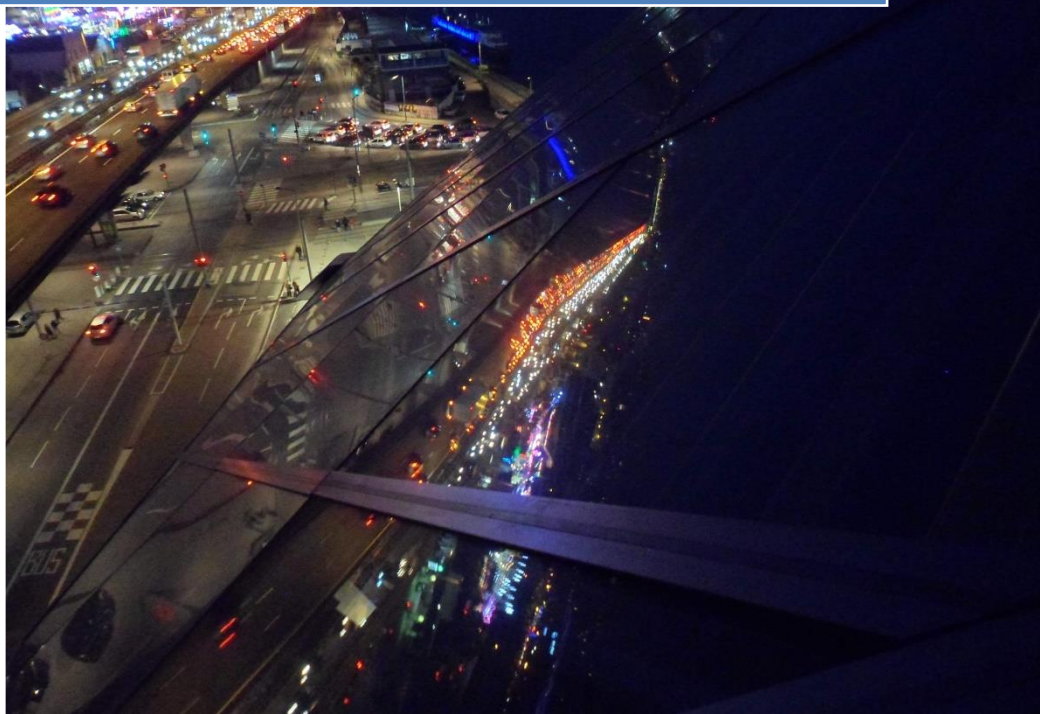


2015

Plus loin avec Free Pascal et Lazarus



Roland Chastain – Gilles Vasseur

07/06/2015

TABLE DES MATIERES

Expressions régulières.....	5
Introduction	5
Définition des expressions régulières.....	5
La fonction ExecRegExpr() de l'unité RegExpr	6
Les caractères symbolisant le début et la fin de la chaîne.....	6
Échappement des caractères spéciaux.....	6
Cases de l'échiquier	7
Comparaison avec le type ensemble du langage Pascal	7
Classes de caractères	8
Numéros de téléphone.....	9
Classes prédéfinies.....	9
Indication du nombre de caractères.....	9
Caractères éventuels	10
Alternative.....	11
Dates	12
Petite révision.....	12
La classe TRegExpr	12
La fonction ReplaceRegExpr()	13
Remplacement complexe par substitution	14
Remplacement complexe par fonction.....	14
Validation d'une chaîne FEN.....	16
Brève présentation de la notation Forsyth-Edwards	16
La fonction SplitRegExpr()	16
Expressions utilisées.....	17
Remplacement complexe	17
Exemple complet	18
Deux fonctions utiles	20
La fonction QuoteRegExprMetaChars()	20
La fonction RegExprSubExpressions().....	21
Modificateurs	21
Conclusion.....	23
La tour de Babel : traduire une application.....	24
<i>What's the matter ?</i>	24
Un programme français... en anglais.....	25
Un peu de bricolage pour traduire.....	27
Une solution plus générale	30

Intérêt de ne pas bricoler la traduction	31
Fonctionnement de la solution générale	35
Une quatrième solution	36
De l'anglais au français.....	37
Préparation du programme souche.....	37
Fichiers LRT et PO.....	39
Traduction automatique complète.....	43
Encore plus loin : de l'anglais au choix de la langue	44
Générer la liste des langues et choisir la langue.....	44
La mémorisation du choix et le redémarrage de l'application	46
Bilan	49
P00 à gogo : la Programmation Orientée Objet.....	50
Classes et objets.....	50
La programmation orientée objet	50
Classes.....	51
Champs, méthodes et propriétés	52
Les objets	55
Constructeur.....	57
Destructeur	59
Premiers gains de la P00	60
Principes et techniques de la P00.....	60
Encapsulation.....	60
Notion de portée	61
Héritage.....	62
Notion de polymorphisme	65
Les opérateurs IS et AS.....	67
Bilan	69
P00 à gogo : Les méthodes	70
Ce qu'il faut savoir.....	70
Méthodes statiques.....	70
Méthodes virtuelles	72
Compléments sur <i>inherited</i>	75
... et ce qu'il est utile de savoir	77
Méthodes abstraites.....	77
Méthodes de classe	78
Méthodes statiques de classe	84
Méthodes de messages.....	88
Surcharge de méthodes	91

Bilan	94
P00 à gogo : Les propriétés.....	95
Qu'est-ce qu'une propriété ?.....	95
Travailler avec les propriétés	95
Lecture et écriture d'une propriété : <i>getter</i> et <i>setter</i>	95
Propriétés et variables	105
Redéfinition d'une propriété	106
Propriété par défaut (default nodefault)	106
Les informations de stockage.....	106
Les propriétés indexées.....	106
Les tableaux de propriétés	109
Propriétés de classe.....	109
Exemple d'utilisation des méthodes et propriétés de classe	113
Bilan	120
Les exceptions	121
Travailler avec les exceptions	121
Les types d'exception.....	121
Déclencher une exception.....	121
Redéclencher une exception	121
Try... except	121
Try... finally	121
Exceptions avec modération	121
Bilan	121
Free Pascal et la P00	122
Au cœur de Free Pascal : RTL, LCL et FCL	122
Travailler avec la P00	122
Un ancêtre vénérable : TObject	122
Le grand oublié : TPersistent.....	122
Un prince reconnu : TComponent	122
L'omniprésente : TForm.....	122
Parent et propriétaire.....	122
Free et freeandnil	123
Les événements.....	123
Bilan	123
Les paquets.....	124
Les composants de Lazarus	125
Concevoir ses propres composants.....	126

EXPRESSIONS REGULIERES

Objectifs : dans ce chapitre, vous apprendrez à utiliser et à créer des expressions régulières pour la manipulation des chaînes.

Sommaire : Introduction – Cases de l'échiquier – Numéros de téléphone – Dates – Validation d'une chaîne FEN – Deux fonctions utiles

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

INTRODUCTION

Les expressions régulières sont un outil puissant pour la détection et la manipulation de chaînes de caractères. Vous allez découvrir la syntaxe des expressions régulières et apprendre à utiliser les fonctions de l'unité *RegExpr* livrée avec le compilateur **Free Pascal**.

Avant de commencer, faites un premier essai de compilation. Voici un programme qui affiche le numéro de version de l'unité *RegExpr* :

```
uses
  sysutils, regexpr;

const
  S = 'Expressions r'#130'guli'#138'res version %d.%d';

begin
  writeln(Format(S, [TRegExpr.VersionMajor(), TRegExpr.VersionMinor()]));
```

Vous obtenez le résultat suivant :

```
Expressions régulières version 0.952
```

La documentation et les exemples originellement joints à l'unité n'ont pas été inclus dans le paquetage de **Lazarus**. Il est toutefois possible de les télécharger sur le site personnel de l'auteur, Andrey V. Sorokin :

<http://regexstudio.com/TRegExpr/TRegExpr.html>

La syntaxe supportée par l'unité *RegExpr* est un sous-ensemble des expressions régulières de Perl.

DEFINITION DES EXPRESSIONS REGULIERES

On appelle *expression régulière* une chaîne de caractères qui représente un ensemble de chaînes de caractères.

Ainsi, la chaîne 'abc', considérée comme expression régulière, représente un ensemble contenant un seul élément, la chaîne 'abc'.

LA FONCTION EXECREGEXPR() DE L'UNITE REGEXPR

La fonction *ExecRegExpr()* de l'unité *RegExpr* permet de savoir si une chaîne donnée appartient à l'ensemble représenté par une expression. Le premier argument que reçoit la fonction est l'expression régulière elle-même, le second est la chaîne à examiner :

```
uses
  regexpr;

begin
  writeln(ExecRegExpr('abc', 'abc')); // TRUE
```

La chaîne 'abc' appartient bien à l'ensemble représenté par l'expression régulière 'abc'.

Pour être plus exact, *ExecRegExpr()* permet de savoir si dans une chaîne donnée se trouve une chaîne appartenant à l'ensemble représenté par l'expression régulière :

```
writeln(ExecRegExpr('abc', 'abcd')); // TRUE
```

Ce résultat signifie qu'une partie de la chaîne 'abcd' appartient à l'ensemble représenté par l'expression régulière 'abc'.

LES CARACTERES SYMBOLISANT LE DEBUT ET LA FIN DE LA CHAINE

Si à présent vous vouliez vous assurer que la chaîne entière, et non pas seulement l'une de ses parties, appartient à l'ensemble représenté par l'expression régulière, il suffirait d'ajouter au début et à la fin de l'expression deux caractères qui signifient respectivement « début de la chaîne » et « fin de la chaîne » :

```
writeln(ExecRegExpr('^abc$', 'abc')); // TRUE
writeln(ExecRegExpr('^abc$', 'abcd')); // FALSE
```

Le deuxième appel renvoie le résultat *FALSE*, car la chaîne 'abcd' ne se termine pas par 'abc'.

ÉCHAPPEMENT DES CARACTERES SPECIAUX

Les caractères '^' et '\$' ont donc une signification spéciale. On les appelle *métacaractères*.

Supposons qu'on veuille détecter ces deux caractères sans tenir compte de la signification particulière qui est la leur dans le langage des expressions régulières. Comment, par exemple, représenter l'ensemble contenant comme seul élément la chaîne `'^abc$'` ?

Pour cela, il faut adjoindre aux caractères spéciaux `'^'` et `'$'` un autre caractère spécial signifiant que le caractère qui le suit est à interpréter de façon littérale. C'est ce qu'on appelle, d'après l'anglais, « échapper » les caractères spéciaux.

Dans la syntaxe des expressions régulières, il s'agit du caractère `'\''` :

```
writeln(ExecRegExpr('\^abc$', '^abc$')); // TRUE
```

Le caractère `'\''` peut d'ailleurs servir à « s'échapper » lui-même :

```
writeln(ExecRegExpr('\\\\', '\\')); // TRUE
```

CASES DE L'ECHIQUIER

Il est temps de découvrir le vrai pouvoir des expressions régulières. Vous allez maintenant composer et utiliser des expressions représentant des ensembles contenant plus d'un élément.

COMPARAISON AVEC LE TYPE ENSEMBLE DU LANGUAGE PASCAL

Supposons par exemple que vous vouliez savoir si une chaîne donnée est le nom d'une case de l'échiquier, c'est-à-dire une chaîne de deux caractères, dont le premier est une lettre comprise entre `'a'` et `'h'`, et le second un chiffre compris entre `'1'` et `'8'`. En outre, supposons que vous soyez disposé à accepter les majuscules aussi bien que les minuscules.

Certes, il serait possible de résoudre ce problème en pur Pascal, sans passer par les expressions régulières. En effet, vous pouvez définir en Pascal des ensembles de caractères, et vérifier ensuite qu'un caractère donné appartient à l'un de ces ensembles :

```
{ $B- }
function IsChessSquare(aStr: string): Boolean;
const
  E1: set of char = ['a'..'h'];
  E2: set of char = ['1'..'8'];
begin
  Result := (Length(aStr) = 2)
    and (aStr[1] in E1)
    and (aStr[2] in E2);
end;

var
  s: string;
begin
  s := 'e2';
  writeln(IsChessSquare(s)); // TRUE
```




La directive `{B-}` signifie que l'évaluation des expressions booléennes doit s'arrêter dès que le résultat est connu. En l'occurrence, si la chaîne passée en paramètre n'était pas longue de deux caractères, l'évaluation devrait s'arrêter, de façon à éviter le risque d'une tentative d'accès à un emplacement inexistant dans la chaîne. Pratiquement, la directive `{B-}` n'est pas nécessaire, car elle ne fait que confirmer le mode d'évaluation que **Free Pascal** utilise par défaut.

CLASSES DE CARACTERES

Voici l'expression régulière, ou plutôt *une* expression régulière, qui représente l'ensemble des cases de l'échiquier :

```
'[a-h][1-8]'
```

Soit dit en passant, cette expression aurait pu aussi s'écrire de la façon suivante :

```
'[abcdefgh][12345678]'
```

Dans un cas comme dans l'autre, deux classes de caractères sont définies, semblables aux deux ensembles de notre fonction en pur Pascal, et l'on demande un caractère de l'une et un caractère de l'autre. Le nombre un est implicite : c'est la quantité par défaut.

Pour finir, ajoutez à votre expression les caractères signifiant « début et fin de la chaîne » :

```
'^[a-h][1-8]$'
```

Voici la nouvelle version de la fonction `IsChessSquare()` :

```
uses
  regexpr;
function IsChessSquareRE(aStr: string): boolean;
const
  EXPR = '^[a-h][1-8]$';
begin
  Result := ExecRegExpr(EXPR, aStr);
end;
```

Vous commencez à entrevoir l'un des atouts de ce langage que sont les expressions régulières : la brièveté.

NUMEROS DE TELEPHONE

CLASSES PREDEFINIES

Imaginez que vous vouliez savoir si une chaîne de caractères donnée contient un numéro de téléphone, soit une suite de 10 chiffres. Il y a justement une expression régulière pour dire cela : « une suite de dix chiffres ». À vrai dire, il y en a même plusieurs. Vous pourriez d'abord l'écrire ainsi :

```
'[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]'
```

Heureusement, il y a une classe prédéfinie qui correspond exactement à la classe '[0-9]'. On la note '\d'. Votre expression devient donc :

```
'\d\d\d\d\d\d\d\d\d\d'
```

Le caractère '\', qui tout à l'heure servait à forcer l'interprétation littérale d'un caractère spécial, sert ici au contraire à donner une valeur spéciale au caractère 'd', la valeur « chiffre » (*digit* en anglais).

INDICATION DU NOMBRE DE CARACTERES

La chaîne que vous voulez détecter étant composée de dix caractères de la même classe, vous pouvez présenter votre expression régulière de la façon suivante :

```
'\d{10}'
```

Et vous l'utiliserez ainsi :

```
uses
  regexpr;
begin
  writeln(ExecRegExpr('\d{10}', 'maison 0123456789')); // TRUE
```

La fonction *ExecRegExpr()* renvoie *TRUE*, car la chaîne passée comme second argument contient une suite de dix chiffres.

Pour mémoire, voici un tableau contenant l'ensemble des indicateurs de quantité ou *itérateurs* :

Itérateur	Signification	Équivalent
*	Zéro ou plus (gourmand)	{0, }
+	Un ou plus (gourmand)	{1, }
?	Zéro ou un (gourmand)	
{n}	Exactement n fois (gourmand)	
{n, }	Au moins n fois (gourmand)	

{n,m}	Au moins n fois mais pas plus de m fois (gourmand)	
*?	Zéro ou plus (non gourmand)	{0,}?
+?	Un ou plus (non gourmand)	{1,}?
??	Zéro ou un (non gourmand)	{0,1}?
{n}?	Exactement n fois (non gourmand)	
{n,}?	Au moins n fois (non gourmand)	
{n,m}?	Au moins n fois mais pas plus de m fois (non gourmand)	

CARACTERES EVENTUELS

Tout cela est bel et bon, direz-vous, mais quelquefois les numéros de téléphone contiennent des espaces. Et quelquefois, ils n'en contiennent pas.

Heureusement, il y a une expression régulière pour résoudre ce problème :

```
'\s?'
```

L'expression '\s' représente une classe de caractères, la classe « espace ». Elle est équivalente à l'expression suivante :

```
'[\t\n\r\f]'
```

Cette classe contient donc les caractères espace, tabulation, nouvelle ligne, retour chariot, nouvelle page. Soit dit en passant, au lieu de mettre un espace dans l'expression, vous auriez pu utiliser la notation hexadécimale :

```
'\x20'
```

Voici un tableau récapitulant ce que vous venez d'apprendre :

Caractère	Notation avec échappement	Notation hexadécimale
Tabulation	\t	\x09
Nouvelle ligne	\n	\x0a
Retour chariot	\r	\x0d
Nouvelle page	\f	\x0c

Les trois caractères '\s?' signifient donc, dans le langage des expressions régulières : « un espace éventuel », espace étant entendu au sens de la classe de caractères que nous venons de décrire.

Désormais votre fonction est capable de détecter aussi bien les numéros avec espaces que les numéros sans espaces :

```
const
EXPRESSION = '\d{2}\s?\d{2}\s?\d{2}\s?\d{2}\s?\d{2}';
```

begin

```
writeln(ExecRegExpr(EXPRESSION, 'maison 01 23 45 67 89')); // TRUE  
writeln(ExecRegExpr(EXPRESSION, 'maison 0123456789')); // TRUE
```

Au cas où vous ne voudriez admettre que les espaces au sens strict, il faudrait utiliser l'expression suivante :

```
'\d{2}?\d{2}?\d{2}?\d{2}?\d{2}';
```

Pour mémoire, voici un tableau récapitulant les classes prédéfinies :

Classe	Notation
Caractère alphanumérique (inclut le caractère '_')	\w
Caractère non alphanumérique	\W
Caractère numérique	\d
Caractère non numérique	\D
Espace (équivalent de la classe '[\t\n\r\f]')	\s
Caractère qui n'est pas un espace	\S

ALTERNATIVE

Très bien, très bien, sauf qu'avec ce système, même une chaîne espacée de façon irrégulière sera validée :

```
'maison 01 23456789'
```

Supposez qu'on ne veuille accepter que les chaînes correctement formatées, tout en conservant la possibilité de choisir entre le format avec espace et le format sans espace. Comment procéder ?

Il y a une expression régulière pour dire cela : « ou une suite de dix chiffres non espacés, ou des chiffres groupés par deux » :

```
'(\d{10}|\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2})'
```

C'est-à-dire :

```
'(' + '\d{10}' + '|' + '\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2}' + ')'
```

Le caractère '|' veut dire « ou ». Les parenthèses indiquent où se terminent les expressions alternatives.

DATES

PETITE REVISION

Intéressons-nous à la chaîne que renvoie la fonction `DateToStr()` de l'unité *SysUtils*. En France, le format de cette chaîne est le suivant :

```
'00/00/0000'
```

Vous savez déjà tout ce qu'il faut savoir pour trouver l'expression régulière représentant l'ensemble de toutes les chaînes respectant ce format. Voici cette expression :

```
'\d\d\d\d\d\d\d\d'
```

Mais pas si vite ! Vous pouvez être un peu plus précis :

```
'[0-3]\d/[01]\d/[12]\d\d\d'
```

Le lecteur du troisième millénaire aura une modification à faire. Laquelle ?

Bien entendu, il faudra qu'il remplace '[12]' par '[1-3]'.

Voici le code complet de l'exemple :

```
uses
  regexpr, sysutils;

const
  EXPRESSION = '[0-3]\d/[01]\d/[12]\d\d\d';

var
  s: string;

begin
  s := DateToStr(Now);
  writeln(ExecRegExpr(EXPRESSION, s)); // TRUE
end.
```

LA CLASSE TREGEXPR

Savoir qu'une chaîne donnée contient une date ou est une date, c'est fort bien, mais supposez que vous vouliez extraire cette date. Comment faire ?

