterminée. Dans le cas présent, ce serait *Batard* qui poserait problème.

Le polymorphisme est une propriété très importante des objets : il va permettre un ensemble d’échanges de propriétés et de partages de méthodes.

## La basse-cour des Contes du Chat perché.

## Structure d’une classe

### Sections d’une classe

### Les champs

### Les méthodes

#### Méthodes abstraites

#### Méthodes virtuelles

#### Méthodes statiques

#### Méthodes de classes

### La variable Self

## La POO en action : des chiens animés

# RTL et LCL

# Les événements

## Les événements standards

## Les événements des objets créés dynamiquement

## Créer ses propres événements

# Les exceptions

## Le rôle des exceptions

## Mauvais usage des exceptions

# La tour de Babel : traduire son logiciel

Le programmeur qui souhaite adapter un logiciel à une autre langue peut penser qu’il suffit de traduire les termes affichés et de présenter cette traduction à l’utilisateur final pour venir à bout d’une tâche somme toute triviale. Cependant, ce programmeur naïf se heurtera rapidement à des problèmes qu’il n’imaginait pas !

Il aura certes prévu une série de messages et pensé à les regrouper 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 supportant ou non les accents. 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), 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 ;
* Le mois d’une date en anglais est donné avant le jour, contrairement au français, etc.

Dans le cadre de ce chapitre, afin de ne pas lui donner une ampleur démesurée, ne sera abordée que la simple traduction d’un point de vue purement technique en tant que programmeur : comment faire pour qu’un logiciel s’adapte au mieux à une autre langue ? Mais il ne faut pas oublier ce qui vient d’être écrit avant de se lancer dans l’internationalisation d’un travail !

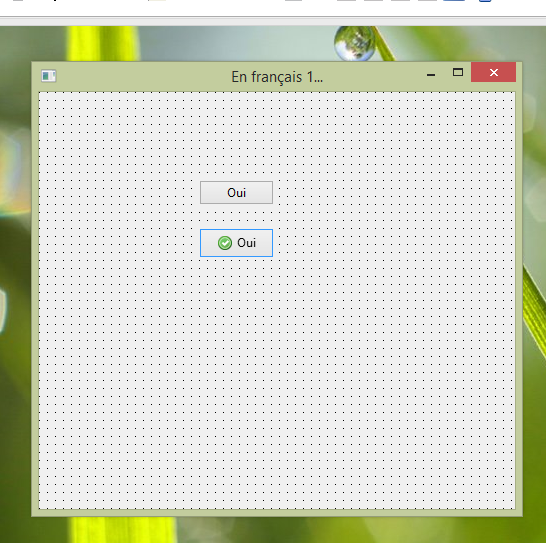
Quant à ceux qui seraient tentés d’ignorer ce chapitre en pensant limiter leur production à la langue française, ils sont invités à en lire au moins la première partie. En effet, Lazarus et Free Pascal sont des outils conçus en anglais pour un public anglophone. Les complications évoquées plus haut sont d’actualité dès lors qu’un projet envisage d’utiliser une autre langue que celle d’origine.

## Un programme français… en anglais

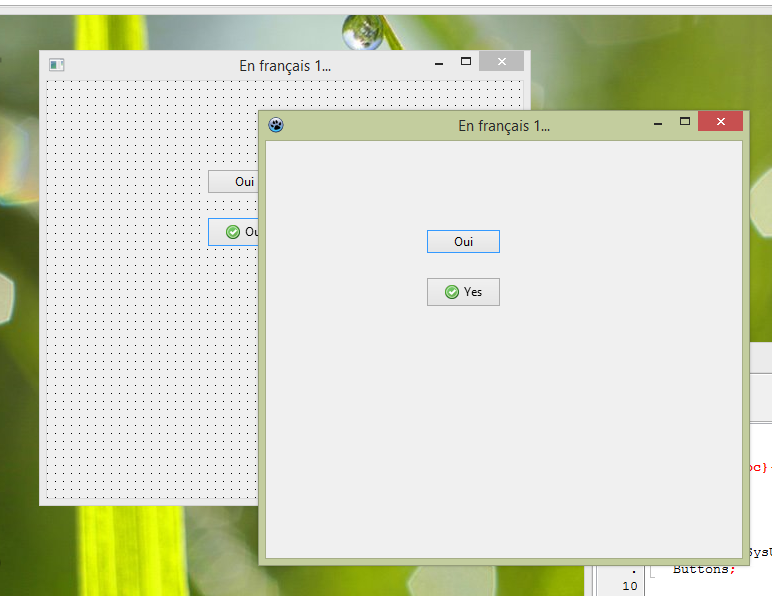
Les problèmes vont surgir dès la réalisation d’un programme aussi élémentaire que celui-ci :

