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 opérateur *Is* et *As*

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

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 maitrisables.

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 attributs[1].

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 possèderait les procédures et les fonctions pour manipuler ses données. Vous pouvez aussi voir une classe comme une boîte noire fournissant un certain nombre de fonctionnalités à propos d'une entité aux attributs bien définis. Peu importe ce qu'il se passe dans cette boîte dans la mesure où elle remplit au mieux les tâches pour lesquelles elle a été conçue.

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

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

Les difficultés commenceraient 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;
```

[1] *L'orienté objet* – Bersini Hugues – Eyrolles 2007

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

Dans un tel cas de figure, la **POO** va d'emblée montrer son efficacité. Il vous faudra déclarer une classe[2] :

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

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

CHAMPS, METHODES ET PROPRIETES

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

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

Une *propriété* est un moyen d'accéder à un champ : *fNom* est par exemple est accessible grâce à la propriété *Nom*. Les propriétés se servent des mots réservés *read* et *write* pour cet accès[3].

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

[3] Les propriétés seront étudiées en détail dans le chapitre suivant : POO à gogo.

[Exemple PO_01]

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

- {$mode objfp}
- {$mode delphi}

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

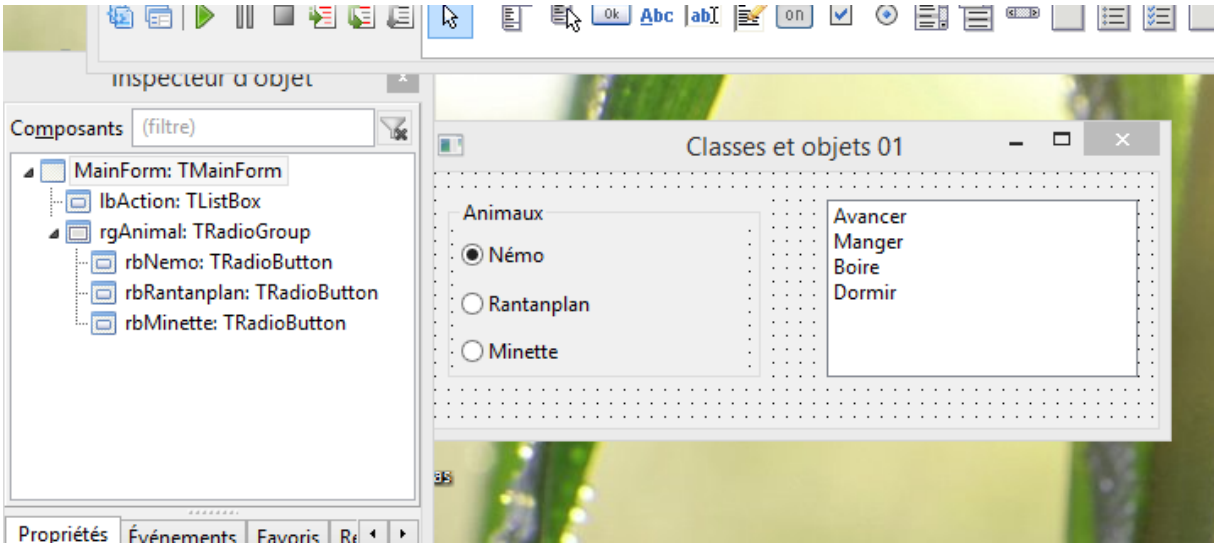
Contrairement à la *classe* qui est une structure abstraite, l'*objet* est la concrétisation de cette classe : on parlera *d'instanciation* pour l'action qui consiste essentiellement à allouer de la mémoire pour l'objet et à renvoyer un pointeur vers l'adresse de son implémentation. L'objet lui-même 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 moule[4].

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 :

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



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

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

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

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

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