Pour cela, la fonction `ExecRegExpr()` ne vous suffit plus. Vous devez faire connaissance avec la classe *TRegExpr*. Voici un exemple d'utilisation de cette classe :

```
const
  EXPRESSION = '[0-3]\d/[01]\d/[12]\d\d\d';
var
```

```

s: string;
e: TRegExpr;

begin
s := DateToStr(Now);

e := TRegExpr.Create;
e.Expression := EXPRESSION;
e.Exec(s);

writeln(e.Match[0]);           // 09/06/2015
writeln(e.MatchPos[0]);        // 1
writeln(e.MatchLen[0]);         // 10

e.Free;
end.

```

Comme vous le voyez, les variables *Match*, *MatchPos* et *MatchLen* sont des tableaux, dont seul le premier élément vous intéresse pour le moment. Vous apprendrez plus loin à quoi servent les autres éléments de ces tableaux.

La variable *Match[0]* contient la chaîne correspondant au motif. En l'occurrence, c'est tout simplement la valeur de la chaîne s, c'est-à-dire la valeur de retour de la fonction *DateToStr()*, qui par définition est une date.

La valeur de la variable *MatchPos[0]* est 1 parce que la chaîne correspondant au motif a été trouvée à partir du premier caractère de la chaîne s.

Enfin, la valeur de la variable *MatchLen[0]* est 10, soit la longueur de la chaîne correspondant au motif.

LA FONCTION REPLACEREGEXPR()

Maintenant que vous avez une expression régulière représentant l'ensemble de toutes les dates possibles, vous pouvez vous en servir, non seulement pour détecter une date, mais (pourquoi pas) pour la remplacer. Pour ce faire, vous utiliserez la fonction *ReplaceRegExpr()* :

```

const
EXPRESSION = '([0-3]\d)/([01]\d)/([12]\d\d\d)';
var
s: string;

begin
s := '00/00/2000';
writeln(ReplaceRegExpr(EXPRESSION, s, DateToStr(Now), FALSE)); // 09/06/2015

```



La valeur *FALSE* du quatrième paramètre signifie que vous souhaitez un remplacement simple, c'est-à-dire sans substitutions.

REEMPLACEMENT COMPLEXE PAR SUBSTITUTION

En vous servant toujours du même exemple, voyez maintenant ce que sont les substitutions.

Imaginez que vous vouliez conserver les nombres trouvés dans la chaîne (c'est-à-dire dans la date du jour), mais en changer l'ordre.

Il faut d'abord modifier l'expression régulière, plus précisément y ajouter des parenthèses pour délimiter les trois sous-expressions destinées à capturer les parties de la date, c'est-à-dire les suites de chiffres séparées par les barres obliques :

```
'([0-3]\d)/([01]\d)/([12]\d\d\d)'
```

Ces suites de chiffres, une fois capturées, serviront à composer la chaîne de remplacement. La place à laquelle elles devront être insérées sera marquée par les séquences de caractères '\$1', '\$2' et '\$3'.

Il ne reste plus qu'à appeler la fonction `ReplaceRegExpr()`, avec la constante `TRUE` comme quatrième paramètre, afin d'activer le mécanisme de substitution :

```
writeln(ReplaceRegExpr(EXPRESSION, s, '$3-$2-$1', TRUE)); // 2015/06/09
```

La chaîne trouvée est remplacée par une chaîne composée à la volée. Cette dernière contient les suites de chiffres extraites de la chaîne originale, placées dans un ordre différent, et séparées par des traits d'union.

REEMPLACEMENT COMPLEXE PAR FONCTION.

Mais la fonction `ReplaceRegExpr()` vous permet, au besoin, un contrôle encore plus grand sur le processus de remplacement. En effet, le troisième paramètre peut être une fonction, qui composera la chaîne de remplacement à partir des éléments capturés dans la chaîne originale.

Vous pouvez par exemple, non seulement changer l'ordre des parties de la date, comme vous l'avez déjà fait précédemment, mais aussi remplacer le mois par son nom en anglais.

Voici la fonction qui composera la chaîne de remplacement :

```
function TMyApplication.MyReplaceFunc(ARegExpr: TRegExpr): string;  
const  
MONTH_NAMES: array[1..12] of string = (  
    'January',  
    'February',  
    'March',  
    'April',  
    'May',  
    'June',  
    'July',
```

```

'August',
'September',
'October',
'November',
'December'
);
var
i: Integer;
begin
with ARegExpr do
begin
i := StrToInt(Match[2]);
Result := MONTH_NAMES[i] + ' ' + Match[1] + ', ' + Match[3]; // June 09, 2015
end;
end;

```

La variable *Match[0]* vous aurait fourni la date entière ; les variables *Match[1]* et *Match[3]* vous donnent les parties de la date correspondant aux première et troisième sous-expressions.

La fonction *ReplaceRegExpr()* est appelée de la façon suivante :

```

var
e: TRegExpr;
begin
e := TRegExpr.Create;
e.Expression := '(\d{2})/(\d{2})/(\d{4})';
{$IFDEF OBJFPC}
writeln(e.Replace(DateToStr(Now), @MyReplaceFunc));
{$ELSE}
writeln(e.Replace(DateToStr(Now), MyReplaceFunc));
{$ENDIF}
e.Free;

```



Il est à noter que la fonction qui compose la chaîne de remplacement doit être une méthode de la classe au sein de laquelle la fonction *ReplaceRegExpr()* est appelée.

L'exemple ci-dessus est basé sur le type *TCustomApplication*. Le projet a été créé en choisissant, dans le menu Projet/Nouveau projet, le modèle « Application console ». De cette façon vous n'avez plus qu'à ajouter la déclaration de votre fonction de remplacement dans la déclaration de la classe *TMyApplication* fournie par **Lazarus** :

```

type
TMyApplication = class(TCustomApplication)
protected
procedure DoRun; override;
public
constructor Create(TheOwner: TComponent); override;
destructor Destroy; override;
function MyReplaceFunc(ARegExpr: TRegExpr): string;
end;

```

VALIDATION D'UNE CHAÎNE FEN

Il est temps de récapituler vos connaissances sur la syntaxe des expressions régulières.

Pour ce faire, vous allez imaginer une expression qui servira à contrôler la validité d'une chaîne représentant une position au jeu des échecs.

BREVE PRESENTATION DE LA NOTATION FORSYTH-EDWARDS

La notation Forsyth-Edwards permet de faire tenir dans une chaîne de caractères toutes les données constituant une position au jeu des échecs : placement des pièces, couleur active, droit au roque, droit à la prise « en passant », nombre de demi-coups, nombre de coups.

La position de départ se note ainsi :

```
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
```

Comme vous le voyez, la chaîne est faite de six sous-chaînes séparées par des espaces.

La première chose que vous allez faire est précisément de vérifier que ces sous-chaînes sont bien présentes au nombre de six. Pour cela, vous allez utiliser la fonction [*SplitRegExpr\(\)*](#).

LA FONCTION SPLITREGEXPR()

La fonction [*SplitRegExpr\(\)*](#) reçoit trois arguments : l'expression régulière à remplacer, la chaîne à traiter et une liste de chaînes pour recueillir le résultat.

```
function TFENValidator.IsFEN(const aInputStr: string): Boolean;
var
  a: TStrings;
begin
  a := TStringList.Create;

  SplitRegExpr(' ', aInputStr, a);
  Result := (a.Count = 6);
```

Si le résultat obtenu est conforme au résultat attendu, vous appelez une seconde fois la fonction [*SplitRegExpr\(\)*](#), pour obtenir les sous-chaînes correspondant aux différentes lignes de l'échiquier :

```
var
  a, b: TStrings;

begin
  a := TStringList.Create;
  b := TStringList.Create;
```

```
SplitRegExpr(' ', aInputStr, a);  
Result := (a.Count = 6);
```

```
if Result then  
begin  
  SplitRegExpr('/', a[0], b);  
  Result := (b.Count = 8);  
end;
```

Si à ce stade la variable *Result* vaut toujours *TRUE*, vous pouvez passer à l'étape suivante, qui est de contrôler les différents champs un à un au moyen d'expressions régulières appropriées.

EXPRESSIONS UTILISEES

Voici les expressions dont vous vous servirez :

```
const  
WHITEKING = 'K';  
BLACKKING = 'k';  
PIECES = '^1-8BKNPQRbknpr]+$';  
ACTIVE = '^wb]';  
CASTLING = '^KQkq]+$|^-$';  
ENPASSANT = '^a-h][36]$|^-$';  
HALFMOVE = '^d+$';  
FULLMOVE = '^1-9]d*$';
```

Il n'y a rien dans ces expressions que vous n'avez déjà rencontré, mis à part le caractère spécial '*' qui signifie que le caractère en question (un chiffre) peut apparaître « autant de fois qu'on veut ou pas du tout ».

Vous n'avez plus qu'à vérifier que les chaînes contenues dans les listes a et b correspondent bien au motif, au moyen de la fonction *ExecRegExpr()*.

REMPLACEMENT COMPLEXE

Les chaînes de la liste b vont toutefois être soumises à une vérification un peu plus sophistiquée. En effet, vous ne voulez pas seulement savoir si ces chaînes ne contiennent que des caractères autorisés : vous voulez également vous assurer que le nombre total de cases pour chaque ligne est bien de huit.

Pour ce faire, vous allez remplacer les chiffres représentant le nombre de cases vides successives par une quantité correspondante de symboles répétés. Ainsi la ligne 'pp1ppppp' deviendra 'pp-ppppp'. De cette façon vous connaîtrez le nombre total de cases, d'après la longueur de la chaîne.

Voici la fonction qui va effectuer ce remplacement :

```
function TFENValidator.ExpandDigit(aRegExpr: TRegExpr): string;
```

```

const
  SYMBOL = '-';
begin
  Result := '';
  with aRegExpr do
    Result := StringOfChar(SYMBOL, StrToInt(Match[0]));
end;

```

EXEMPLE COMPLET

Et voici l'exemple complet :

```

{$MODE DELPHI}
{$B-}

uses
  classes, sysutils, regexpr;

type
  TFENValidator = class
  public
    function ExpandDigit(aRegExpr: TRegExpr): string;
    function IsFEN(const aInputStr: string): Boolean;
  end;

var
  loglist: TStringList;
  logfile: string;

procedure ToLog(const aText: string);
begin
  loglist.Add(Format('%s %s', [TimeToStr(Now), aText]));
end;

function TFENValidator.ExpandDigit(aRegExpr: TRegExpr): string;
const
  SYMBOL = '-';
begin
  Result := '';
  with aRegExpr do
    Result := StringOfChar(SYMBOL, StrToInt(Match[0]));
end;

function TFENValidator.IsFEN(const aInputStr: string): boolean;
const
  WHITEKING = 'K';
  BLACKKING = 'k';
  PIECES   = '^1-8BKPNQRbknpqr]+';
  ACTIVE   = '^1[wb]$';
  CASTLING = '^1[KQkq]+$|^1-$';
  ENPASSANT = '^1[a-h][36]$|^1-$';
  HALFMOVE = '^1\d+$';
  FULLMOVE = '^1[1-9]\d*+';
var
  a, b: TStrings;
  i: Integer;

```

```

e: TRegExpr;
s: string;

begin
a := TStringList.Create;
b := TStringList.Create;

e := TRegExpr.Create;
e.Expression := '\d';

SplitRegExpr(' ', aInputStr, a);

Result := (a.Count = 6);

if Result then
begin
SplitRegExpr('/', a[0], b);
Result := (b.Count = 8);
end;

if Result then
begin
Result := Result and ExecRegExpr(WHITEKING, a[0]);
Result := Result and ExecRegExpr(BLACKKING, a[0]);

for i := 0 to 7 do
begin
Result := Result and ExecRegExpr(PIECES, b[i]);
if Result then
begin
s := b[i];
repeat
s := e.Replace(s, ExpandDigit);
until not ExecRegExpr('\d', s);
ToLog(Format('%s %s', [{ $I %LINE% }, s]));
Result := Result and (Length(s) = 8);
end;
end;

Result := Result and ExecRegExpr(ACTIVE, a[1]);
Result := Result and ExecRegExpr(CASTLING, a[2]);
Result := Result and ExecRegExpr(ENPASSANT, a[3]);
Result := Result and ExecRegExpr(HALFMOVE, a[4]);
Result := Result and ExecRegExpr(FULLMOVE, a[5]);
end;

a.Free;
b.Free;
e.Free;
end;

const
SAMPLE: array[1..5] of string = (
'rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1',
'rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2',
'rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2',
'4k3/8/8/8/8/4P3/4K3 w - - 5 39',
'5k3/8/8/8/8/4P3/4K3 w - - 5 39'
);

```

```

var
  i: Integer;

begin
  loglist := TStringList.Create;
  logfile := ChangeFileExt({'$I %FILE%'}, '.log');

  with TFENValidator.Create do
    for i := Low(SAMPLE) to High(SAMPLE) do
      writeln(BoolToStr(IsFEN(SAMPLE[i]), TRUE));

  with loglist do
    begin
      SaveToFile(logfile);
      Free;
    end;
  end.

```

L'une des chaînes du tableau SAMPLE n'est pas valide. Voyez-vous laquelle ?

Si vous ne voyez pas, rendez-vous dans le fichier ***.LOG** :

```
-----k---
```

Il y a une case de trop sur cette ligne : la chaîne n'est donc pas valide.

DEUX FONCTIONS UTILES

Vous avez fait le tour de la syntaxe des expressions régulières et des fonctions de l'unité *RegExpr*. Il vous reste à faire connaissance avec deux fonctions qui peuvent rendre service.

LA FONCTION QUOTEREGEXPRMETACHARS()

La fonction *QuoteRegExprMetaChars()* reçoit comme argument une expression régulière dont on aurait oublié « d'échapper » les caractères spéciaux. La fonction détecte ces caractères et les fait précéder du symbole '\'. Par exemple, le code suivant :

```
writeln(QuoteRegExprMetaChars('(abc)'));
```

donnera ce résultat :

```
\(abc\)
```

Pratique, n'est-ce pas ?

LA FONCTION `RegExprSubExpressions()`

La fonction `RegExprSubExpressions()` extrait les sous-expressions d'une expression donnée :

```
var
  liste: TStringList;
  i: Integer;

begin
  liste := TStringList.Create;

  if RegExprSubExpressions('(\d)(\d)', liste, FALSE) = 0 then
    for i := 0 to liste.Count - 1 do
      writeln(liste.Strings[i]);
```

Lorsque l'extraction s'est faite sans erreur, la fonction renvoie 0. C'est le cas ici, et voici ce qu'affiche votre programme :

```
(\d)(\d)
\d
\d
```

Le premier élément de la liste est l'expression complète ; les deux éléments suivants sont les sous-expressions.

Le troisième paramètre de la fonction `RegExprSubExpressions()` permet d'autoriser ou non la syntaxe étendue. Concrètement, comme vous l'apprend la documentation de l'unité *RegExpr*, ce paramètre doit avoir la valeur `TRUE` si le modificateur `m` est activé.

MODIFICATEURS

Encore un mot pour vous dire ce que sont ces *modificateurs*, et vous aurez terminé ce tour d'horizon des expressions régulières.

Les modificateurs permettent de changer le comportement des fonctions de la bibliothèque. Il y a six modificateurs, désignés par les lettres `i`, `m`, `s`, `g`, `x`, `r`.

Le modificateur `i`, par exemple, permet de ne pas tenir compte de la différence entre majuscules et minuscules. Voici comment on l'utilise :

```
var
  re: TRegExpr;
begin
  re := TRegExpr.Create;
  re.ModifierI := TRUE;
```

Le modificateur `m`, dont nous avons déjà parlé, étend la signification des caractères spéciaux `'^'` et `'$'` aux débuts et aux fins de ligne à l'intérieur de la chaîne traitée.

Récapitulation :

Modificateur	Fonction
<code>i</code>	La détection ne tient pas compte de la différence entre majuscules et minuscules.
<code>m</code>	Détection des fins et des débuts de ligne au moyen des caractères spéciaux <code>'^'</code> et <code>'\$'</code> .
<code>s</code>	Traiter toute la chaîne comme une ligne. Pratiquement, étend la signification du caractère <code>'.'</code> à tous les caractères, y compris l'espace au sens large.
<code>g</code>	Mode vorace. La valeur par défaut de ce modificateur est <code>TRUE</code> . Lorsque la valeur est <code>FALSE</code> , <code>'+'</code> fonctionne comme <code>'*?'</code> , <code>'*'</code> comme <code>'*?'</code> et ainsi de suite.
<code>x</code>	Extension de la syntaxe pour une plus grande lisibilité des expressions. Autorise espaces et commentaires.
<code>r</code>	Inclut des caractères russes. Il est possible d'adapter cette option à un autre langage en changeant la valeur de la variable <code>RegExprModifierR</code> .

Le modificateur `x` permet d'écrire, par exemple, la chose suivante :

```
const
PATTERN = '[KQkq]+ | \- # Les espaces ne seront pas pris en compte !';
```

Vous pourrez l'utiliser ainsi :

```
var
re: TRegExpr;

begin
  re := TRegExpr.Create;
  with re do
    begin
      Expression := PATTERN;
      ModifierX := TRUE;
      writeLn(ModifierStr);
      writeLn(Exec('-'));
      Free;
    end;
```

Et le résultat sera :

```
xgsr-im
TRUE
```

CONCLUSION

Il reste trois ou quatre petites choses dont nous aurions pu parler, mais peut-être pensez-vous que vous savez déjà pratiquement tout ce qu'il faut savoir sur les expressions régulières et l'unité *RegExpr* d'Andrey V. Sorokin.

À vous de jouer !

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.

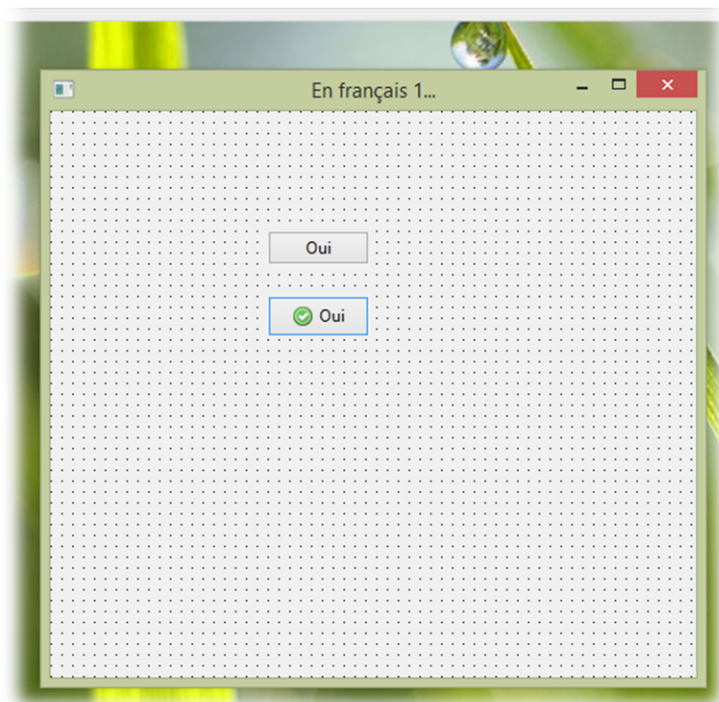
UN PROGRAMME FRANÇAIS... EN ANGLAIS

[Exemple TR_01]

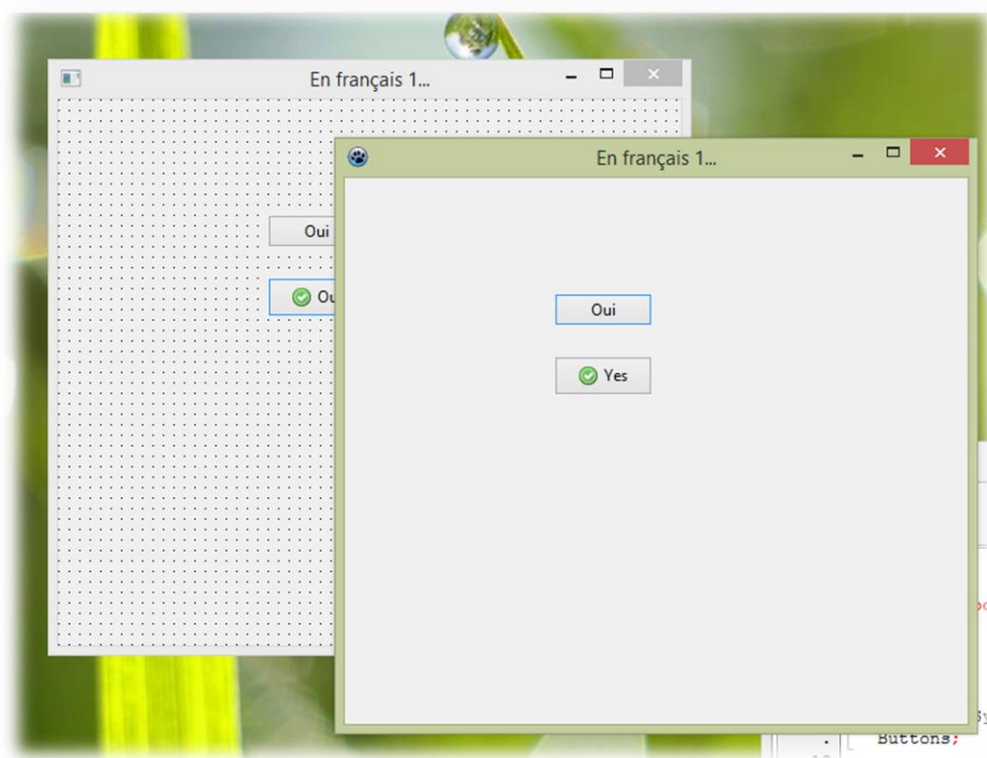
Pour vous en persuader, examinez le comportement d'un programme aussi élémentaire que celui-ci :

- 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.