* 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 :



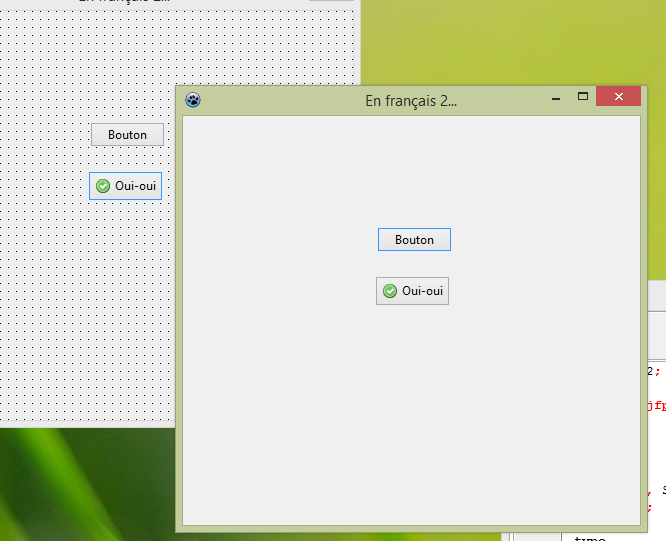
Vous serez sans doute tenté de croire que la transformation du **Oui** en **Yes** pour le composant *TBitBtn* est un bogue de Lazarus puisque le composant *TButton* ne semble pas souffrir de la même tare. Cependant, avant de vous ruer sur la rubrique *Bugtracker* du site de Lazarus, il est de nouveau 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

Une première manière de contourner le problème rencontré est 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 :

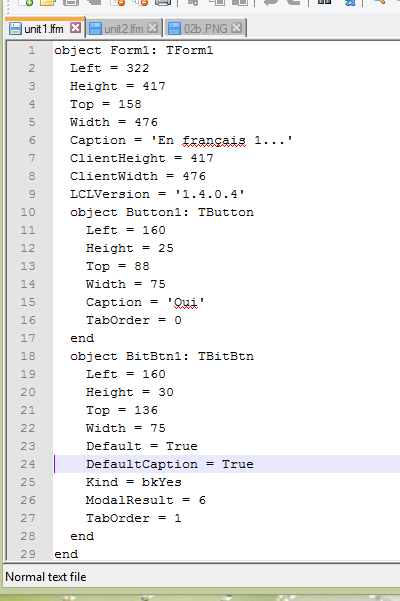
* Modifiez la valeur de la légende (*Caption*) en la passant de ***&Oui*** à ***&Oui-oui*** ;
* Compilez le programme ;
* Lancez son exécution.

Vous obtiendrez cet écran :



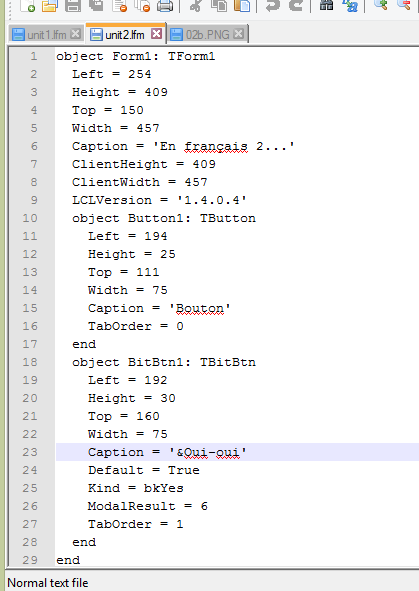
Que s’est-il passé ? Pour le comprendre, il faut examiner les fichiers **LFM** qui contiennent la description des fiches. Comme ce sont de simples fichiers textes, l’utilisation d’un outil tel **Notepad++** est tout à fait adaptée.