```
UnAnimal := Rantanplan ;[5]
```

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 :

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

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

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

Pour ce faire :

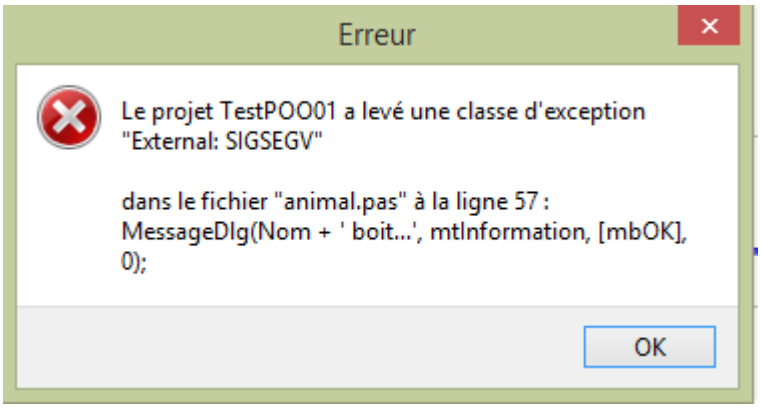
- cliquez tour à tour sur chacun des composants voulus de telle manière que Lazarus crée pour vous le squelette des méthodes ;
- complétez le corps des maéthodes comme suit :

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

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

procedure TMainForm.rbNemoClick(Sender: TObject);
// *** l'animal est 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 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 gérer[6]. 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*[7], 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 !

[6] Vous verrez par la suite que cette obligation ne s'applique pas pour un objet dont le propriétaire est défini.
[7] Il est possible d'appeler autant de fois que vous le désirez la méthode *Create*, même s'il est rare d'avoir à le faire.

 C'est à partir du nom de la classe (ici, *TAnimal*) qu'on crée un objet.

À partir du moment où un objet a été créé, une variable appelée *Self* est définie implicitement pour chaque méthode de cet objet. Son utilisation la plus fréquente est de servir de paramètre à une méthode ou à une routine qui a besoin d'une référence à l'objet[8].

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 ***
```

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

```
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 cause[9]. 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

[9] 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.

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

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

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

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

```
// Erreur : compilation refusée
 MonObjet.AFaimAussi := MonAnimal.fAfaim ;
// OK si ATresSoif est une propiété 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 méthodes[10].
- 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 !

[10] Voir le chapitre suivant : POO à gogo.

- enfin, *published* sera un outil précieux lors de l'intégration de composants dans la palette de Lazarus.


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

HERITAGE

Jusqu'à présent, les classes vous ont sans doute semblé de simples enregistrements (*record*) aux capacités étendues : en plus de proposer une structure de données, elles fournissent les méthodes pour travailler sur ces données. Cependant, la notion de classe est bien plus puissante que ce qu'apporte l'encapsulation : il est aussi possible de dériver des sous-classes d'une classe existante qui hériteront de toutes les fonctionnalités de leur ancêtre. 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 ancêtre, 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 ancêtre, de l'ancêtre de son ancêtre, 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 manipuliez la classe *TAnimal*, vous pourriez avoir à travailler avec un ensemble de chiens et envisager alors de créer un descendant *TChien* aux propriétés et méthodes étendues.

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

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

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

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

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

 { TChien }

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


implementation

uses
 Dialogs; // pour les boîtes de dialogue

{ TChien }

procedure TChien.SetBatard(AValue: Boolean);
begin
```

```
end;

procedure TChien.Aboyer;
begin

end;

procedure TChien.RemuerDeLaQueue;
begin

end;

{ TAnimal }

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

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

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

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

```
{ TChien }

procedure TChien.SetBatard(AValue: Boolean);
begin
  fBatard := AValue;
end;

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

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

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

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

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }


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

NOTION DE POLYMORPHISME

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

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

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

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

```
end;
```

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

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

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

Grâce au polymorphisme, l'affectation suivante est correcte :

```
UnAnimal := Rantanplan ;
```

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

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

```
Rantanplan := UnAnimal ;
```

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


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

LES OPERATEURS IS ET AS

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

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

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

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

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

Et ceci avec *as*:

```
(Rantaplan 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 Rantaplan mais pas pour les autres
```

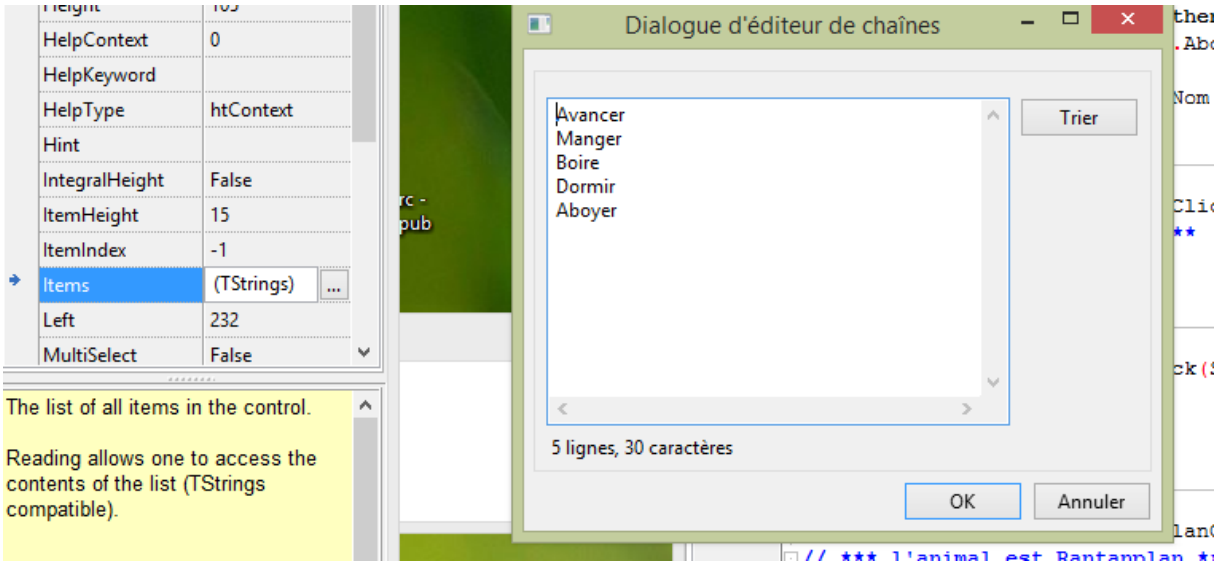
Le déclenchement possible d'une erreur avec *as* conduit à l'accompagner la plupart du temps d'un test préalable avec *is* :

```
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 *poo* du répertoire e*xemples*.

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 l'ancêtre *TAnimal*, vous estimerez à juste titre qu'elle a besoin d'être adaptée au régime d'un carnivore. Afin de la redéfinir, il suffirait de l'inclure à nouveau dans l'interface puis de coder son comportement actualisé.

[Exemple PO-04]

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

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

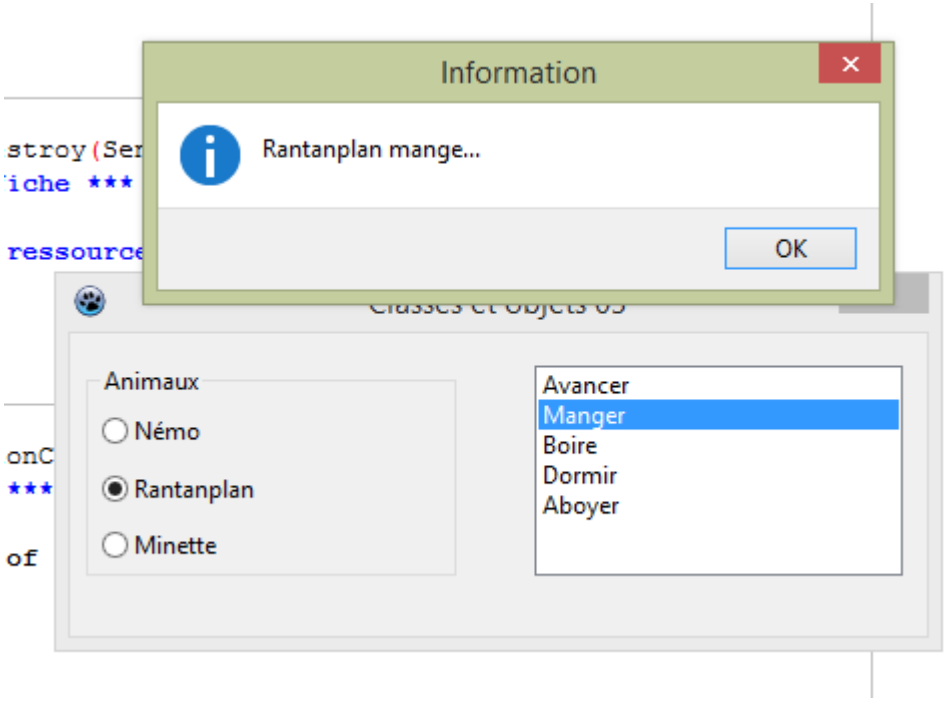
```
TChien = class(TAnimal)
strict private
  fBatard : Boolean ;
  procedure SetBatard;
public
  procedure Manger; //  <= la méthode est redéfinie
  procedure Aboyer;
  procedure RemuerDeLaQueue;
  property Batard: Boolean read fBatard write SetBatard;
```

```
end;
```