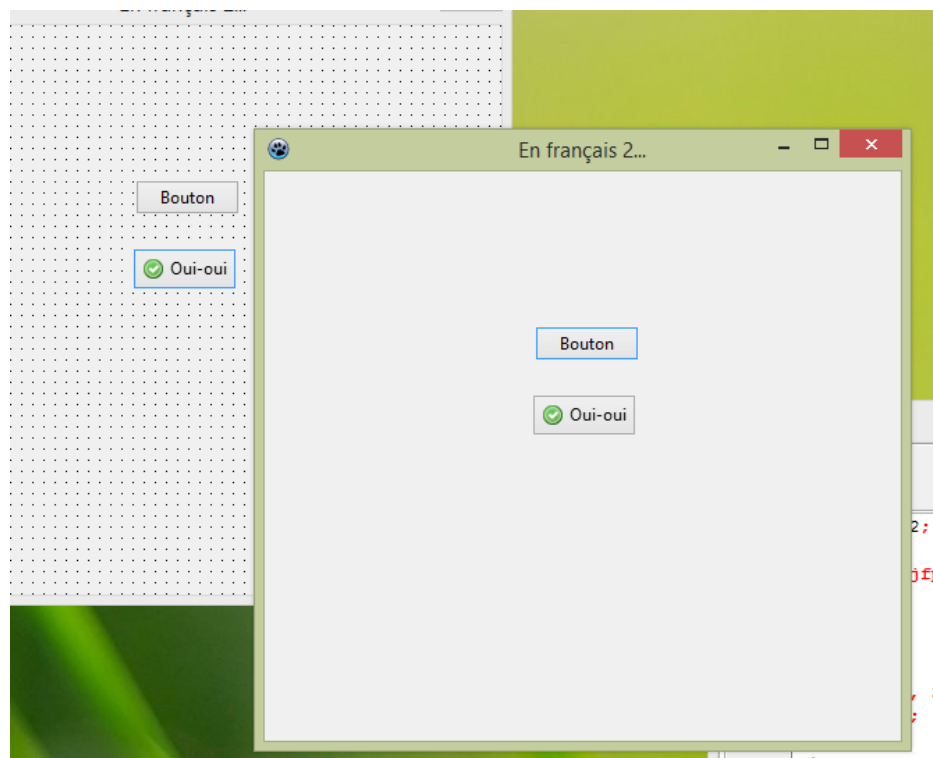
UN PEU DE BRICOLAGE POUR TRADUIRE

[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 :

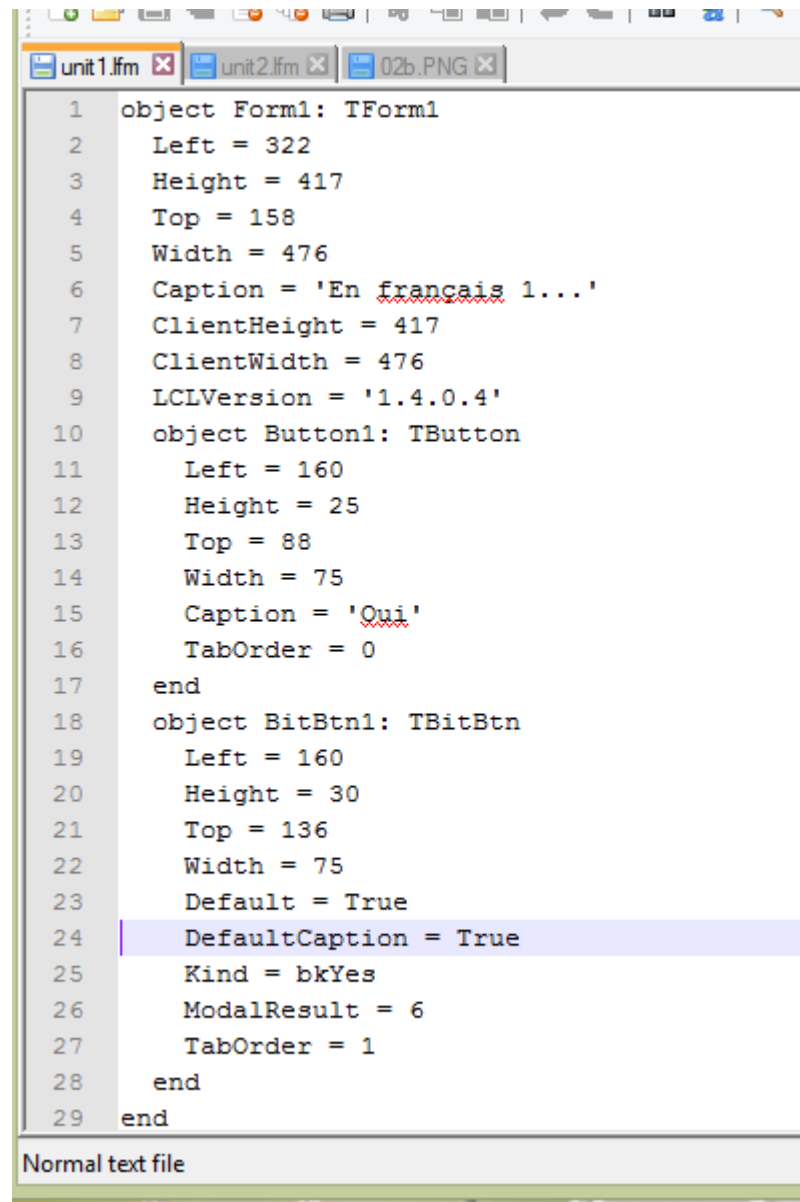
- modifiez la valeur de la légende (**Caption**) en la passant de **&Oui** à **&Oui-oui** ;
- compilez le programme ;
- lancez son exécution.

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 :

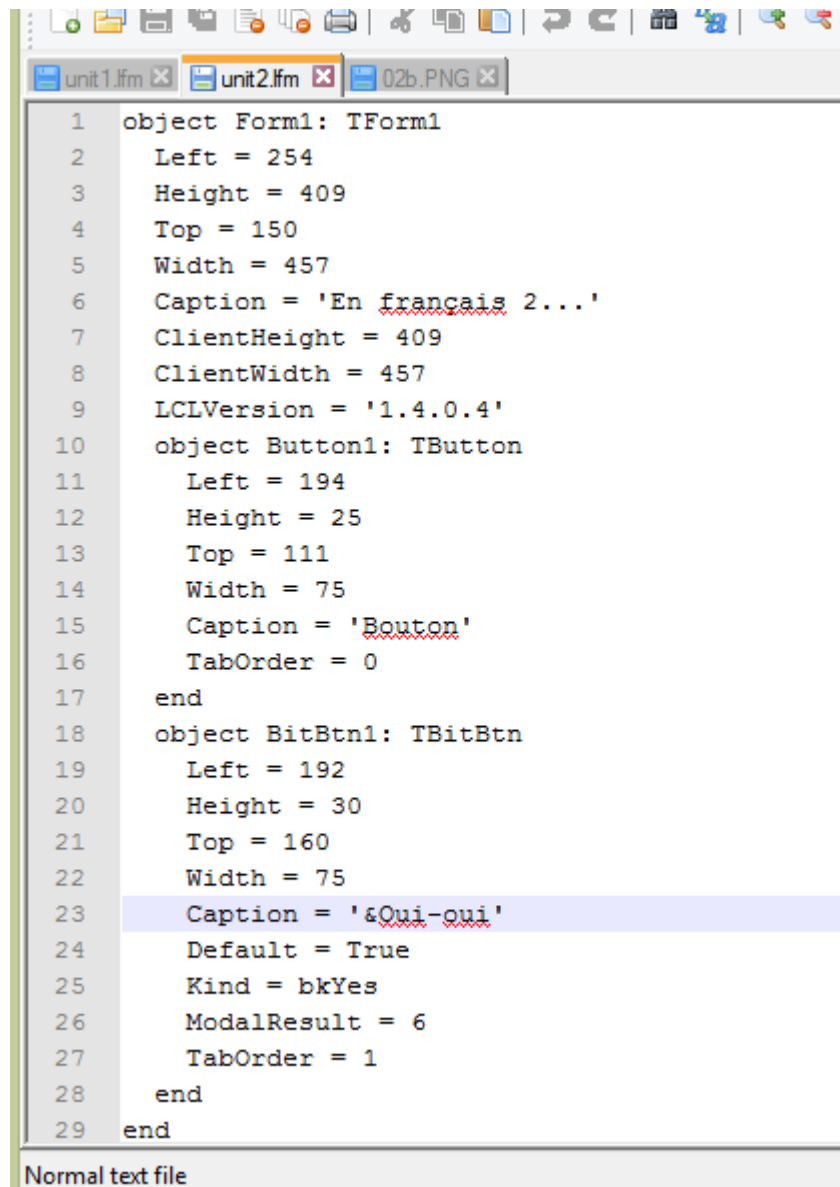


```
1  object Form1: TForm1
2    Left = 322
3    Height = 417
4    Top = 158
5    Width = 476
6    Caption = 'En français 1...'
7    ClientHeight = 417
8    ClientWidth = 476
9    LCLVersion = '1.4.0.4'
10   object Button1: TButton
11     Left = 160
12     Height = 25
13     Top = 88
14     Width = 75
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 *BitBtn1* 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

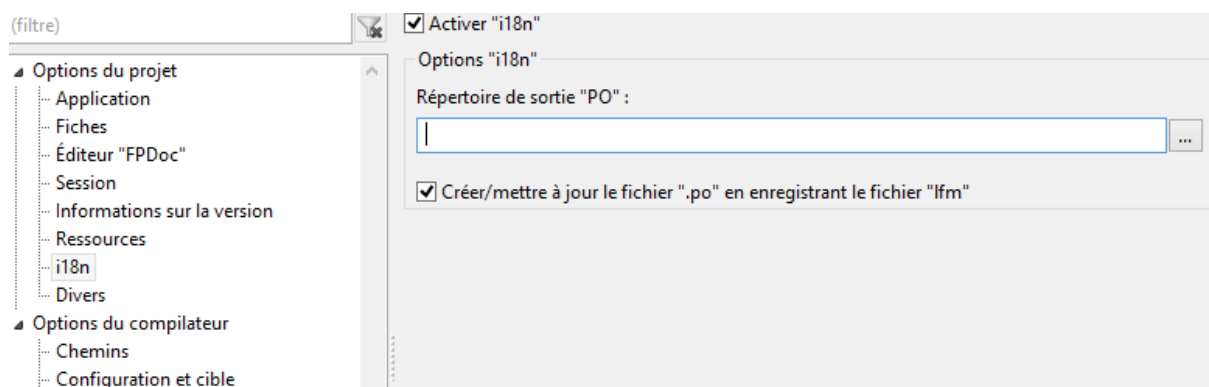
Vous pourriez vous satisfaire des deux premières solutions pour les propriétés accessibles en écriture. Malheureusement, de nombreux messages ne sont pas de ce type : certaines propriétés (comme le nom des couleurs) et la plupart des messages d'erreur sont hors de portée des unités créées.

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;
{$mode objfpc}{$SH+}
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**.

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

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

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 !';

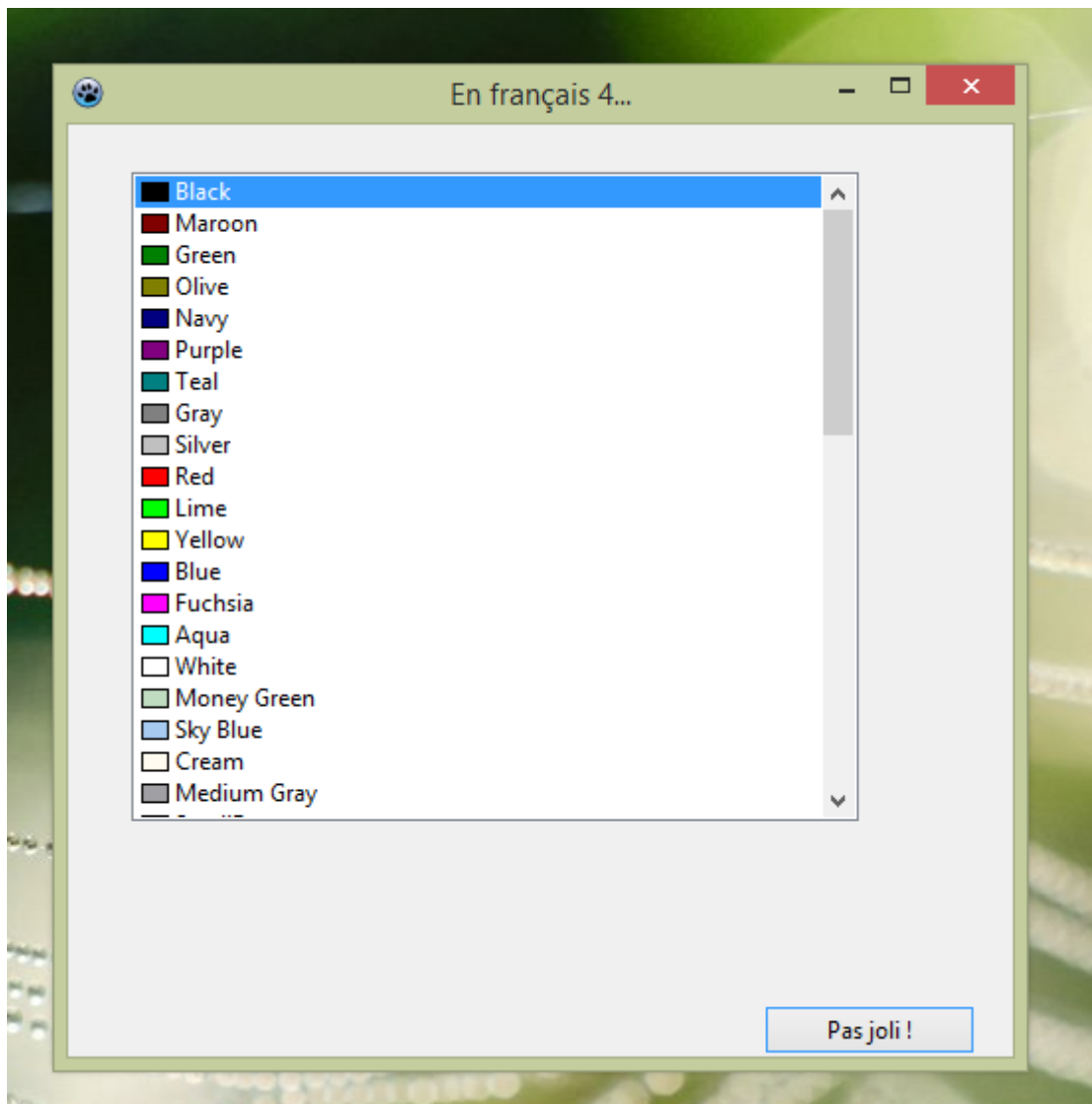
{$R *.lfm}

{ TForm1 }

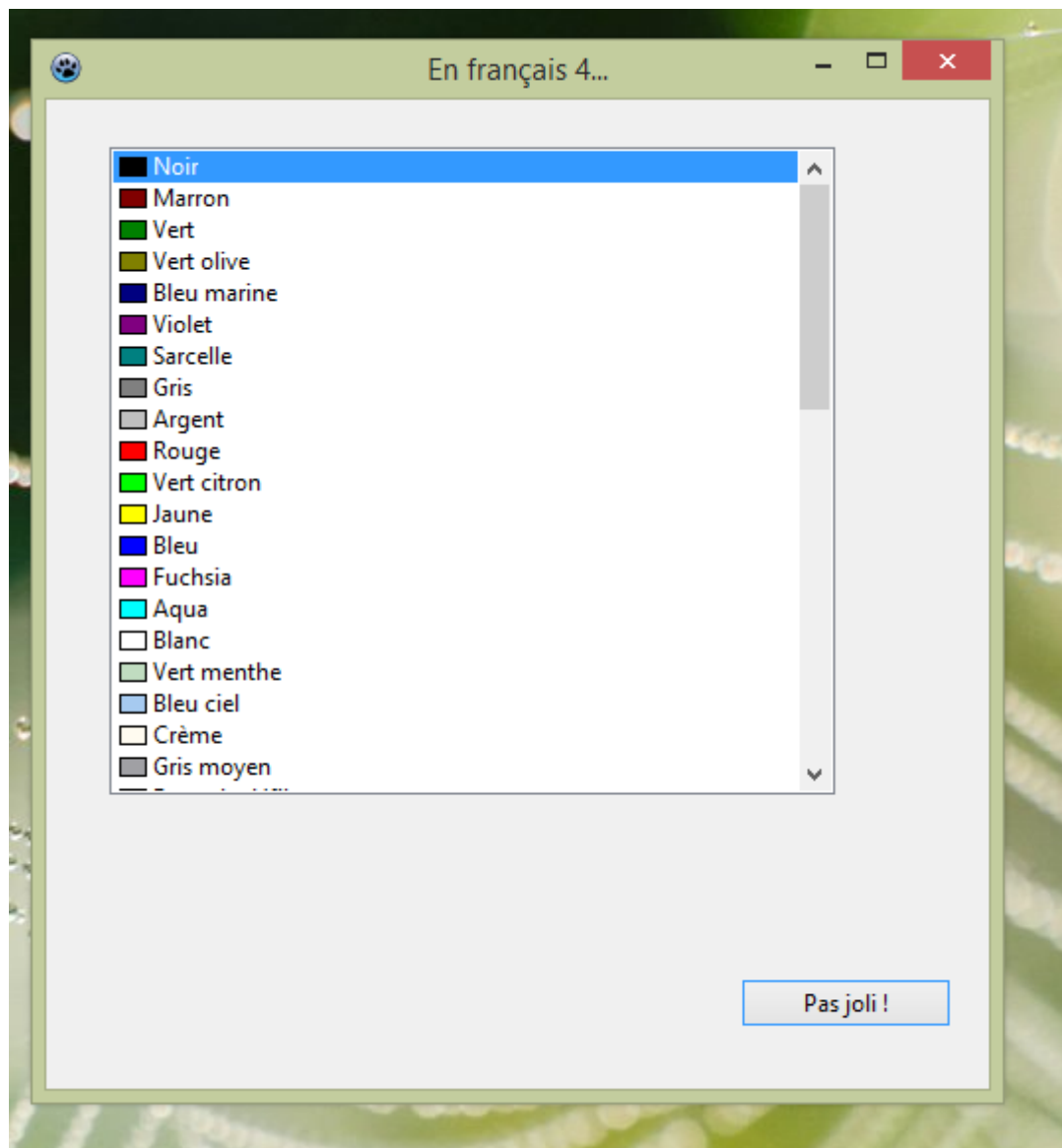
procedure TForm1.Button1Click(Sender: TObject);
begin
  if cbPrettyNames in ColorListBox1.Style then
    begin
      // propriété exclue
      ColorListBox1.Style := ColorListBox1.Style - [cbPrettyNames];
      Button1.Caption := RS_Pretty;
    end
  else
    begin
      // propriété incluse
      ColorListBox1.Style := 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 PO³ 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.

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 !



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 ""

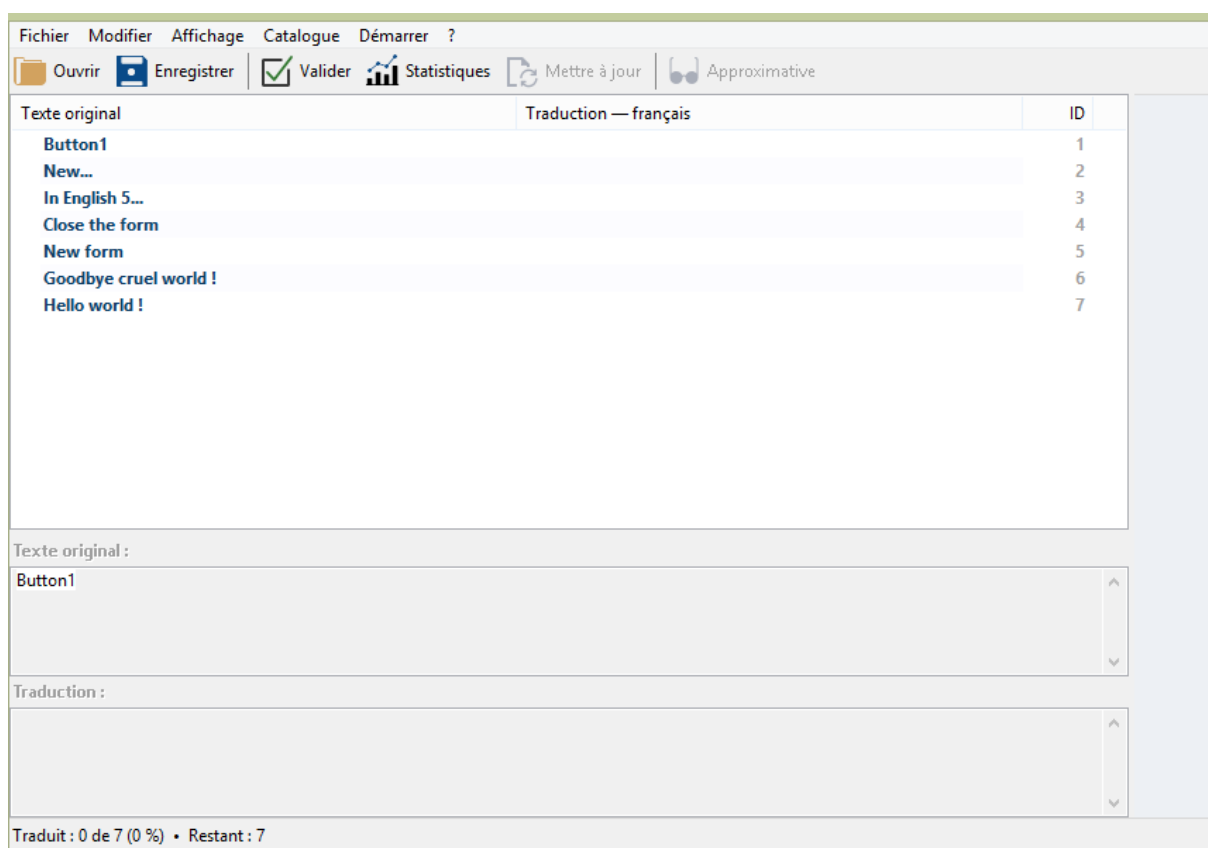
#: unit1.rs_hello
msgid "Hello world !"
msgstr ""
```