Dans la première version du programme, on lit ceci :



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 :



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.

Avec le composant *TBitBtn*, on aurait obtenu le même affichage en changeant la valeur de *DefaultCaption* de ***True*** à ***False***. C’est une seconde solution qui dépend des possibilités qu’offre le composant en action : cette propriété n’est en effet présente que pour les descendants de *TCustomBitBtn* !

Mais revenons à notre question initiale : que s’est-il passé ? Par défaut, afin d’éviter d’encombrer le fichier **LFM** de données inutiles, **l’EDI n’enregistre pas les valeurs des propriétés lorsqu’elles sont identiques à leur valeur par défaut**.

En modifiant le libellé manuellement, vous aurez 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 aurez 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.

Il faut surtout se souvenir 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.

### Une solution plus générale

S’il n’était question que de propriétés accessibles en écriture, vous pourriez vous satisfaire des deux premières solutions. Malheureusement, de nombreux messages ne sont pas de ce type : certaines propriétés (comme le nom des couleurs) et la plupart des messages d’erreur ne sont pas disponibles directement.

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

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 *project3.lpr* ;
* Ouvrez depuis le navigateur le répertoire utilisé ;
* Créez un sous-répertoire que vous baptiserez *languages* (ce nom a son importance !) ;
* Déplacez le fichier *projet3.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) ;
* Copiez le fichier *lclstrconsts.fr.po* depuis son répertoire d’origine jusqu’au répertoire *languages* que vous venez de créer (l’emplacement de *lclstrconsts.fr.po* est le sous-répertoire *lcl/languages* du répertoire d’installation de Lazarus) ;
* Éditez le fichier du projet *project3.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 **D*efaultTranslator*** 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, unit3

{ 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.

### Intérêt 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 susceptibles de modifier l’affichage ;
* L’utilisation du système automatique intégré de Lazarus *via* l’unité ***DefaultTranslator***.

Si les deux premières sont légères, la dernière est de loin celle recommandée : elle fonctionne en effet pour tous les messages des unités du projet et elle est automatique. 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.

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];

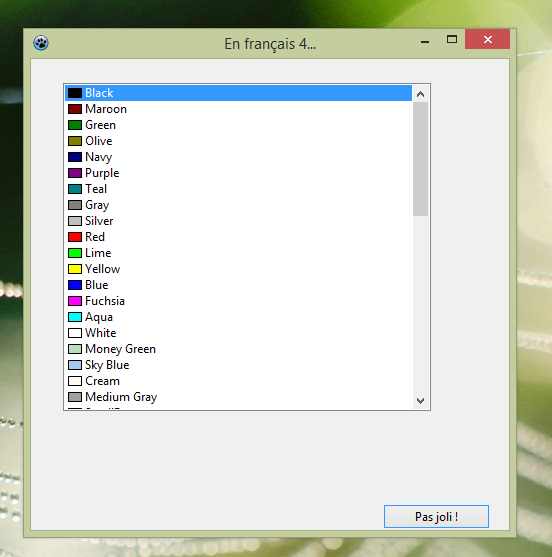
Button1.Caption := RS\_NotPretty;

end;

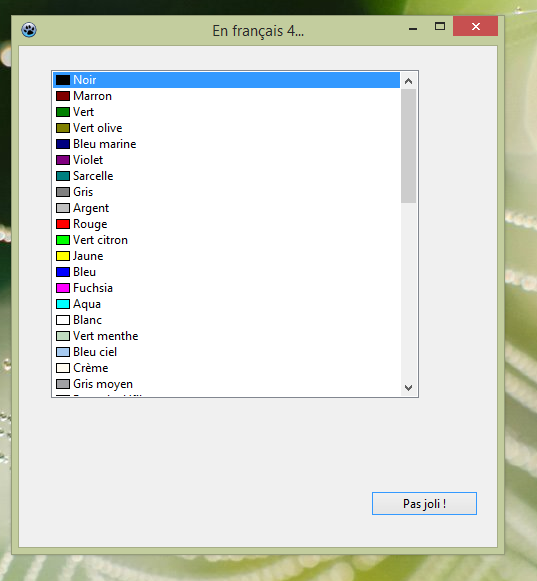
end;

La procédure introduite permet de modifier l’affichage du bouton en fonction de la propriété Style de 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 mêmes fichiers **PO** dans un sous-répertoire *languages* du répertoire de l’application, les noms de couleurs seront traduits :



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 au mécanisme automatique mis en place.