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



Proposez la traduction suivante :

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;
    lblLanguages: TListBox;
    pnlData: TPanel;
  procedure btnRestartClick(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure lblLanguagesClick(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 : *GVTTranslate*. C'est elle qui a en charge l'accès aux fichiers de langue, mais aussi le redémarrage de l'application après l'enregistrement des changements.

LA MEMORISATION DU CHOIX ET LE REDEMARRAGE DE L'APPLICATION

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

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

```
{ TGVTranslate }

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

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

Voici le listing commenté de cette méthode :

```
procedure TGVTranslate.Restart;
// *** redémarrage de l'application ***
Var
  Exe: TProcessUTF8;
begin
  Exe := TProcessUTF8.Create(nil); // processus créé
  try
    Exe.Executable := Application.ExeName; // il porte le nom de l'application
    // ajout des paramètres
    Exe.Parameters.Add(Language); // langue en paramètre
    Exe.Parameters.Add(FileDir); // répertoire
    Exe.Parameters.Add(FileName); // nom de fichier
    Exe.Execute; // on démarre la nouvelle application
  finally
    Exe.Free; // processus libéré
    Application.Terminate; // l'application en cours est terminée
  end;
end;
```

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



Cette procédure est facilement réutilisable dans d'autres contextes.

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

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

resourcestring
  RS_FallBackLanguage = 'auto';

{ TGVTranslate }

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

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

procedure TGVTranslate.SetFileName(const AValue: string);
// *** détermine le nom du fichier ***
begin
  if AValue <> '' then // pas valeur par défaut ?
    // à partir de l'extraction du nom du fichier
```



```

    fFileName := ExtractFileName(AValue)
  else
    // à partir du nom du programme
    fFileName := ExtractFileNameOnly(Application.ExeName);
  end;

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

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

```



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

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

```

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

```



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

BILAN

Dans ce chapitre, vous aurez appris à :

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

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;
```

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.

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

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 POO 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 avant tout 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]

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

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

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 POO.

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.

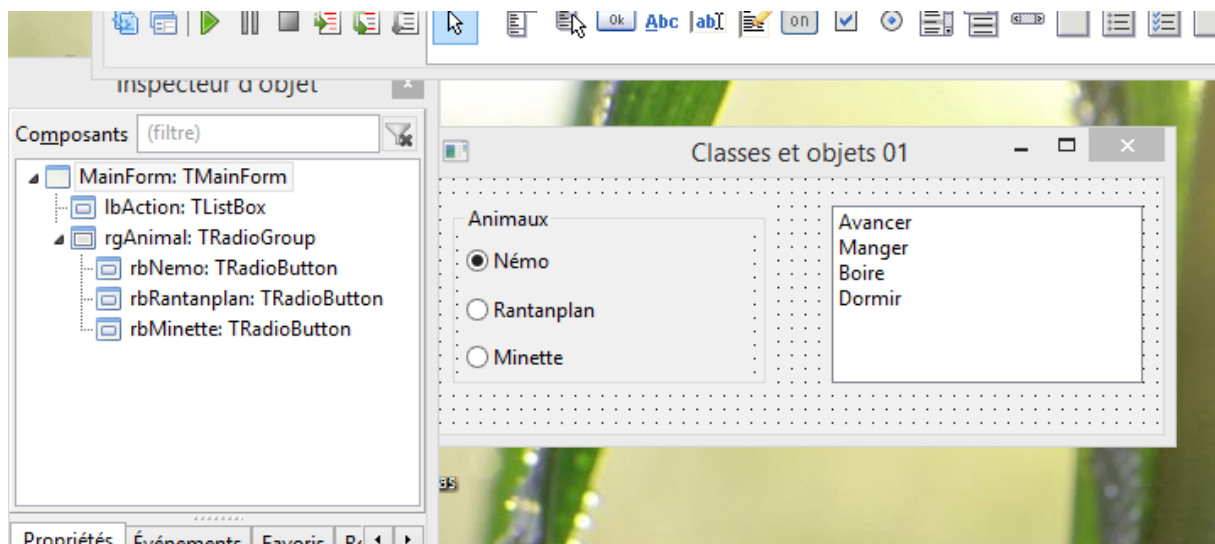
LES OBJETS

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 :



- cliquez sur la propriété *Items* du composant *TListBox* et complétez la liste qui apparaît toujours selon le modèle précédent : « Avancer », « Manger », « Boire » et « Dormir » ;

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.

- 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 :

```
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;
```

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

```

    3: UnAnimal.Dormir;
    end;
end;

procedure TMainForm.rbMinetteClick(Sender: TObject);
// *** l'animal est Minette ***
begin
    UnAnimal := Minette;
end;

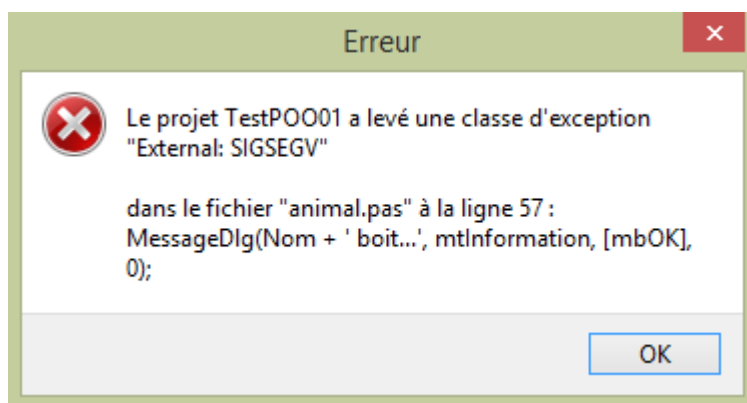
procedure TMainForm.rbNemoClick(Sender: TObject);
// *** l'animal est NémO ***
begin
    UnAnimal := Nemo;
end;

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 !



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.

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.

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

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
  // 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 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;
```

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.

```

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

¹³ Voir le chapitre suivant.

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.



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 RemuerLaQueue;
  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 **RemuerLaQueue**) 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 RemuerLaQueue;
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.RemuerLaQueue;
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.RemuerLaQueue;
begin
  MessageDlg(Nom + ' remue 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;
```

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*, *RemuerLaQueue* 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 sera 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
```

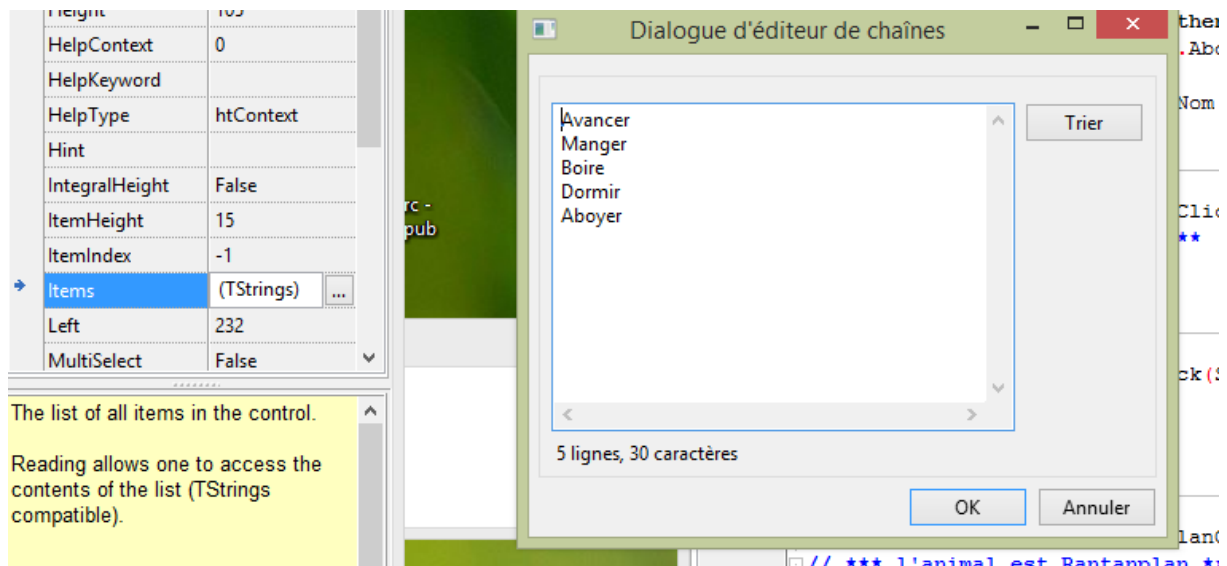
Le déclenchement possible d'une erreur avec *as* conduit à l'accompagner la plupart du temps d'un test préalable avec *is* :

```
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.

BILAN

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...

Dans cette partie, vous étudierez les fondements de l'utilisation des méthodes.

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 RemuerLaQueue;
property Batard: Boolean read fBatard write SetBatard;
end;

```

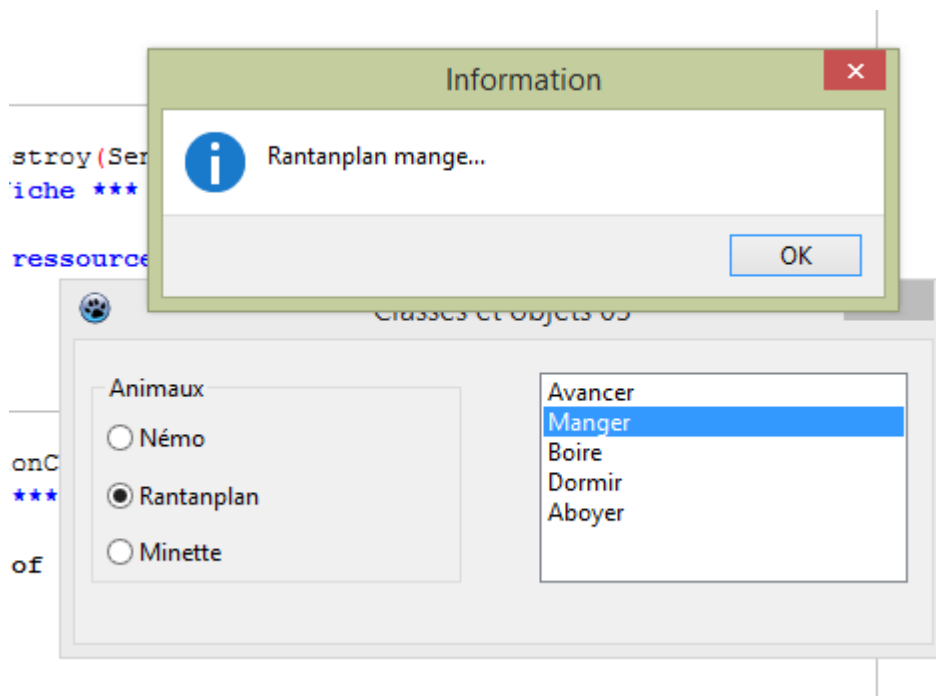
- pressez simultanément Ctrl-Maj-C pour demander à **Lazarus** de générer le squelette de la nouvelle méthode ;
- complétez ce squelette en vous servant du modèle suivant :

```

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 // [...]

```

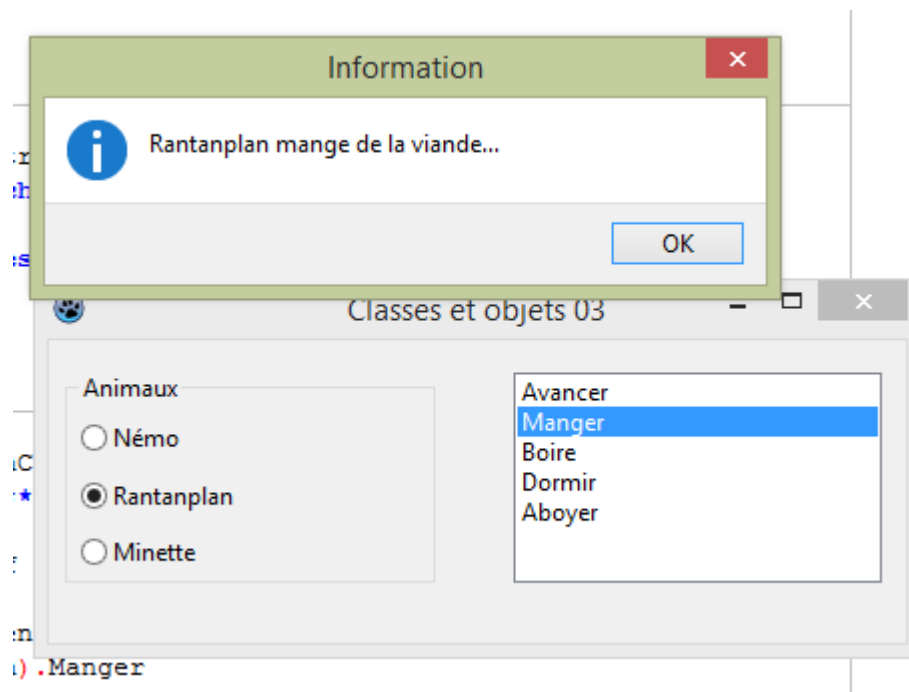
En effet, en écrivant *UnAnimal.Manger*, vous demandez à un animal de manger et non à un chien ! Vous obtenez logiquement ce que sait faire tout animal, à savoir manger, et non la spécialisation de ce que fait un chien carnivore.

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

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

d'exécution qui permet de retrouver à l'exécution 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 RemuerLaQueue;
```

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

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.

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
```

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*.

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 !

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  
  [...]  
end;
```

La place d'*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 Destroy ; // 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 :

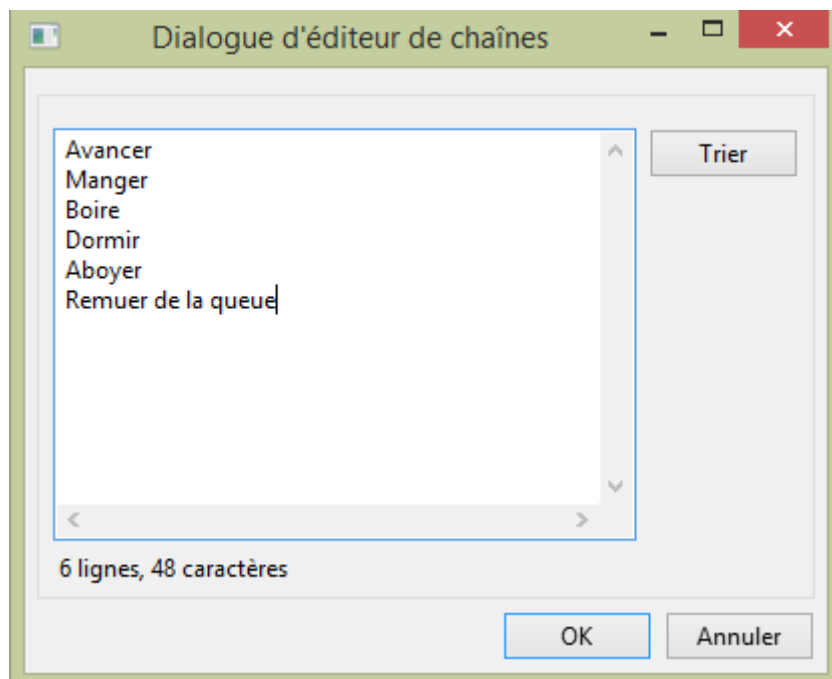
```
procedure TChien.RemuerLaQueue;  
begin  
    inherited Manger; // <= Manger vient de TAnimal !  
    MessageDlg('C'est pourquoi il remue la queue...', mtInformation, [mbOK], 0);  
end;
```

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 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).RemuerLaQueue
else
    MessageDlg(UnAnimal.Nom + ' ne sait pas remuer 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

Dans cette partie, vous étudierez certains aspects des méthodes qu'il n'est pas nécessaire d'approfondir dans un premier temps, mais qui seront parfois très utiles.

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
```

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

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);
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. En revanche, elles peuvent avoir accès aux champs de classe, propriétés de classe et méthodes de classe : comme les autres membres d'une classe qui sont indépendants de l'instanciation, leur déclaration commence toujours par le mot *class*. Par exemple, une variable de classe sera déclarée ainsi :