Par la même occasion, les messages d’exception le sont aussi. Pour vous en assurer, 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);

// 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 générale

Comme le monde de l’informatique est étranger à la magie, la traduction miraculeuse 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) : *projet3.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 avions 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 quatrième solution

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 pas besoin d’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 project3;

{$mode objfpc}{$H+}

uses

{$IFDEF UNIX}{$IFDEF UseCThreads}

cthreads,

{$ENDIF}{$ENDIF}

Interfaces, // this includes the LCL widgetset

Forms, unit3

{ you can add units after this },

sysutils, // une unite 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*** et non en dur, ce qui autorise 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

### Préparation du programme souche

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 pas saisies en « dur », c’est-à-dire qu’elles sont isolées dans une section particulière (*resourcestring*) qui indique que ces chaînes sont des ressources et feront l’objet d’un stockage particulier. Sans ce stockage particulier, les traductions ne s’effectueront pas : les libellés des constantes de ressources servent 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 *project5.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 entièrement en anglais. L’objectif va justement être de 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 un éditeur comme **Notepad++**, 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...**

**unit2.lrt** est évidemment construit selon le même modèle :

**TFORM2.CAPTION=New form**

**TFORM2.BITBTN1.HINT=Close the form**

De son côté, le contenu de **project5.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 !"**

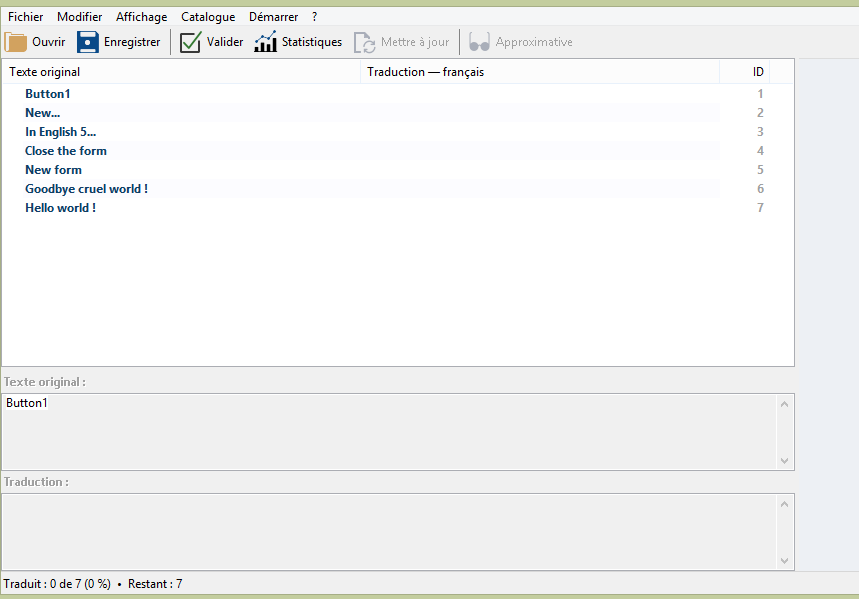
**msgstr ""**

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. Quant à la première ligne de ces triplets, elle correspond à un repère dans le code source ou dans le fichier **LFM**.