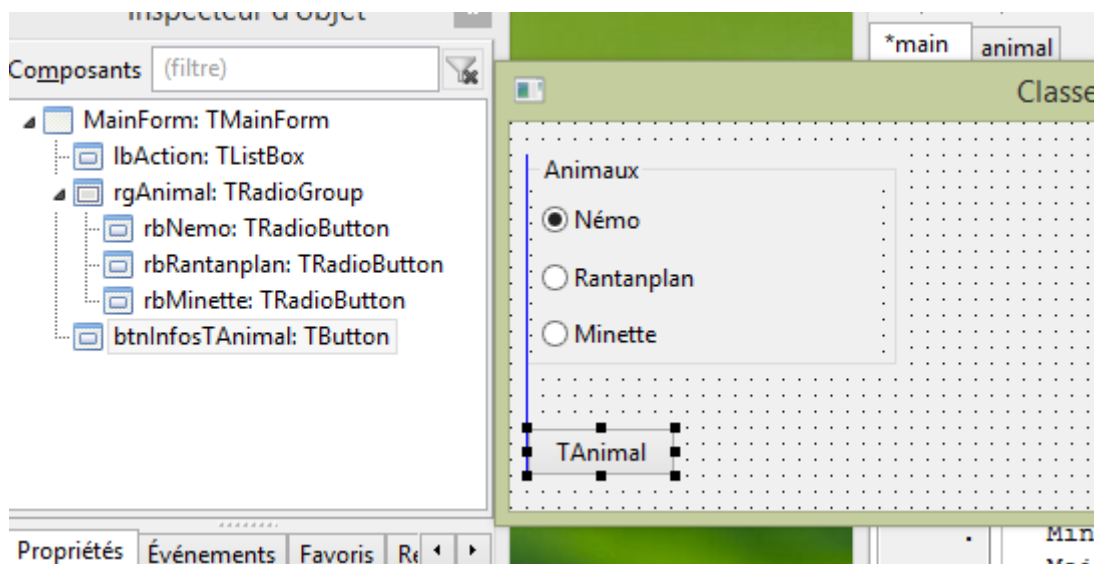
```
class var MyVar : Integer ;
```

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 :

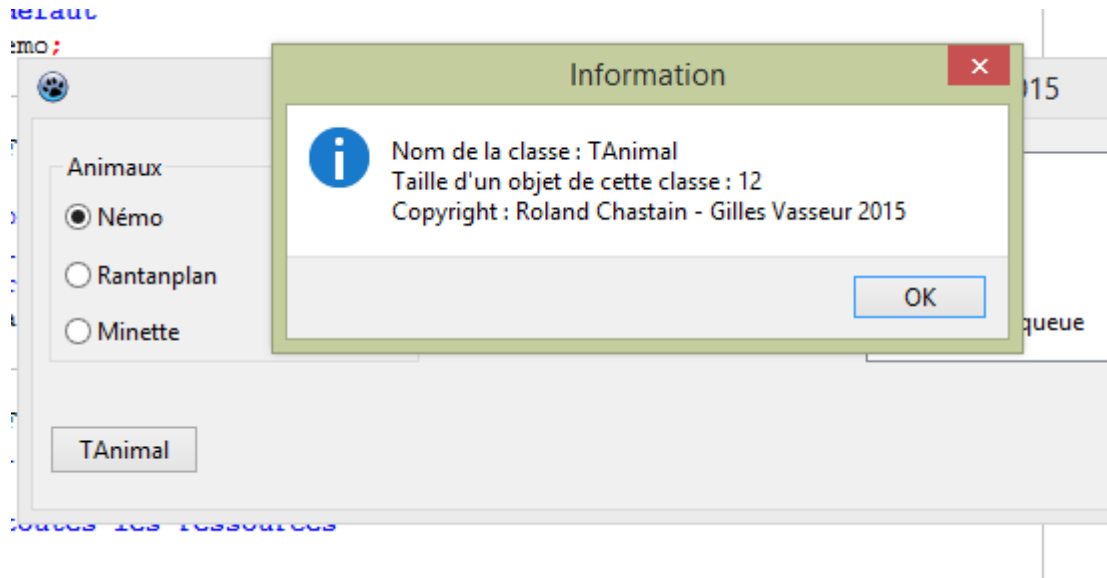
- ajoutez un bouton à la fiche principale, renommez-le *btnInfosTAnimal* et changez sa légende en *TAnimal* ;



- 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 *IntanceSize* 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 :

```

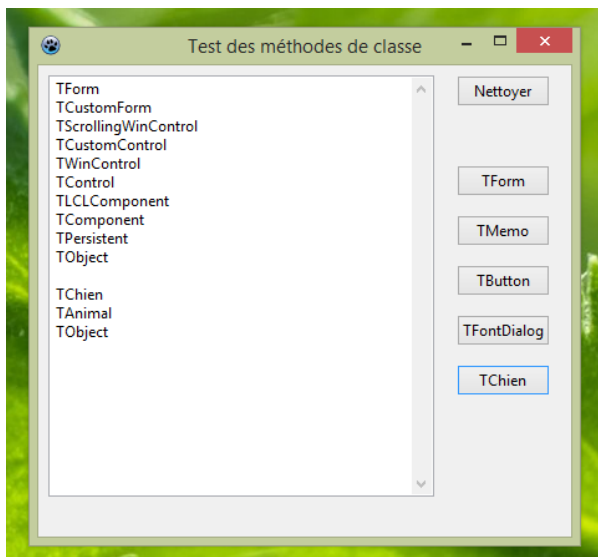
procedure TMainForm.Display(AClass: TClass);
begin
  repeat
    mmoDisplay.Lines.Add(AClass.ClassName);
    AClass := AClass.ClassParent;
  until AClass = nil;
  mmoDisplay.Lines.Add("");
end;

```

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; // m  mo 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; // unit   pour le traitement des animaux

{ TMainForm }

procedure TMainForm.btnFormClick(Sender: TObject);
// *** g  n  alogie de TForm ***
begin
  Display(TForm);
end;

procedure TMainForm.btnMemoClick(Sender: TObject);
// *** g  n  alogie de TMemo ***
begin
  Display(TMemo);
end;

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

procedure TMainForm.btnFontDialogClick(Sender: TObject);
// *** g  n  alogie de TFontDialog ***
begin
  Display(TFontDialog);
end;

procedure TMainForm.btnButtonClick(Sender: TObject);
// *** g  n  alogie de TButton ***
begin
  Display(TButton);
end;

procedure TMainForm.btnChienClick(Sender: TObject);
// *** g  n  alogie 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.

```

METHODES STATIQUES DE CLASSE

En ajoutant le mot réservé *static* à la fin de la déclaration d'une méthode classe, vous obtenez une *méthode statique de classe*. 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 statiques de classe ne peuvent ni accéder à *Self* ni être déclarées virtuelles.

Les méthodes statiques de classe 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, comme pour les méthodes de classe ordinaires, vous pouvez déclarer des *variables de classe*, des *propriétés de classe* et des *méthodes statiques de classe* 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 statiques de classe 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 := Uppcase(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.

Vous pouvez enfin créer des *constructeurs et des destructeurs de classe*. Cette possibilité est utile si vous avez besoin d'initialiser des variables de classe avant même d'utiliser votre classe. Les avantages de cette technique par rapport à l'utilisation d'*initialization* et *finalization* sont que le code de la classe ne sera pas chargé par le compilateur, gagnant par conséquent en place mémoire, et que les structures n'auront pas besoin d'être toutes initialisées, ce qui accélère le traitement.

Des restrictions s'appliquent à leur utilisation : ils doivent impérativement s'appeler *Create* et *Destroy*, ne pas être déclarés virtuels et ne pas comporter de paramètres.

Leur comportement est aussi atypique :

- le constructeur est appelé automatiquement au lancement de l'application avant même l'exécution de la section *initialization* de l'unité dans laquelle il a été déclaré ;
- le destructeur est lui aussi appelé automatiquement, mais après l'exécution de la section *finalization* de la même unité ;
- Une conséquence importante de ces particularités est que les deux vont être appelés même si la classe n'est jamais utilisée dans l'application ;
- Une autre conséquence est que vous ne les invoquerez jamais explicitement.

[Exemple PO-11]

Pour exemple, une petite application permet de récupérer et d'afficher le résultat d'une méthode ordinaire d'une classe sans avoir apparemment à l'instancier. En fait, c'est le constructeur de classe qui se charge de l'instanciation avant même que le code d'initialisation de l'unité n'ait été exécuté :

```

unit main;

{$mode objfpc}{$H+}

interface

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

type

```

```

{ TMyClass }

TMyClass = class // classe de test
private
  class var fClass: TMyClass;
  class constructor Create;
  class destructor Destroy;
public
  function MyFunct: string; // méthode ordinaire
  class property Access: TMyClass read fClass; // accès au champ de classe
end;

{ TMainForm }

TMainForm = class(TForm)
  btnGO: TButton;
  procedure btnGOClick(Sender: TObject); // test en cours d'exécution
private
  { private declarations }
public
  { public declarations }
end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMyClass }

class constructor TMyClass.Create;
// *** constructeur de classe ***
begin
  fClass := TMyClass.Create; // on crée la classe
  MessageDlg('Class constructor', mtInformation, [mbOK], 0);
end;

class destructor TMyClass.Destroy;
// *** destructeur de classe ***
begin
  fClass.Free; // on libère la classe
  MessageDlg('Class destructor', mtInformation, [mbOK], 0);
end;

function TMyClass.MyFunct: string;
// *** fonction de test ***
begin
  Result := 'C'est fait!';
end;

{ TMainForm }

procedure TMainForm.btnGOClick(Sender: TObject);
// *** appel direct de la classe ***
begin
  btnGO.Caption := TMyClass.Access.MyFunct;
end;

```


initialization

```
MessageDlg('Initialization : ' + TMyClass.Access.MyFuncnt, mtInformation, [mbOK], 0);
```

finalization

```
MessageDlg('Finalization', mtInformation, [mbOK], 0);
```

end.

Afin de bien montrer l'ordre d'appel, des fonctions *MessageDlg* ont été incorporées dans les méthodes. Vous constaterez que les appels du constructeur et du destructeur de classe encadrent bien ceux des sections *initialization* et *finalization*.

Le mécanisme de l'ensemble est celui-ci :

- le constructeur de classe *Create* est appelé automatiquement : il est chargé de créer une instance de la classe qui est assignée à la variable de classe *fClass* et d'afficher le message spécifiant qu'il a été exécuté ;
- le code de la section *initialization* est appelé : un message adapté est affiché ;
- un éventuel clic sur le bouton *btnGo* affecte le résultat de la méthode *MyFuncnt* à sa propriété *Caption* : ce résultat est récupéré par la propriété de classe *Access* via la classe *TMyClass* (et non une instance de cette classe) ;
- le code de la section *finalization* est appelé : un message adapté est affiché ;
- le destructeur de classe *Destroy* est appelé : il libère l'instance de classe et affiche son propre message.

METHODES DE MESSAGES

Un autre type de méthode mérite d'être signalé bien que son importance soit moins cruciale pour le programmeur contemporain : les *méthodes de messages*. Chargées de traiter les messages les concernant, elles restent essentiellement utiles pour le traitement au plus près des systèmes d'exploitation (routines *callback* en particulier). Il reste cependant intéressant de les connaître pour résoudre certains problèmes avec une économie de moyens remarquable.

La génération et la distribution des messages s'organisent ainsi :

- un événement survient dans le système : un clic de la souris, une touche du clavier pressée, un élément de l'interface modifié...
- l'OS génère un message qui est placé dans la file d'attente de l'application concernée ;
- l'application récupère le message depuis la file à partir d'une boucle pour le transmettre à l'élément concerné ;
- l'élément concerné réagit en fonction du message.

Si, comme le fait sans cesse **Free Pascal** avec ses bibliothèques, il est possible de gérer les messages de l'OS, vous serez invité ici à générer vos propres messages.

Sans que vous ayez à le préciser, les méthodes de messages sont toujours virtuelles. Leur déclaration se clôt par la directive *Message*, elle-même suivie d'un entier ou d'une chaîne courte de caractères :

```
procedure Changed(var Msg: TLMMessage); message M_CHANGEDMESSAGE;  
procedure AClick(var Msg); message 'OnClick';
```

Dans l'exemple précédent, Msg est soit une variable sans **type** soit du **type** *TLMessage* de l'unité *LMessages*. De son côté, *M_CHANGEDMESSAGE* est une constante définie par l'utilisateur à partir de la constante *LM_User* de la même unité *LMessages* :

```
const  
  M_CHANGEDMESSAGE = LM_User + 1; // message de changement
```

L'implémentation d'une méthode de message ne diffère en rien d'une méthode ordinaire si ce n'est qu'elle ne sera jamais appelée directement, mais *via* une méthode de répartition : *Dispatch* pour un message de type entier et *DispatchStr* pour un message de type chaîne. C'est cette méthode de répartition qui émet le message et attend son traitement. Si le message n'est pas traité, il parvient en bout de course à la méthode *DefaultHandler* (ou *DefaultHandlerStr* pour une chaîne) de *TObject* : cette méthode ne fait rien, mais elle peut être redéfinie puisque déclarée *virtual*.

[Exemple PO-12]

Afin que tout cela devienne plus clair, vous allez créer une nouvelle application dont les objectifs vont être de récupérer les messages émis à propos des changements d'un éditeur *TEdit* et de signaler l'absence de traitement du clic sur un bouton *TButton* :

```
unit main;  
  
{$mode objfpc}{$H+}  
  
interface  
  
uses  
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,  
  LMessages; // unité pour les messages  
  
const  
  M_CHANGEDMESSAGE = LM_User + 1; // message de changement  
  M_LOSTMESSAGE = LM_User + 2; // message perdu  
  
type  
  
  { TMainForm }  
  
  TMainForm = class(TForm)  
    btnLost: TButton;  
    edtDummy: TEdit;  
    mmoDisplay: TMemo;  
    procedure btnLostClick(Sender: TObject);  
    procedure edtDummyChange(Sender: TObject);
```

```

private
  { private declarations }
public
  { public declarations }
  procedure Changed(var Msg: TLMMessage); message M_CHANGEDMESSAGE;
  procedure DefaultHandler(var AMessage); override;
end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.edtDummyChange(Sender: TObject);
// *** l'éditeur signale un changement ***
var
  Msg: TLMMessage;
begin
  Msg.msg := M_CHANGEDMESSAGE; // assignation du message
  Dispatch(Msg); // répartition
  //Perform(M_CHANGEDMESSAGE, 0, 0); // envoi sans queue d'attente
end;

procedure TMainForm.btnLostClick(Sender: TObject);
// *** le bouton envoie un message perdu ***
var
  Msg: TLMMessage;
begin
  Msg.msg := M_LOSTMESSAGE; // assignation du message
  Dispatch(Msg); // répartition
  //Perform(M_LOSTMESSAGE, 0, 0); // envoi sans queue d'attente
end;

procedure TMainForm.Changed(var Msg: TLMMessage);
// *** changement récupéré avec numéro du message ***
begin
  mmoDisplay.Lines.Add('Changement ! Message : ' + IntToStr(Msg.msg));
end;

procedure TMainForm.DefaultHandler(var AMessage);
// *** message perdu ? ***
begin
  // transtypage de la variable sans type en TLMMessage
  if TLMMessage(AMessage).msg = M_LOSTMESSAGE then // perdu ?
    mmoDisplay.Lines.Add('Non Traité ! Message : ' +
      IntToStr(TLMMessage(AMessage).msg));
  inherited DefaultHandler(AMessage); // on hérite
end;

end.

```

Vous aurez remarqué que deux manières de notifier le changement sont utilisables :

- soit on affecte le numéro du message à une variable de type *LMessage* avant de la fournir en paramètre à *Dispatch* ;
- soit on utilise directement *Perform* qui court-circuite la queue d'attente.



Le transtypage de la variable *AMessage* est rendu nécessaire puisque cette dernière n'a pas de type : il se fait simplement en l'encadrant entre parenthèses du type voulu. On peut alors vérifier que la variable *Msg* de l'enregistrement du message correspond bien au message testé.

SURCHARGE DE METHODES

Vous avez vu qu'une méthode peut être déclarée de nouveau dans une classe enfant : une méthode statique sera écrasée alors qu'une méthode virtuelle pourra hériter de son ancêtre. Ce qui précède s'applique pour des méthodes dont les signatures, c'est-à-dire tous les paramètres et l'éventuelle valeur de retour, sont identiques. Mais que se passe-t-il dans le cas contraire ?

Si ses paramètres ou sa valeur de retour sont différents de celle de son ancêtre, la nouvelle méthode coexistera avec la méthode héritée. Vous pourrez par conséquent faire appel aux deux : l'implémentation activée sera celle qui correspondra aux paramètres invoqués. Ce sont vos méthodes qui seront polymorphiques puisqu'elles s'adapteront à vos souhaits



Notez qu'il est impossible dans une même classe de déclarer plusieurs méthodes portant le même nom et que les méthodes qui définissent les propriétés en lecture ou en écriture ne peuvent pas être surchargées.

Pour permettre la surcharge d'une méthode, il suffit d'ajouter la directive *overload* après sa déclaration. Lors de la surcharge d'une méthode virtuelle, il faut ajouter *reintroduce* à la fin de la nouvelle déclaration de la méthode, juste avant *overload*.

[Exemple PO-13]

Afin de tester cette possibilité, vous allez créer un nouveau projet qui procéder des additions sous différentes formes à partir d'une classe et de son enfant :

```
type
```

```
{ TAddition }
```

```
TAddition = class
```

```
  function AddEnChiffres(Nombre1, Nombre2: Integer): string;
```

```
  function AddVirtEnChiffres(Nombre1, Nombre2: Integer): string; virtual;
```

```
end;
```

```
{ TAdditionPlus }
```

```

TAdditionPlus = class(TAddition)
function AddEnChiffres(const St1, St2: string): string; overload;
function AddVirtEnChiffres(St1, St2: string): string; reintroduce; overload;
end;

```

La classe **TAddition** permet d'additionner deux entiers à partir de deux méthodes dont l'une est statique et la seconde virtuelle. **TAdditionPlus** est une classe dérivée de la première qui surcharge les méthodes héritées pour leur faire accepter des chaînes en guise de paramètres.

Leur définition reprend un système de trace grâce à des boîtes de dialogue qui vont s'afficher lorsqu'elles seront invoquées :

```

{ TAddition }

function TAddition.AddEnChiffres(Nombre1, Nombre2: Integer): string;
// *** addition avec méthode statique ***
begin
    Result := IntToStr(Nombre1 + Nombre2);
    MessageDlg('Entiers...', 'Addition d"entiers effectuée', mtInformation,
        [mbOK], 0);
end;

function TAddition.AddVirtEnChiffres(Nombre1, Nombre2: Integer): string;
// *** addition avec méthode virtuelle ***
begin
    Result := IntToStr(Nombre1 + Nombre2);
    MessageDlg('Entiers (méthode virtuelle)...', 'Addition d"entiers effectuée',
        mtInformation, [mbOK], 0);
end;