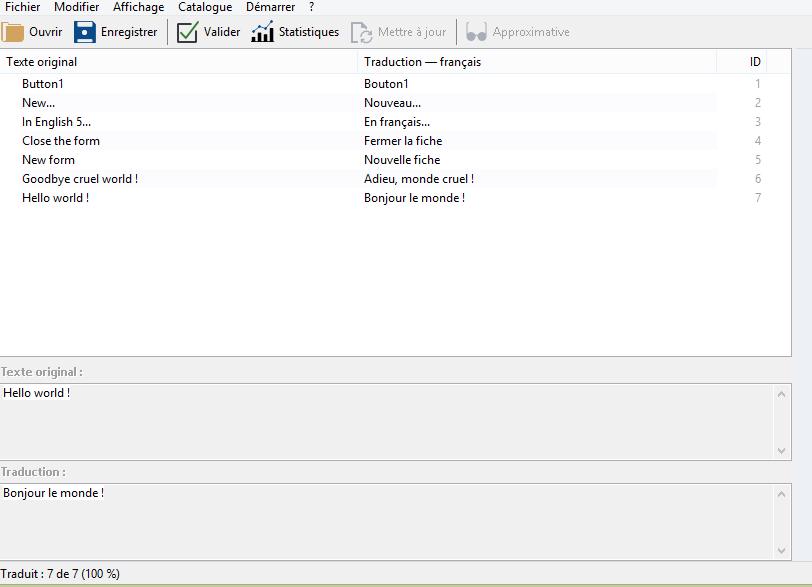
Même si la traduction peut se faire manuellement, l’utilisation d’outils spécialisés dans le traitement des fichiers **PO** est vivement recommandée : non seulement ils éviteront bien des erreurs, mais ils fournissent aussi des outils d’édition et souvent des propositions de traduction qui s’appuient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

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

Avant de traduire, le fichier souche doit être préservé : faites-en une copie dans le répertoire *languages* et rebaptisez-le *project5.fr.po*. L’infixe *fr* est celui qui indique qu’il s’agit de la traduction française : sans lui, il faudra préciser la langue de traduction. **poEdit** reconnaît immédiatement cet infixe et présente le fichier sous cette forme :



Proposez la traduction suivante :



Après avoir enregistré votre travail de traduction, vous pouvez éditer le fichier modifié avec un outil comme **NotePad++** :

**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 complète

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 « dure » : 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 project5;

{$mode objfpc}{$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.

## De l’anglais au choix de la langue

# Persistence des données : fichiers INI, XML et JSON

### De vieux amis : les fichiers INI

### Le XML pour tous

### Un avatar prometteur : JSON

# Les paquets de Lazarus

## Gestion des paquets

## Créer un paquet

## Problèmes courants avec les paquets

# De nouveaux composants

## Étendre la palette des composants

### Modifier un composant

### Hériter d’un composant

### Créer un composant

## Tester un composant

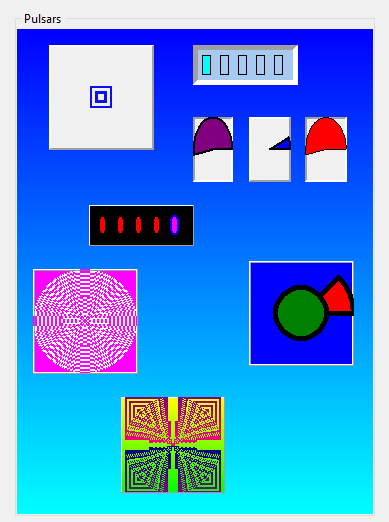
## Intégrer un composant à la palette

## Exemples

### Un composant graphique dérivé de TGraphicControl : TGVGradient



### Une série de composants : TGVPulse et ses descendants



### Un composant entièrement nouveau : TGVSizeMover



# Des classes génériques

# Les enregistrements étendus

# Class Helpers

# Bibliothèques tierces

## BGRABITMAP : Le graphisme amélioré

# Google APIs

# Internet

# SGBD

# L’impression

# Techniques de programmation

## La récursivité

#### Factorielles

#### Le jeu de la tour de hanoi

#### Solution non récursive

#### Solution récursive

## Les tris

Table des matières

[POO à gogo : la Programmation Orientée Objet 1](#_Toc419831305)

[Classes et objets 1](#_Toc419831306)

[La programmation orientée objet 1](#_Toc419831307)

[Les classes 1](#_Toc419831308)

[Les objets : construction et destruction 3](#_Toc419831309)

[Principes fondamentaux 4](#_Toc419831310)

[Encapsulation 4](#_Toc419831311)

[Héritage 5](#_Toc419831312)

[Polymorphisme 5](#_Toc419831313)

[Structure d’une classe 5](#_Toc419831314)