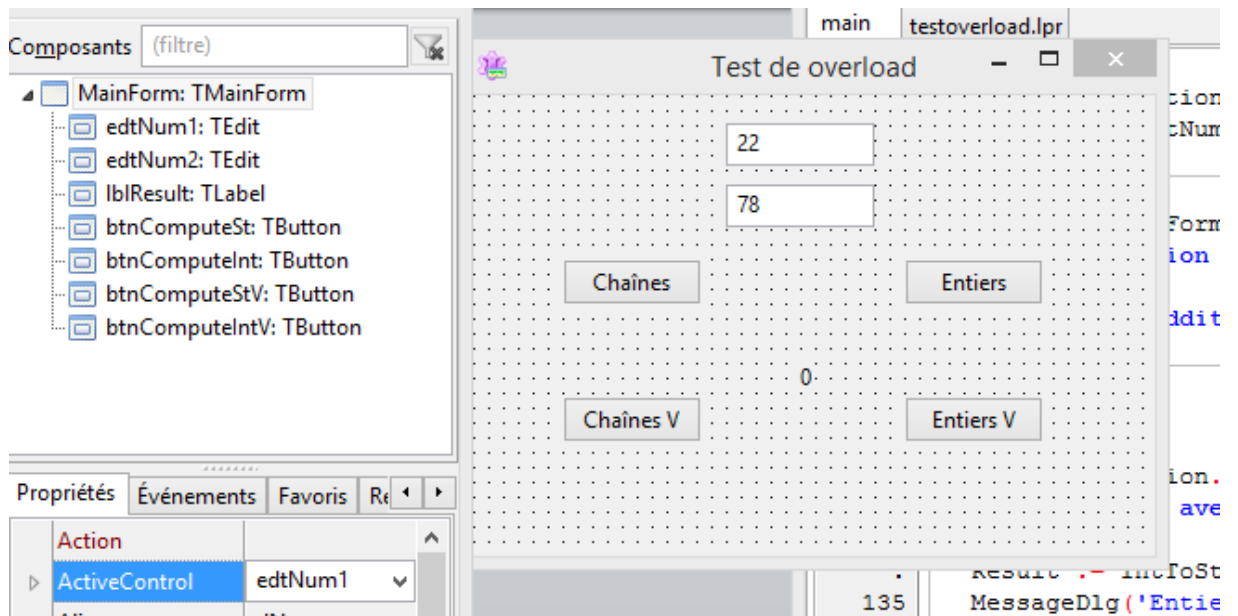
{ TAdditionPlus }

function TAdditionPlus.AddEnChiffres(const St1, St2: string): string;
// *** méthode statique surchargée ***
begin
    Result := IntToStr(StrToInt(St1) + StrToInt(St2));
    MessageDlg('Chaînes...', 'Addition à partir de chaînes effectuée',
        mtInformation, [mbOK], 0);
end;

function TAdditionPlus.AddVirtEnChiffres(St1, St2: string): string;
// *** méthode virtuelle surchargée ***
begin
    Result := inherited AddVirtEnChiffres(StrToInt(St1), StrToInt(St2));
    MessageDlg('Chaînes... (méthode virtuelle)',
        'Addition à partir de chaînes effectuée', mtInformation, [mbOK], 0);
end;

```

Le programme d'exploitation de ces deux classes est trivial puisqu'il se contente de proposer deux éditeurs *TEdit* qui contiendront les nombres à additionner, une étiquette *TLabel* pour le résultat de l'addition et quatre boutons *TButton* pour invoquer les quatre méthodes proposées :



Le code source qui l'accompagne ne devrait pas présenter de difficultés particulières :

implementation

```
{ $R *.lfm }

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche principale ***
begin
  Ad := TAdditionPlus.Create; // additionneur créé
end;

procedure TMainForm.btnComputeStClick(Sender: TObject);
// *** addition simple de chaînes ***
begin
  lblResult.Caption := Ad.AddEnChiffres(edtNum1.Text, edtNum2.Text);
end;

procedure TMainForm.btnComputeStVClick(Sender: TObject);
// *** addition de chaînes par méthode virtuelle ***
begin
  lblResult.Caption := Ad.AddVirtEnChiffres(edtNum1.Text, edtNum2.Text);
end;

procedure TMainForm.btnComputeIntClick(Sender: TObject);
// *** addition simple d'entiers ***
begin
  lblResult.Caption := Ad.AddEnChiffres(StrToInt(edtNum1.Text),
    StrToInt(edtNum2.Text));
```

```

end;

procedure TMainForm.btnComputeIntVClick(Sender: TObject);
// *** addition d'entiers par méthode virtuelle ***
begin
  lblResult.Caption := Ad.AddVirtEnChiffres(StrToInt(edtNum1.Text),
    StrToInt(edtNum2.Text));
end;

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

```



Il faut retenir que des méthodes portant le même nom mais aux signatures différentes sont parfaitement reconnues par le compilateur tant qu'elles ne sont pas déclarées dans la même interface d'une classe.

BILAN

Dans ce chapitre, vous aurez appris à :

- ✓ reconnaître et à manipuler les méthodes d'une classe sous toutes leurs formes : statiques, virtuelles, abstraites, de classe, statiques de classe et de messages ;
- ✓ maîtriser l'héritage et la surcharge des méthodes.

POO A GOGO : LES PROPRIETES

Objectifs : dans ce chapitre, vous allez tout apprendre sur les propriétés.

Sommaire : Définition – *Getter* et *setter* – propriétés et variables –

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

QU'EST-CE QU'UNE PROPRIETE ?

Une *propriété* définit l'attribut d'un objet et est avant tout un moyen d'accéder de manière contrôlée à un champ. Si une propriété a l'apparence d'une variable, elle n'en est pas une dans la mesure où elle n'occupe pas forcément de mémoire et que l'affectation d'une valeur à une propriété ou la lecture de sa valeur peuvent déclencher l'exécution d'une méthode. Par ailleurs, membre d'un descendant de *TComponent* et insérée dans une section *published*, une propriété deviendra visible dans l'inspecteur d'objet de **Lazarus**.

TRAVAILLER AVEC LES PROPRIETES

LECTURE ET ECRITURE D'UNE PROPRIETE : *GETTER* ET *SETTER*

Il est toujours possible de rendre *public* un champ quelconque. Ainsi, la définition d'une classe comme celle-ci est tout à fait correcte :

```
type
  TMyClass = class
  public
    fMyField : string;
  end;
```

L'utilisateur pourra alors affecter une chaîne au champ *fMyField* comme il agirait avec n'importe quelle variable. En supposant que *MyObject* soit une instance de *TMyClass*, les écritures suivantes seront correctes :

```
MyObject.fMyField := 'affectation correcte' ;
ShowMessage(MyObject.fMyField ) ;
```

Cependant, il est vivement conseillé d'éviter cet accès direct, car il est contraire à l'esprit de la POO. Il ne s'agit pas simplement de croyance ou de purisme, mais de profiter des avantages d'une encapsulation correcte !

Considérez par exemple le cas où le contenu du champ *fMyField* devrait toujours apparaître en majuscules dans votre programme. Comme vous le feriez dans le cadre de la programmation procédurale, il vous faudra remplacer toutes les occurrences de votre champ par une expression du genre :

```
UpperCase(MyObject.fMyField)
```

Vous conviendrez que dans un programme complexe et long, réparti dans de nombreuses unités, les risques d'erreurs sont importants. La réutilisation du code et sa maintenance seront aussi très difficiles et fastidieuses.

Les propriétés sont une réponse à ce genre de problème : une propriété permet de déclencher la méthode souhaitée (dans l'exemple en cours, une mise en majuscules). Une propriété, en plus d'accéder au champ visé, peut en effet effectuer les traitements particuliers nécessaires à l'objet auquel elle appartient. Quant à son invocation, elle restera inchangée dans l'ensemble du programme.

L'interface de la classe devrait au minimum ressembler à ceci :

```
type
  TMyClass = class
    strict private
      fMyField : string;
    public
      property MyField: string read fMyField write fMyField;
    end;
```

Une propriété est introduite par le mot réservé *property* suivi de l'identificateur de la propriété, de son type et d'au moins un des deux mots réservés *read* et *write*, eux-mêmes suivis du nom d'un champ ou d'une méthode d'accès.

Le gain est nul à ce niveau puisque l'accès se fait directement grâce au nom d'un champ interne, sinon que ce champ est protégé puisqu'il est devenu inaccessible depuis l'extérieur de l'objet.

Un changement décisif consistera à utiliser une méthode de lecture (*getter*) et/ou une méthode d'écriture (*setter*) :

```
type
  TMyClass = class
    strict private
      fMyField : string;
      function GetMyField: string;
      procedure SetMyField(const AValue: string);
    public
      property MyField: string read GetMyField write SetMyField;
    end;
```

À présent, en supposant toujours que *MyObject* soit une instance de *TMyClass*, les écritures suivantes seront correctes :

```
MyObject.MyField := 'affectation correcte' ;  
ShowMessage(MyObject.MyField ) ;
```



Quelques conventions sont utilisées de manière à rendre le code source plus lisible. Bien qu'elles n'aient pas de caractère obligatoire, elles sont conseillées :

- les champs internes ont leur identificateur précédé de la lettre « f » (ou « F ») pour l'anglais *field* ;
- une méthode *getter* porte un nom au préfixe *Get* ;
- une méthode *setter* porte un nom au préfixe *Set*.

Les définitions des deux méthodes d'accès pourraient être celles-ci :

```
{ TMyClass }  
  
function TMyClass.GetMyField: string;  
begin  
    Result := fMyField;  
end;  
  
procedure TMyClass.SetMyField(const AValue: string);  
begin  
    fMyField := AValue ;  
end;
```

Pour le moment, de telles complications ne sont pas justifiées, mais si vous revenez à votre programme complexe, avec ses nombreuses occurrences du champ *MyField* et ses un peu moins nombreuses unités, la transformation du champ en chaîne en majuscules n'exigera que la modification d'une unique ligne de code :

```
procedure TMyClass.SetMyField(const AValue: string);  
begin  
    fMyField := UpperCase(AValue) ;  
end;
```

En fait, tous les traitements légaux sont permis au sein de ces méthodes d'accès.

Pour montrer l'efficacité des propriétés, vous allez créer une classe chargée de transformer un entier en chaîne de caractères en tenant compte des règles complexes d'accord en français, en particulier pour 80 et 100 qui prennent un « s » lorsqu'ils ne sont pas suivis d'un autre ordinal et pour le tiret employé ou non systématiquement (suivant les... écoles !).

[Exemple PO-14]

Voici l'interface de cette classe :

```
type
{ TValue2St }

TValue2St = class
strict private
  fValue: Integer;
  fStValue: string;
  fWithDash: Boolean;
  fDash: Char;
  procedure SetWithDash(AValue: Boolean);
  procedure SetValue(const AValue: Integer);
protected
  function Digit2St(const AValue: Integer): string; virtual;
  function Decade2St(const AValue: Integer; Plural: Boolean = True): string; virtual;
  function Hundred2St(const AValue: Integer; Plural: Boolean = True): string; virtual;
  function Thousand2St(const AValue: Integer): string; virtual;
  function Million2St(const AValue: Integer): string; virtual;
public
  constructor Create;
  property WithDash: Boolean read fWithDash write SetWithDash;
  property Value: Integer read fValue write SetValue;
  property StValue: string read fStValue;
end;
```

Cette classe appelle les remarques suivantes :

- la section *strict private* abrite les champs et leurs méthodes d'accès : ils sont donc inaccessibles à l'extérieur de la classe ;
- la section *protected* comprend les méthodes qui transforment un entier en chaîne de caractères : cette section ainsi que l'emploi de *virtual* se justifient par le fait que des classes qui descendraient de *TValue2St* auraient probablement à modifier ces méthodes afin d'obtenir d'autres résultats ;
- la section *public* comprend un constructeur qui initialisera des données et trois propriétés : *WithDash* qui déterminera l'emploi systématique ou non du tiret, *Value* qui gèrera la valeur entière de travail, et *StValue* pour la chaîne de retour ;
- les propriétés *WithDash* et *Value* accèdent directement aux champs qui les concernent, mais utilisent une méthode pour les définir : *WithDash* modifiera automatiquement la chaîne si elle est elle-même modifiée tandis que *Value* profitera de sa modification pour construire la chaîne correspondante ;
- la propriété *StValue* est en lecture seule : elle accède directement au champ *fStValue* qui aura été calculé en interne ;
- les méthodes *Decade2St* et *Hundred2St* ont toutes les deux un paramètre *Plural* défini à *True* par défaut : ce paramètre précisera s'il faut ajouter un « s » et simplifiera l'appel de la fonction si c'est le cas en économisant un paramètre (*Decade2St*(45) est équivalent à *Decade2St*(45, True)).

Voici l'implémentation de cette classe :

implementation

const

```
CDigit : array[0..9] of string = ('zéro','un','deux','trois','quatre',  
                                   'cinq','six','sept','huit','neuf');  
CNum1: array[10..19] of string = ('dix','onze','douze','treize','quatorze',  
                                   'quinze','seize','dix-sept','dix-huit','dix-neuf');  
CNum2: array[1..7] of string = ('vingt','trente','quarante','cinquante',  
                                 'soixante','soixante-dix','quatre-vingt');  
CHundred = 'cent';  
CThousand = 'mille';  
CMillion = 'million';
```

```
{ TValue2St }
```

```
procedure TValue2St.SetWithDash(AValue: Boolean);
```

```
// *** tiret obligatoire ou non ***
```

```
begin
```

```
  if fWithDash = AValue then
```

```
    Exit;
```

```
  fWithDash := AValue;
```

```
  if fWithDash then
```

```
    fDash := '-'
```

```
  else
```

```
    fDash := '';
```

```
  Value := Value; // force la mise à jour du texte associé au nombre
```

```
end;
```

```
procedure TValue2St.SetValue(const AValue: Integer);
```

```
// *** nouvelle valeur ***
```

```
begin
```

```
  fValue := AValue;
```

```
  fStValue := Million2St(fValue); // transformation de l'entier
```

```
end;
```

```
function TValue2St.Digit2St(const AValue: Integer): string;
```

```
// *** chiffres en lettres ***
```

```
begin
```

```
  Result := CDigit[AValue];
```

```
end;
```

```
function TValue2St.Decade2St(const AValue: Integer; Plural: Boolean = True): string;
```

```
// *** dizaines en lettres ***
```

```
begin
```

```
  case AValue of
```

```
    0..9: Result := Digit2St(AValue);
```

```
    10..19: Result := CNum1[AValue];
```

```
    20, 30, 40, 50, 60, 70: Result := CNum2[(AValue div 10) - 1];
```

```
    21, 31, 41, 51, 61: Result := CNum2[(AValue div 10) - 1] + '-et-un';
```

```
    22..29: Result := CNum2[1] + '-' + Digit2St(AValue - 20);
```

```
    32..39: Result := CNum2[2] + '-' + Digit2St(AValue - 30);
```

```
    42..49: Result := CNum2[3] + '-' + Digit2St(AValue - 40);
```

```
    52..59: Result := CNum2[4] + '-' + Digit2St(AValue - 50);
```

```
    62..69: Result := CNum2[5] + '-' + Digit2St(AValue - 60);
```

```
    71: Result := 'soixante-et-onze';
```

```
    72..79: Result := CNum2[5] + '-' + CNum1[AValue - 60];
```

```
    80: if Plural then // cas de 80 avec ou sans s
```

```

        Result := CNum2[7] + 's'
    else
        Result := CNum2[7];
    81..89: Result := CNum2[7] + '-' + Digit2St(AValue - 80);
    90..99: Result := CNum2[7] + '-' + CNum1[AValue - 80];
end;
end;

function TValue2St.Hundred2St(const AValue: Integer; Plural: Boolean = True): string;
// *** centaines en lettres ***
begin
    if (AValue < 100) then
        Result := Decade2St(AValue, Plural)
    else
        if (AValue = 100) then
            Result := CHundred
        else
            if (AValue < 200) then
                Result := CHundred + fDash + Decade2St(AValue - 100, Plural)
            else
                if ((AValue mod 100) = 0) then
                    Result := Decade2St(AValue div 100, False) + fDash + CHundred
                else
                    Result := Decade2St(AValue div 100, False) + fDash + CHundred + fDash +
                        Decade2St(AValue - 100 * (AValue div 100));
                if Plural and ((AValue mod 100) = 0) and (AValue <> 100) and (AValue <> 0) then
                    Result := Result + 's'; // cas de 100 multiplié
                end;
            end;
        end;

function TValue2St.Thousand2St(const AValue: Integer): string;
// *** milliers en lettres ***
begin
    if AValue < 1000 then
        Result := Hundred2St(AValue)
    else
        if AValue = 1000 then
            Result := CThousand + fDash
        else
            if AValue < 2000 then
                Result := CThousand + fDash + Hundred2St(AValue - 1000)
            else
                if (AValue mod 1000) = 0 then
                    Result := Hundred2St(AValue div 1000, False) + fDash + CThousand + fDash
                else
                    Result := Hundred2St(AValue div 1000, False) + fDash + CThousand + fDash
                        + Hundred2St(AValue - 1000 * (AValue div 1000));
                end;
            end;

function TValue2St.Million2St(const AValue: Integer): string;
// *** millions en lettres ***
begin
    if AValue = 1000000 then
        Result := CDigit[1] + fDash + CMillion
    else
        if AValue < 1000000 then
            Result := Thousand2St(AValue)
        else
            if AValue < 2000000 then
                Result := CDigit[1] + fDash + CMillion + fDash + Thousand2St(AValue - 1000000)
            end;
        end;
    end;
end;

```

```

else
if(AValue mod 1000000) = 0 then
    Result := Thousand2St(AValue div 1000000)+ fDash + CMillion + 's'
else
    Result := Thousand2St(AValue div 1000000)+ fDash + CMillion + 's' + fDash +
        Thousand2St(AValue - 1000000 * (AValue div 1000000));
end;

constructor TValue2St.Create;
// *** création de l'objet ***
begin
    inherited Create; // on hérite
    fDash := '';
end;

```

La transformation d'un nombre en chaîne a été décomposée en cinq méthodes travaillant respectivement sur les chiffres, les dizaines, les centaines, les milliers et les millions. Cette décomposition de l'entier à traiter évite les méthodes trop longues et complexes : *Million2St* va déléguer le travail de précision à ses consœurs.

L'essentiel est de constater que derrière une simple affectation se cache souvent un ensemble complexe d'instructions :

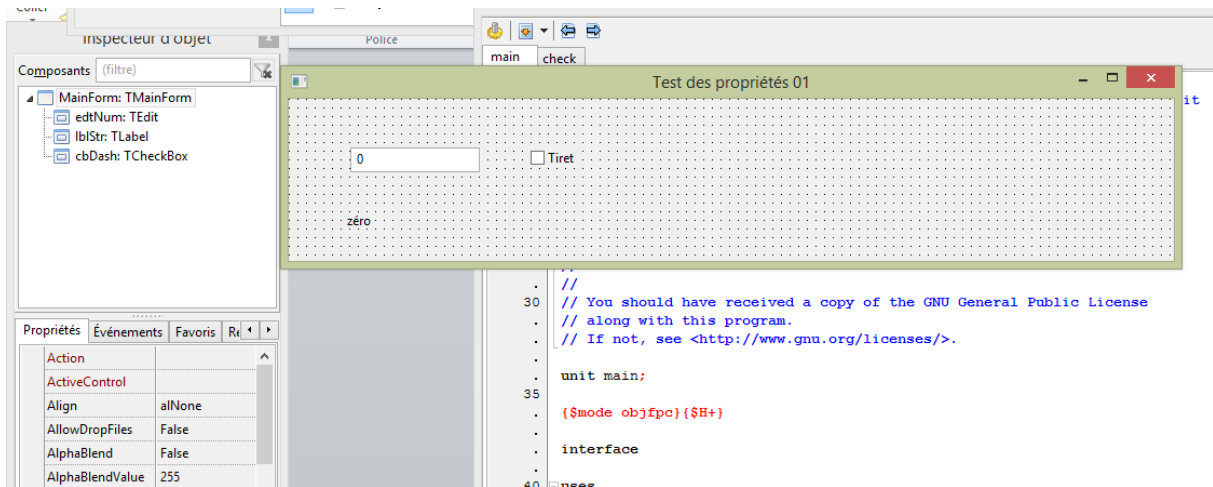
```
MyObject.Value := 123456 ;
```

Apparemment, *Value* de l'objet *MyObject* prend la valeur 123456. En réalité, une série de calculs construira la chaîne « cent vingt-trois mille quatre cent cinquante-six ». En passant *WithDash* à *True*, le résultat serait « cent-vingt-trois-mille-quatre-cent-cinquante-six » sans aucune autre intervention de l'utilisateur.

Afin de tester votre unité baptisée *check*, procédez comme suit :

- créez une nouvelle application ;
- enregistrez les squelettes créés automatiquement par **Lazarus** sous les noms suivants : *project1.lpi* sous *testproperties01.lpi* et *unit1.pas* sous *main.pas* ;
- ajoutez l'unité *check* à la clause *uses* de l'interface de *main.pas* ;
- changez *Caption* de la fenêtre principale en « Test des propriétés 01 » ;

- ajoutez un *TEdit*, un *TCheckBox* et un *TLabel* à votre fiche principale en les renommant respectivement *edtNum*, *cbDash* et *lblStr* :



L'éditeur prendra la valeur de l'entier à transformer, la case à cocher indiquera si l'emploi des tirets est obligatoire ou non, tandis que l'étiquette contiendra la chaîne calculée.

Continuez votre travail ainsi :

- ajoutez un champ *Value* de type *TValue2St* dans la section *private* de l'interface de la fiche ;
- créez les gestionnaires *OnCreate* et *OnDestroy* de la fiche grâce à l'inspecteur d'objet ;
- faites de même avec les gestionnaires *OnChange* de *edtNum* et *cbDash* ;
- complétez l'unité *main* ainsi :

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  check; // unité ajoutée

type

  { TMainForm }

  TMainForm = class(TForm)
    cbDash: TCheckBox;
    edtNum: TEdit;
    lblStr: TLabel;
    procedure cbDashChange(Sender: TObject);
    procedure edtNumChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { private declarations }
```

```

    Value: TValue2St; // champ de travail
public
    { public declarations }
end;

var
    MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.edtNumChange(Sender: TObject);
// *** l'éditeur change ***
var
    Li: Integer;
begin
    if edtNum.Text = '' then // chaîne vide ?
        Exit;
    if TryStrToInt(edtNum.Text, Li) then // nombre entier correct ?
        begin
            Value.Value := Li; // la propriété fait son travail !
            lblStr.Caption:= Value.StValue; // l'étiquette contient la chaîne
        end
    else
        ShowMessage('"' + edtNum.Text + '" n'est pas un nombre entier correct !');
    end;

    procedure TMainForm.cbDashChange(Sender: TObject);
    // *** avec ou sans tirets ***
    begin
        Value.WithDash := cbDash.Checked;
        lblStr.Caption:= Value.StValue; // mise à jour de l'étiquette
    end;

    procedure TMainForm.FormCreate(Sender: TObject);
    // *** création de la fiche ***
    begin
        Value := TValue2St.Create;
    end;

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

end.

```

Vous avez ainsi créé une instance de *TValue2St* dans la méthode *FormCreate* sans oublier de la libérer dans la méthode *FormDestroy*. Par ailleurs, à chaque fois qu'une modification est apportée à *edtNum*, la validité de l'entrée est vérifiée grâce à *TryStrToInt*, une fonction de la RTL qui essaye de transformer une chaîne en entier : si

cette transformation réussit, l'entier obtenu est affecté à la propriété *Value* de l'instance de *TValue2St* puis la chaîne calculée affichée dans l'étiquette¹⁹.

[Exemple PO-15]

À présent, un locuteur belge fera remarquer que « septante », « octante » et « nonante » sont des facilités auxquelles il ne voudrait pour rien au monde reconcer. Pour satisfaire ses besoins, il faudrait compléter le tableau *CNum2* :

```
CNum2: array[1..10] of string = ('vingt','trente','quarante','cinquante',  
                                'soixante','soixante-dix','quatre-vingt','septante','octante','nonante');
```

Les modifications à apporter à la classe *TValue2St* seraient minimales :

```
type  
  TValue2StBelg = class(TValue2St)  
    protected  
      function Decade2St(const AValue: Integer; Plural: Boolean = True): string; override;  
    end;
```

Quant à l'implémentation de la méthode surchargée, elle serait bien plus simple que celle de l'ancêtre :

```
function TValue2StBelg.Decade2St(const AValue: Integer; Plural: Boolean = True): string;  
// *** dizaines en lettres – version belge ***  
begin  
  Result := inherited Decade2St(AValue, Plural); // on hérite de la valeur de l'ancêtre  
  case AValue of // on ne modifie que les nouvelles valeurs  
    70, 80, 90 : Result := CNum2[(AValue mod 10) + 1] ;  
    71, 81, 91 : Result := CNum2[(AValue mod 10) + 1] + 'et-un';  
    72..79 : Result := CNum2[8] + '-' + Digit2St(AValue - 70);  
    82..89 : Result := CNum2[9] + '-' + Digit2St(AValue - 80);  
    92..99 : Result := CNum2[10] + '-' + Digit2St(AValue - 90);  
  end;  
end;
```

Enfin, dans le programme principal, il faudrait déclarer une variable de type *TValue2StBelg* au lieu d'une variable de type *TValue2St* :

```
TMainForm = class(TForm)  
  // [...]  
  private  
    { private declarations }  
    Value: TValue2StBelg; // champ de travail modifié  
  public  
    { public declarations }  
end;
```

¹⁹ Dans un projet plus cohérent, on aurait intérêt à inclure ces tests dans la classe elle-même et sans doute à prévoir une entrée du nombre sous forme de chaîne. Les simplifications sont ici à visée pédagogique.

L'instanciation de la classe devrait suivre ce nouveau type :

```
procedure TMainForm.FormCreate(Sender: TObject);  
// *** création de la fiche ***  
begin  
  Value := TValue2StBelg.Create; // nouvelle création  
end;
```

À peu de frais, vous aurez une version belge de l'application !

PROPRIETES ET VARIABLES

Mais revenons un peu en arrière. Une ligne du code de la méthode *SetValue* de l'unité *check* vous aura peut-être surpris :

```
Value := Value;
```

En temps ordinaire, avec une variable, cette affectation n'aurait aucun sens : pourquoi affecter à une variable la valeur qu'elle possède déjà ? Il en va différemment avec les propriétés : l'affectation à *Value* va activer la méthode *SetValue* qui va recalculer la valeur de la chaîne et l'affecter au champ *fStValue*. Pour des raisons de lisibilité, une telle écriture n'est pas courante, mais elle illustre bien la différence entre une variable et une propriété.

En fait, une propriété n'occupe pas forcément d'espace en mémoire. Elle n'est même pas forcément en rapport avec un champ interne. On peut par exemple imaginer une propriété en lecture seule qui renverrait un entier tiré au hasard :

```
{ TMyClass }  
  
TMyClass = class  
strict private  
  function GetMyProp: Integer;  
published  
  property MyProp: Integer read GetMyProp;  
end;  
// [...]  
implementation  
  
function TMyClass.GetMyProp: Integer;  
// *** renvoie un entier de 0 à 99 ***  
begin  
  Result := Random(100);  
end;
```



Une conséquence de ce mécanisme est qu'une propriété ne peut pas servir de paramètre de type *var* dans une routine. Pas plus vous ne pourrez utiliser *@* ou modifier une propriété avec *Inc* ou *Dec*.

REDEFINITION D'UNE PROPRIETE

[Exemple PO-16]

PROPRIETE PAR DEFAULT (DEFAULT NODEFAULT)

[Exemple PO-17]

LES INFORMATIONS DE STOCKAGE

[Exemple PO-18]

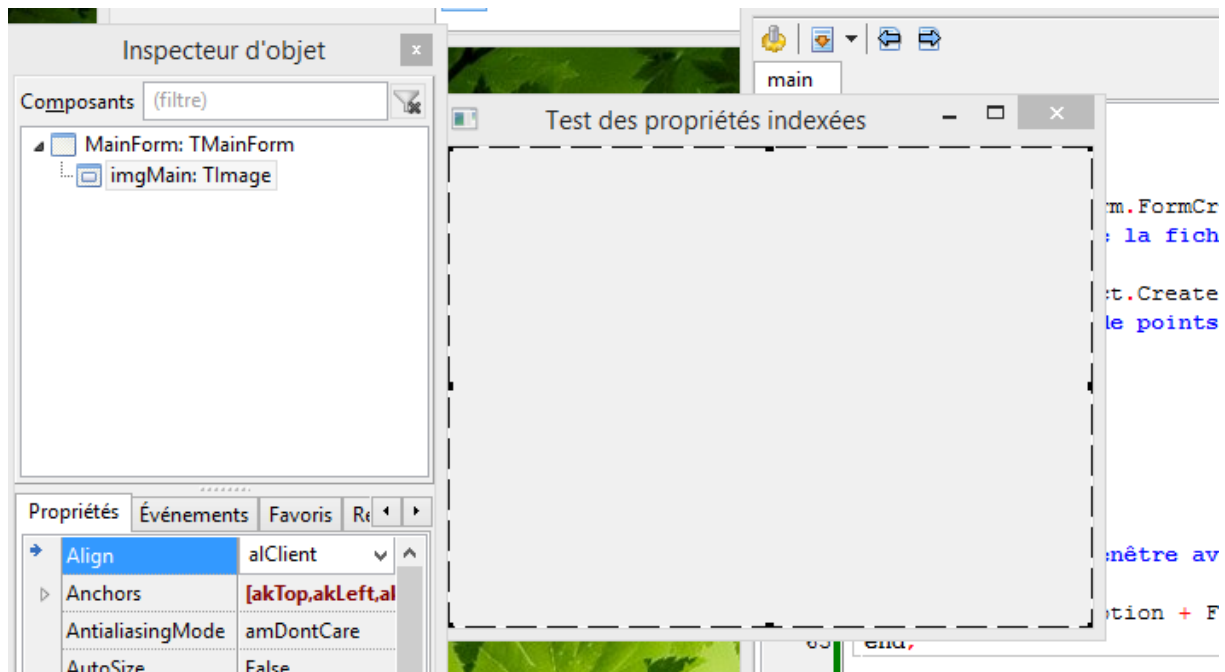
LES PROPRIETES INDEXEES

Il est possible de lire et d'écrire plusieurs propriétés à partir d'une même méthode à condition qu'elles soient du même type. Dans ce cas, chaque déclaration de type de propriété sera suivie du mot réservé *index* lui-même suivi d'un entier précisant le rang de l'index. Les méthodes *getter* et *setter* seront forcément une fonction et une procédure.

[Exemple PO-19]

Pour tester cette possibilité, procédez comme suit :

- créez une nouvelle application baptisée *testindexedproperties* dont l'unité principale sera renommée *main* ;
- renommez la fiche principale *MainForm* ;
- modifiez la propriété *Caption* de cette fiche pour qu'elle affiche « Test des propriétés indexées » ;
- déposez un composant *TImage* sur la fiche principale, baptisez-le *imgMain* et changez sa propriété *Align* à *alClient* :



- créez un gestionnaire OnCreate pour la fiche principale et un gestionnaire OnResize pour le composant *TImage* ;

- complétez le code ainsi :

type

{ TMyRect }

```
TMyRect = class
strict private
  fValues: array[0..3] of Integer;
  function GetValue(AIndex: Integer): Integer;
  procedure SetValue(AIndex: Integer; AValue: Integer);
public
  property Left: Integer index 0 read GetValue write SetValue;
  property Top: Integer index 1 read GetValue write SetValue;
  property Width: Integer index 2 read GetValue write SetValue;
  property Height: Integer index 3 read GetValue write SetValue;
end;
```

{ TMainForm }

```
TMainForm = class(TForm)
  imgMain: TImage;
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure imgMainResize(Sender: TObject);
private
  { private declarations }
  MyRect: TMyRect;
public
  { public declarations }
end;
```

var

MainForm: TMainForm;

implementation

{ \$R *.lfm }

{ TMainForm }

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  MyRect := TMyRect.Create; // rectangle créé
  // affectation de points
  with MyRect do
    begin
      Left := 50;
      Top := 30;
      Width := 320;
      Height := 250;
    end;
  // en-tête de fenêtre avec les coordonnées
  with MyRect do
    Caption := Caption + Format(' ( %d, %d, %d, %d)', [Left, Top, Width, Height]);
end;
```

```

procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
    MyRect.Free; // libération du rectangle
end;

procedure TMainForm.imgMainResize(Sender: TObject);
// *** dessin ***
begin
    // couleur bleue
    imgMain.Canvas.Brush.Color := clBlue;
    // dessin du rectangle
    with MyRect do
        imgMain.Canvas.Rectangle(Left, Top, Width - Left, Height - Top);
end;

{ TMyRect }

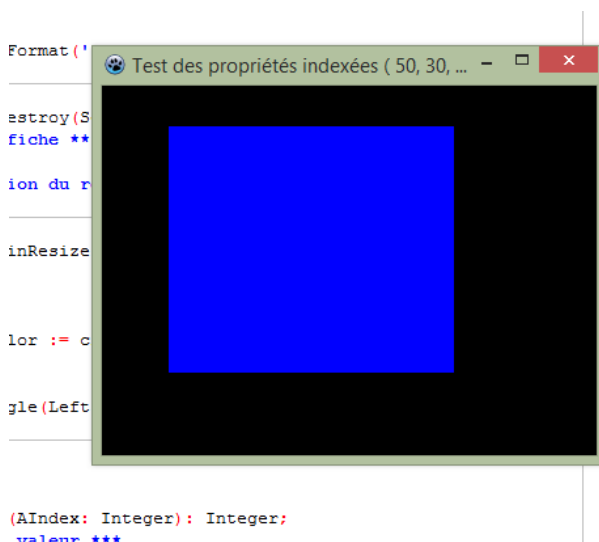
function TMyRect.GetValue(AIndex: Integer): Integer;
// *** récupération d'une valeur ***
begin
    Result := fValues[AIndex];
end;

procedure TMyRect.SetValue(AIndex: Integer; AValue: Integer);
// *** établissement d'une valeur ***
begin
    fValues[AIndex] := AValue;
end;

end.

```

L'exécution du programme donne ceci :



Dans l'exemple, l'index renvoie à celui d'un tableau, mais ce fait n'a rien d'obligatoire. Il aurait été possible de déclarer quatre champs privés et d'y accéder

depuis une seule méthode grâce à une construction de type *case...of*. La déclaration aurait ressemblé à ceci :

```
strict private
  fLeft, fTop, fWidth, fHeight : Integer ;
```

Dans ce cas, les méthodes d'accès auraient eu cette allure :

```
{ TMyRect }

function TMyRect.GetValue(AIndex: Integer): Integer;
// *** récupération d'une valeur ***
begin
  case AIndex of
    0: Result := fLeft ;
    1: Result := fTop ;
    2: Result := fWidth ;
    3: Result := fHeight;
  end;

  procedure TMyRect.SetValue(AIndex: Integer; AValue: Integer);
  // *** établissement d'une valeur ***
  begin
    case AIndex of
      0: fLeft := AValue ;
      1: fTop := AValue;
      2: fWidth := AValue;
      3: fHeight := AValue;
    end;
```

LES TABLEAUX DE PROPRIETES

[Exemple PO-20]

PROPRIETES DE CLASSE

Comme les méthodes de classe, les *propriétés de classe* sont accessibles sans référence d'objet et doivent être déclarées avec **class** en premier lieu :

```
class property Version : Integer read fVersion write SetVersion ;
class property SubVersion: Real read fSubBersion write SetSubVersion;
```

Les méthodes et les champs auxquels la propriété fait référence doivent être des méthodes statiques de classe et des champs de classe. La déclaration d'une classe comprenant les deux propriétés de classe définies plus haut pourrait être :

```
TMyClass = class
strict private
  class var
    fVersion: Integer ;
    fSubVersion: Real;
  class procedure SetVersion(const AValue: Integer); static;
  class procedure SetSubVersion(const AValue: Real); static;
```

```
// [...]
public
// [...]
class property Version : Integer read fVersion write SetVersion ;
class property SubVersion: Real read fSubVersion write SetSubVersion;
end;
```

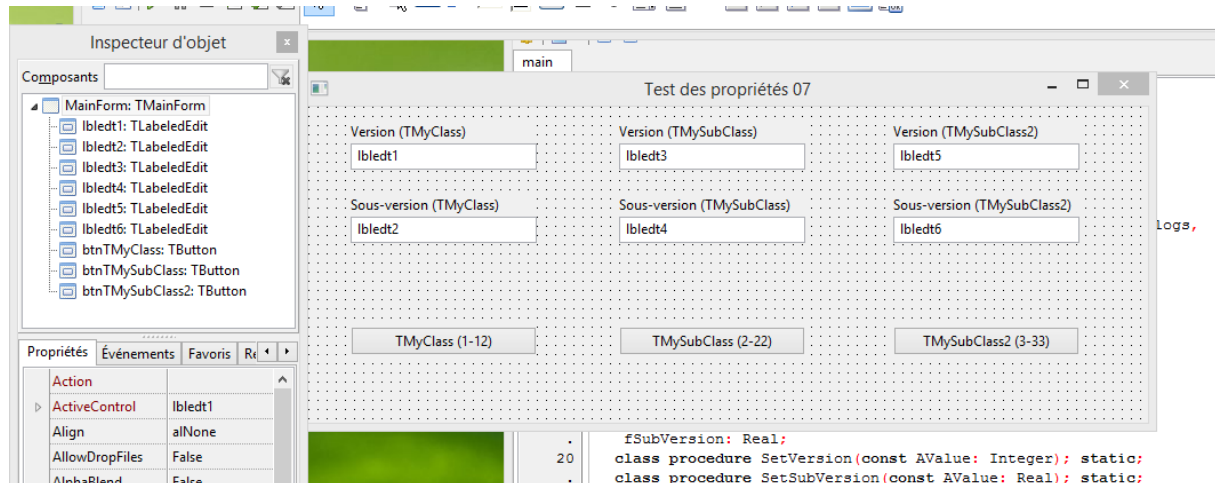
Une propriété de classe étant toujours associée à une classe particulière, il est impossible d'utiliser comme *setter* ou *getter* une méthode de classe non statique : en cas de surcharge de la méthode, la propriété de classe n'aurait plus accès aux données qui lui sont nécessaires. Par conséquent, il est nécessaire de préciser *static* à la fin de la ligne de déclaration des méthodes utilisées par une propriété.

[Exemple PO-21]

Afin de tester les propriétés de classe, vous allez créer un nouveau programme baptisé *testproperties07.lpi*. Les objectifs seront de montrer que les classes n'ont pas à être instanciées pour être accessibles *via* les propriétés de classe et que ces dernières réagissent bien de manière statique, suivant les derniers changements opérés dans les champs de classe.

Procédez donc comme suit :

- modifiez la fiche principale de telle façon qu'elle ressemble à ceci (le composant *TLabelEdit* est présent dans la page « Additional » de la palette) :



- créez les gestionnaires *OnCreate* de la fiche principale et *OnClick* des trois boutons ;
- reproduisez le code suivant dans l'unité (*main*) de la fiche principale (*MainForm*) :

```
type
{ TMyClass }

TMyClass = class
strict private
class var
```

```

fVersion : Integer ;
fSubVersion: Real;
class procedure SetVersion(const AValue: Integer); static;
class procedure SetSubVersion(const AValue: Real); static;
// [...]
public
// [...]
class property Version : Integer read fVersion write SetVersion;
class property SubVersion: Real read fSubVersion write SetSubVersion;
end;

{ TMySubClass }

TMySubClass = class(TMyClass)
strict private
  class procedure SetSubVersion(const AValue: Real); static;
  class procedure SetVersion(const AValue: Integer); static;
public
  class property Version : Integer read fVersion write SetVersion ;
  class property SubVersion: Real read fSubVersion write SetSubVersion;
end;

{ TMySubClass2 }

TMySubClass2 = class(TMyClass)
strict private
  class procedure SetSubVersion(const AValue: Real); static;
  class procedure SetVersion(const AValue: Integer); static;
public
  class property Version : Integer read fVersion write SetVersion ;
  class property SubVersion: Real read fSubVersion write SetSubVersion;
end;

{ TMainForm }

TMainForm = class(TForm)
  btnTMyClass: TButton;
  btnTMySubClass: TButton;
  btnTMySubClass2: TButton;
  lbledt1: TLabelEdit;
  lbledt2: TLabelEdit;
  lbledt3: TLabelEdit;
  lbledt4: TLabelEdit;
  lbledt5: TLabelEdit;
  lbledt6: TLabelEdit;
  procedure btnTMyClassClick(Sender: TObject);
  procedure btnTMySubClassClick(Sender: TObject);
  procedure btnTMySubClass2Click(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  { private declarations }
  procedure UpdateEditors;
public
  { public declarations }
end;

var
  MainForm: TMainForm;

```


implementation

```
{ $R *.lfm }

{ TMySubClass2 }

class procedure TMySubClass2.SetSubVersion(const AValue: Real);
// *** numéro de sous-version (autre sous-classe) ***
begin
  fSubVersion := AValue / 100;
end;

class procedure TMySubClass2.SetVersion(const AValue: Integer);
// *** numéro de version (autre sous-classe) ***
begin
  fVersion := AValue * 100;
end;