[Sections d’une classe 5](#_Toc419831315)

[Les champs 5](#_Toc419831316)

[Les méthodes 5](#_Toc419831317)

[La variable Self 6](#_Toc419831318)

[La POO en action : des chiens animés 6](#_Toc419831319)

[RTL et LCL 7](#_Toc419831320)

[Les événements 8](#_Toc419831321)

[Les événements standards 8](#_Toc419831322)

[Les événements des objets créés dynamiquement 8](#_Toc419831323)

[Créer ses propres événements 8](#_Toc419831324)

[Les exceptions 9](#_Toc419831325)

[Le rôle des exceptions 9](#_Toc419831326)

[Mauvais usage des exceptions 9](#_Toc419831327)

[La tour de Babel : traduire son logiciel 10](#_Toc419831328)

[Un programme français… en anglais 10](#_Toc419831329)

[Un peu de bricolage pour traduire 12](#_Toc419831330)

[Une solution plus générale 16](#_Toc419831331)

[Intérêt de ne pas bricoler la traduction 17](#_Toc419831332)

[Fonctionnement de la solution générale 21](#_Toc419831333)

[Une quatrième solution 22](#_Toc419831334)

[De l’anglais au français 23](#_Toc419831335)

[Préparation du programme souche 23](#_Toc419831336)

[fichiers LRT et PO 25](#_Toc419831337)

[Traduction automatique complète 30](#_Toc419831338)

[De l’anglais au choix de la langue 30](#_Toc419831339)

[Persistence des données : fichiers INI, XML et JSON 31](#_Toc419831340)

[De vieux amis : les fichiers INI 31](#_Toc419831341)

[Le XML pour tous 31](#_Toc419831342)

[Un avatar prometteur : JSON 31](#_Toc419831343)

[Les paquets de Lazarus 32](#_Toc419831344)

[Gestion des paquets 32](#_Toc419831345)

[Créer un paquet 32](#_Toc419831346)

[Problèmes courants avec les paquets 32](#_Toc419831347)

[De nouveaux composants 33](#_Toc419831348)

[Étendre la palette des composants 33](#_Toc419831349)

[Modifier un composant 33](#_Toc419831350)

[Hériter d’un composant 33](#_Toc419831351)

[Créer un composant 33](#_Toc419831352)

[Tester un composant 33](#_Toc419831353)

[Intégrer un composant à la palette 33](#_Toc419831354)

[Exemples 33](#_Toc419831355)

[Un composant graphique dérivé de TGraphicControl : TGVGradient 33](#_Toc419831356)

[Une série de composants : TGVPulse et ses descendants 34](#_Toc419831357)

[Un composant entièrement nouveau : TGVSizeMover 35](#_Toc419831358)

[Des classes génériques 37](#_Toc419831359)

[Les enregistrements étendus 38](#_Toc419831360)

[Class Helpers 39](#_Toc419831361)

[Le graphisme amélioré 40](#_Toc419831362)

[Google APIs 41](#_Toc419831363)

[Internet 42](#_Toc419831364)

[SGBD 43](#_Toc419831365)

[L’impression 44](#_Toc419831366)

1. *L’orienté objet* – Bersini Hugues – Eyrolles 2007 [↑](#footnote-ref-1)
2. Ne vous inquiétez pas si vous ne maîtrisez pas le contenu de cette structure : son étude se fera bientôt. [↑](#footnote-ref-2)
3. Cette notion est expliquée p. XXX. [↑](#footnote-ref-3)
4. Rien ne vous interdit d’appeler autant de fois que vous le désirez la méthode *Create*, même s’il est rare d’avoir à le faire. Pour rappel, son nom est arbitraire. [↑](#footnote-ref-4)
5. On parle de *routines* dans ce cas. [↑](#footnote-ref-5)
6. Voir ci-après : l’héritage. [↑](#footnote-ref-6)
7. On verra plus loin qu’il est possible d’hériter de l’ancienne méthode et d’en modifier simplement le comportement. [↑](#footnote-ref-7)
8. L’étymologie en est grecque : *poly* signifie *plusieurs* et *morphie* signifie *forme*. [↑](#footnote-ref-8)