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  TMyClass.Version := 1; // initialisation des propriétés
  TMyClass.SubVersion := 11;
  UpdateEditors; // mise à jour
end;

procedure TMainForm.btnTMyClassClick(Sender: TObject);
// *** choix de TMyClass ***
begin
  TMyClass.Version := 1;
  TMyClass.SubVersion := 11;
  UpdateEditors;
end;

procedure TMainForm.btnTMySubClassClick(Sender: TObject);
// *** choix de TMySubClass ***
begin
  TMySubClass.Version := 2;
  TMySubClass.SubVersion := 22;
  UpdateEditors;
end;

procedure TMainForm.btnTMySubClass2Click(Sender: TObject);
// *** choix de TMySubClass2 ***
begin
  TMySubClass2.Version := 3;
  TMySubClass2.SubVersion := 33;
  UpdateEditors;
end;

procedure TMainForm.UpdateEditors;
// *** mise à jour des éditeurs ***
begin
  lbledt1.Caption := IntToStr(TMyClass.Version);
  lbledt2.Caption := FloatToStr(TMyClass.SubVersion);
  lbledt3.Caption := IntToStr(TMySubClass.Version);
  lbledt4.Caption := FloatToStr(TMySubClass.SubVersion);
```

```

lbledt5.Caption := IntToStr(TMySubClass2.Version);
lbledt6.Caption := FloatToStr(TMySubClass2.SubVersion);
end;

{ TMySubClass }

class procedure TMySubClass.SetSubVersion(const AValue: Real);
// *** numéro de version (sous-classe) ***
begin
  fSubVersion := AValue / 10;
end;

class procedure TMySubClass.SetVersion(const AValue: Integer);
// *** numéro de sous-version (sous-classe) ***
begin
  fVersion := AValue * 10;
end;

{ TMyClass }

class procedure TMyClass.SetVersion(const AValue: Integer);
// *** numéro de version ***
begin
  fVersion := AValue;
end;

class procedure TMyClass.SetSubVersion(const AValue: Real);
// *** numéro de sous-version ***
begin
  fSubVersion := AValue;
end;

end.

```

Trois classes sont définies (*TMyClass*, *TMySubClass*, *TMySubClass2*) dont l'une est l'ancêtre des deux autres (*TMyClass*). Chacune de ses classes définit ses propres propriétés de classe et ses propres méthodes statiques de classe associées. On s'aperçoit à l'exécution qu'il n'est jamais nécessaire d'instancier les classes et que c'est toujours le dernier changement d'une propriété qui est pris en compte, y compris entre classes sœurs. C'est aussi uniquement à travers les méthodes de la classe invoquée que les propriétés sont modifiées²⁰.



Les propriétés de classes ne peuvent ni être du type *published* ni avoir de définition *stored* ou *default*.

EXEMPLE D'UTILISATION DES METHODES ET PROPRIETES DE CLASSE

Fort de votre nouvelle expérience, vous devriez comprendre le fonctionnement du programme qui va suivre : imaginez une application qui utiliserait une série de descendants de *TMemo* pour, par exemple, gérer telle ou telle langue.

²⁰ Afin de bien se rendre compte du phénomène, chaque méthode a une particularité : valeur laissée telle quelle, multipliée ou divisée par 10, multipliée ou divisée par 100.

En programmation procédurale, vous pourriez créer des tests avec *if...then...else* pour savoir quel descendant utiliser. Avec la POO, une option plus souple et plus sûre est disponible : les classes s'enregistreront elles-mêmes de manière à éviter toute erreur et le corps de la fiche principale ne sera pas affecté par l'ajout d'un nouvel objet. Mieux : la fiche principale ne saura rien des classes instanciées en dehors du fait que ce sont des descendantes d'une classe de base.

[Exemple PO-22]

Le principe est de créer une classe dérivée de la classe à créer dynamiquement. Procédez ainsi :

- créez un nouveau programme nommée *testclassmethods* ;
- sauvegardez la fiche principale sous le nom *MainForm* ;
- ajoutez une nouvelle unité au projet et baptisez-la *submemos* ;
- référencez cette unité dans la clause *uses* de la section *interface* de la fiche principale.

L'ancêtre des classes à utiliser sera défini ainsi dans l'unité *submemos* :

```
{ TSubmemo }

TSubMemo = class(TMemo)
strict private
  class var
    fmemos: TList;
protected
  class procedure NewMemo;
public
  class constructor Create;
  class destructor Destroy;
  class property Memos: TList read fMemos;
  class function ClassText: string; virtual;
  class function Version: Integer; virtual;
  class function ClassColor: TColor; virtual;
end;
```

La propriété de classe *Memos* est associée à une variable de classe qui est une liste : elle comprendra toutes les classes qui se seront enregistrées. La liste *TList* sera créée et détruite automatiquement grâce à la déclaration d'un constructeur de classe *Create* et d'un destructeur de classe *Destroy*²¹.

ClassText, *Version* et *ClassColor* sont des méthodes de classe qui renverront respectivement une chaîne de caractères, un entier et une couleur. Le choix s'est porté sur des méthodes afin de bénéficier des avantages de la virtualité : des propriétés auraient été identiques pour toutes les classes définies.

²¹ Pour rappel : ce constructeur et ce destructeur sont invoqués respectivement avant la section *initialization* et après la section *finalization*, sans aucun appel explicite.

La méthode *NewMemo* enregistrera la classe. Comme aucun appel à un constructeur pour instancier cette classe n'aura lieu, il faudra prévoir d'invoquer cette méthode dans la partie *initialization* de l'unité abritant la classe.

L'implémentation ne pose pas vraiment de problèmes :

```
{ TSubMemo }

class procedure TSubMemo.NewMemo;
// *** nouveau mémo ***
begin
  // mémo non enregistré ?
  if (fMemos.IndexOf(Self) = -1) then
    // on l'ajoute
    fMemos.Add(Self);
end;

class constructor TSubMemo.Create;
// *** création ***
begin
  // on crée la liste de travail
  fMemos := TList.Create;
end;

class destructor TSubMemo.Destroy;
// *** destruction ***
begin
  // la liste de travail est libérée
  fMemos.Free;
end;

class function TSubMemo.ClassText: string;
// *** légende 0 ***
begin
  Result := 'Mémo ANCETRE';
end;

class function TSubMemo.Version: Integer;
// *** version 0 ***
begin
  Result := 0;
end;

class function TSubMemo.ClassColor: TColor;
// *** couleur 0 ***
begin
  Result := clRed;
end;
```

Il reste à définir des descendants. Vous en créerez trois à titre d'exemples.

Voici comment compléter l'unité avec leurs déclarations :

```
{ TSubMemoOne }

TSubMemoOne = class(TSubMemo)
public
```

```

class function ClassText: string; override;
class function Version: Integer; override;
class function ClassColor: TColor; override;
end;

{ TSubMemoTwo }

TSubMemoTwo = class(TSubMemo)
public
  class function ClassText: string; override;
  class function Version: Integer; override;
  class function ClassColor: TColor; override;
end;

{ TSubMemoThree }

TSubMemoThree = class(TSubMemo)
public
  class function ClassText: string; override;
  class function Version: Integer; override;
end;

```

On voit que chacune se contente de modifier les fonctions de classe. Seule *TSubMemoThree* se dispense de toucher à *ClassColor* pour montrer qu'elle héritera de cette méthode virtuelle grâce à *TSubMemo*. L'implémentation sera faite ainsi :

```

{ TSubMemoThree }

class function TSubMemoThree.ClassText: string;
// *** légende 3 ***
begin
  Result := inherited ClassText + ' + 3 !';
end;

class function TSubMemoThree.Version: Integer;
// *** version 3 ***
begin
  Result := inherited Version + 30;
end;

{ TSubMemoTwo }

class function TSubMemoTwo.ClassText: string;
// *** légende 2 ***
begin
  Result := 'Mémo DEUX';
end;

class function TSubMemoTwo.Version: Integer;
// *** version 2 ***
begin
  Result := 2;
end;

class function TSubMemoTwo.ClassColor: TColor;
// *** couleur 2 ***
begin

```

```

    Result := clGreen;
end;

{ TSubMemoOne }

class function TSubMemoOne.ClassText: string;
// *** légende 1 ***
begin
    Result := 'Mémo UN';
end;

class function TSubMemoOne.Version: Integer;
// *** version 1 ***
begin
    Result := 1;
end;

class function TSubMemoOne.ClassColor: TColor;
// *** couleur 1 ***
begin
    Result := clBlue;
end;

```

Afin de bien reconnaître les classes invoquées, chacune disposera d'une couleur, d'un texte et d'un numéro de version propres. En guise de variantes, *TSubMemoThree* utilise l'héritage pour récupérer le texte et le numéro de version de son ancêtre avant de les modifier à sa manière.

Pour terminer cette unité, il manque deux éléments. La section *initialization* devra ressembler à ceci :

```

initialization
// enregistrement des classes
TSubMemoOne.NewMemo;
TSubMemoTwo.NewMemo;
TSubMemoThree.NewMemo;
end.

```

Vos trois classes seront ainsi enregistrées, *NewMemo* les ajoutant dans la liste interne *fMemos*.

Le dernier élément à déclarer est un type :

```

TSubMemoClass = class of TSubMemo;

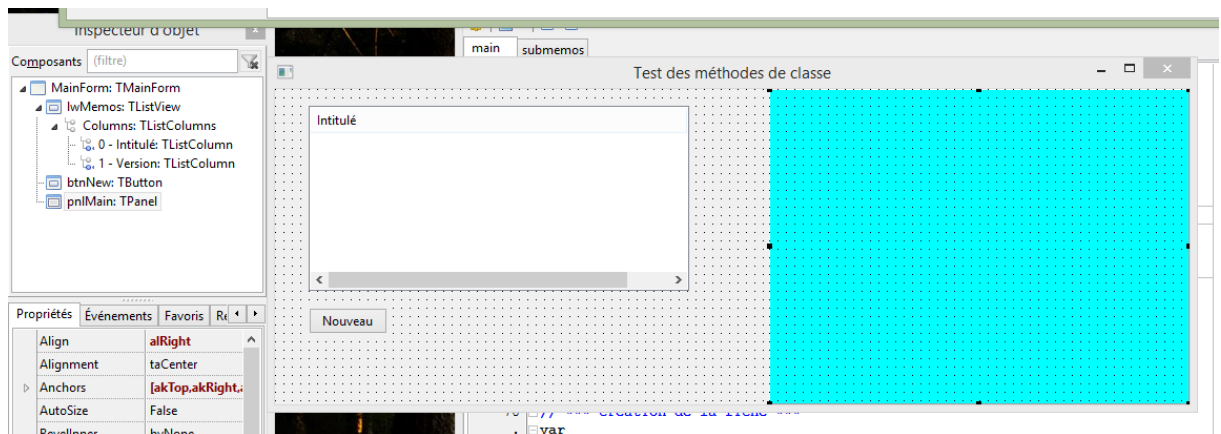
```

Il s'agit de déclarer une *métaclasse* : avec *TSubMemoClass*, on ne s'intéressera pas aux objets à instancier, mais uniquement aux champs, méthodes et propriétés de classe qui viennent d'être définis.

Il est à présent temps de s'occuper de la fiche principale :

- ajoutez un *TPanel* aligné à droite, au fond bleu clair que vous nommerez *pnlMain* ;

- ajoutez un *TListView* que vous nommerez *lwMemos* ;
- cliquez sur la propriété *columns* de ce dernier pour obtenir deux éléments en cliquant deux fois sur « ajouter » ;
- passez à *True* la propriété *AutoSize* de deux éléments obtenus ;
- passez à « Intitulé » la propriété *Caption* du premier élément et à « Version » celle du second élément :



- passez à *True* les propriétés *RowSelect* et *ReadOnly* de *lwMemos* ;
- passez à *vsReport* la propriété *ViewStyle* de *lwMemos* afin de voir des lignes comprenant les éléments et les sous-éléments ;
- créez un gestionnaire *OnCreate* de la fiche principale :

```

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
var
  LPtr: Pointer;
  LNewItem: TListItem;
begin
  // remplissage de la liste
  lwMemos.Items.BeginUpdate; // évite les scintillements
  try
    // on parcourt la liste des mémos enregistrés
    for LPtr in TSubMemo.Memos do
      begin
        LNewItem := lwMemos.Items.Add; // position de l'ajout
        // légende récupérée
        LNewItem.Caption := (TSubMemoClass(LPtr)).ClassText;
        // idem pour la version
        LNewItem.SubItems.Add(IntToStr(TSubMemoClass(LPtr).Version));
        // enregistrement de la classe
        LNewItem.Data := LPtr;
      end;
    finally
      lwMemos.Items.EndUpdate; // affichage
    end;
  end;

```

La propriété *Memos* de *TSubMemo* ayant été initialisée au démarrage du programme par une série de pointeurs vers des classes de ce type, on récupère les pointeurs et on complète la liste d'éléments du contrôle *lwMemos* avec les données

appropriées : la chaîne de caractères dans *Caption*, la version dans *SubItems* et le pointeur vers la classe dans *Data*. Le tout est protégé dans une instruction *try...finally...end* afin de s'assurer que la suspension d'affichage que provoque *BeginUpdate* soit toujours levée, même en cas d'erreur, par *EndUpdate*.



Remarquez que les données sont obtenues en transtypant les pointeurs avec *TSubMemoClass* : en effet, il s'agit de manipuler les classes en tant que telles et non des instances de ces classes.

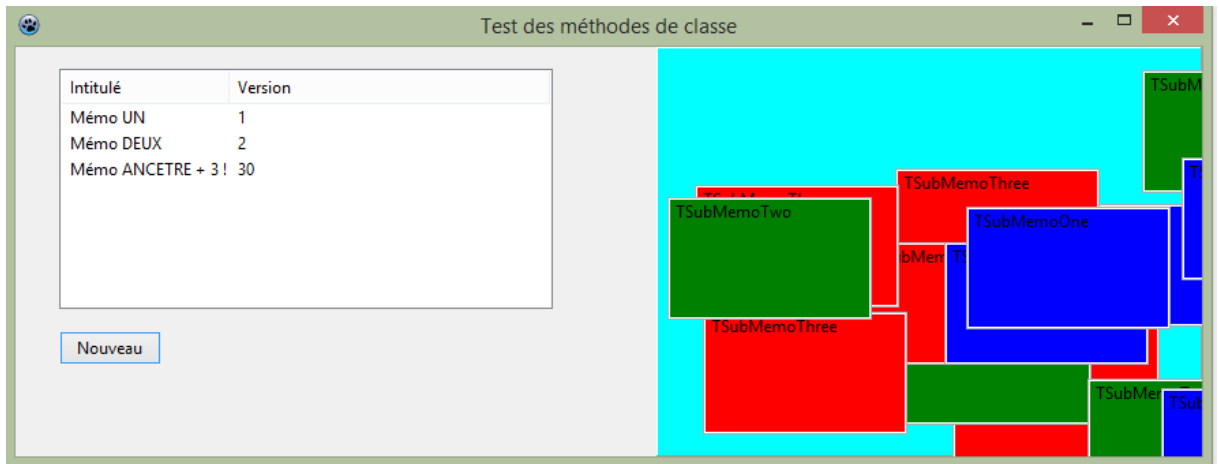
- ajoutez un bouton nommé *btnNew* ;
- associez à ce bouton le gestionnaire *OnClick* suivant :

```
procedure TMainForm.btnNewClick(Sender: TObject);
// *** nouveau mémo ***
var
  LNewMemo: TMemo;
begin
  if lwMemos.SelCount <> 0 then // des mémos enregistrés ?
  begin
    // on crée le mémo à partir de l'élément choisi dans la liste
    LNewMemo := TSubMemoClass(lwMemos.Selected.Data).Create(pnlMain);
    // position aléatoire
    LNewMemo.Left := random(400);
    LNewMemo.Top := random(300);
    // nom affiché
    LNewMemo.Caption := LNewMemo.ToString;
    // couleur de fond dépendant de la classe
    LNewMemo.Color := TSubMemoClass(lwMemos.Selected.Data).ClassColor;
    // le parent est le panneau pour l'affichage
    LNewMemo.Parent := pnlMain;
  end;
end;
```

La méthode va créer une instance de la classe qui descend de *TMemo* suivant l'élément sélectionné du *TListView*. Pour ce faire, elle transtype le pointeur contenu dans la propriété *Data* pour invoquer le constructeur *Create* avec le panneau *pnlMain* pour propriétaire. Elle définit ensuite une série de propriétés pour un affichage adapté de l'instance et termine en désignant le panneau comme *Parent*²²

En exécutant ce programme, vous créerez autant de mémos que vous le voudrez. Ils seront du type sélectionné dans le *TListView*. La libération des ressources se fera automatiquement puisque vous avez désigné *pnlMain* comme le propriétaire en charge de chacune des instances créées.

²² Désigner un *Parent* pour créer dynamiquement un contrôle est une obligation afin qu'il sache où il doit être dessiné.



Que le mécanisme vous paraisse un peu complexe n'aurait rien d'étonnant, mais relevez son efficacité : la fiche principale est indépendante de l'unité en charge des classes à manipuler si bien qu'il est possible d'ajouter et de retirer des classes à volonté sans toucher quoi que ce soit à l'unité principale.

Amusez-vous ainsi :

- transformez en commentaire une ou plusieurs des classes de l'unité *submemos* et examinez le changement de comportement du programme :

initialization

```
// enregistrement des classes
TSubMemoOne.NewMemo;
// TSubMemoTwo.NewMemo; <= la classe est inaccessible
TSubMemoThree.NewMemo;
end ;
```

- au contraire, utilisez la classe ancêtre :

initialization

```
// enregistrement des classes
TSubMemoOne.NewMemo;
TSubMemoTwo.NewMemo;
TSubMemoThree.NewMemo;
TSubMemo.NewMemo ; // la classe ancêtre
end ;
```

Encore mieux, créez votre propre descendant de *TSubMemo*, en lui ajoutant si nécessaire de nouvelles propriétés et méthodes. Vous constaterez que vous n'aurez jamais à modifier votre fiche principale qui saura manipuler les nouvelles classes sans même savoir ce pour quoi elles sont faites !

BILAN

LES EXCEPTIONS

TRAVAILLER AVEC LES EXCEPTIONS

LES TYPES D'EXCEPTION

DECLENCHER UNE EXCEPTION

REDÉCLENCHER UNE EXCEPTION

TRY... EXCEPT

TRY... FINALLY

EXCEPTIONS AVEC MODERATION

BILAN

FREE PASCAL ET LA POO

AU CŒUR DE **FREE PASCAL** : RTL, LCL ET FCL

TRAVAILLER AVEC LA POO

UN ANCETRE VENERABLE : TOBJECT

LE GRAND OUBLIE : TPERSISTENT

UN PRINCE RECONNU : TCOMPONENT

L'OMNIPRESENTE : TFORM

PARENT ET PROPRIETAIRE

On a vu que toute classe a un parent, ne serait-ce que *TObject*, et qu'elle hérite de ce parent. La notion de parenté fait par conséquent référence à la structure d'une classe et aux méthodes pour agir sur cette structure.

La notion de *propriétaire* d'un objet ne s'applique que pour les classes de la LCL qui descendent de la classe *TPersistent* définissant ainsi une propriété *Owner*. Cette dernière sera renseignée pour désigner la classe responsable de l'affichage et de la libération de l'objet en mémoire ainsi que de tous les objets lui appartenant.

Lorsque vous utilisez des composants dont le propriétaire 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 s'occupe d'allouer et de libérer la mémoire de manière transparente. Vous n'avez par conséquent pas à vous en charger.

Voilà pourquoi les composants présents sur la palette qui sont des classes particulières n'ont pas besoin d'être créés par vos soins. La LCL s'en occupe automatiquement à partir du propriétaire qui est à la base un descendant de *TForm*, c'est-à-dire une autre classe.

En revanche, dès que vous créez vous-même l'instance d'une classe, il est de votre responsabilité d'en libérer les ressources lors de sa destruction :

```
var
  MonAnimal : TAnimal ;
begin
  MonAnimal := TAnimal.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...
    MonAnimal.Dormir ;
  finally
    // en interne, la méthode Free appelle le destructeur Destroy
    MonAnimal.Free ; // les ressources seront toujours libérées
  end ;
end ;
```

FREE ET FREEANDNIL

LES ÉVÉNEMENTS

BILAN

LES PAQUETS