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

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

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



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

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

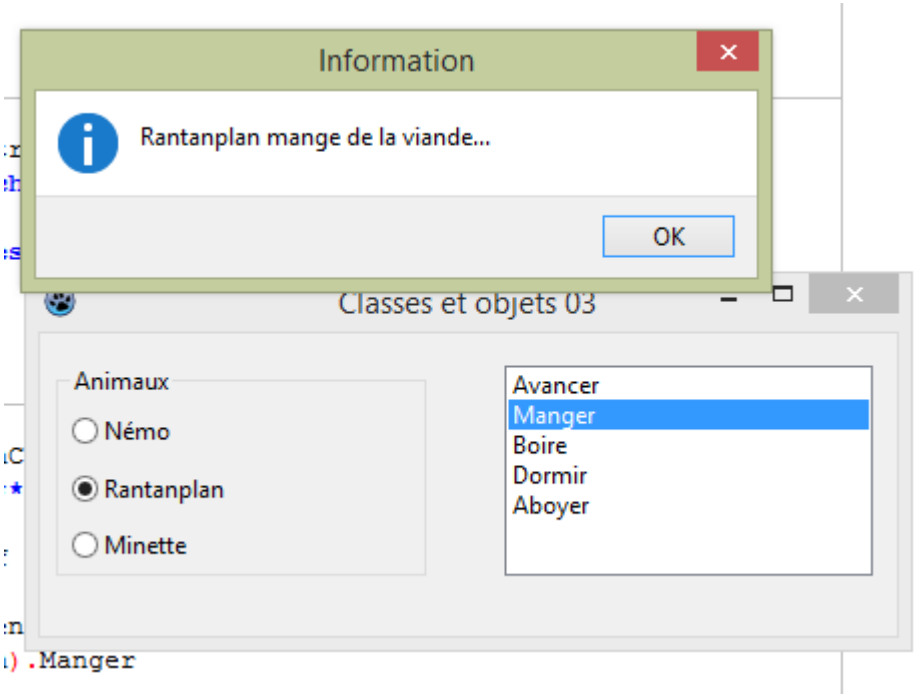
En effet, en écrivant *UnAnimal.Manger*, vous demandez à un animal de manger et non à un chien ! Vous obtenez logiquement ce que sait faire tout animal, à savoir manger, et non la spécialisation de ce que fait un chien carnivore.

Dès lors que votre classe *TChien* a redéfini le comportement de son ancêtre, 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 de son ancêtre tout en autorisant si nécessaire de lui apporter des compléments.

Contrairement aux méthodes statiques dont le compilateur connaît directement les adresses, les méthodes virtuelles sont accessibles en interne *via* une table[11] d'exécution qui permet de retrouver les adresses de chacune des méthodes dont il a hérité et de celles qu'il a définies lui-même.

[Exemple PO-05]

Pour le programmeur, la déclaration d'une telle méthode se fait par l'ajout du mot *virtual* après sa déclaration. Il est aussi possible d'utiliser *dynamic* qui est strictement équivalent pour Free Pascal, mais qui a un sens légèrement différent avec Delphi[12].

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 ancêtre : 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 Aboyer;
  procedure RemuerDeLaQueue;
```

- 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
```

[11] Cette table porte le nom de VMT : *Virtual Method Table*.

[12] 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.

```
  inherited Manger; // on hérite de la méthode de l'ancêtre
  MessageDlg('... mais principalement de la viande...', mtInformation, [mbOK], 0);
end;
```

On a introduit un mot réservé qui fait appel à la méthode de l'ancêtre : *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 aliment[13]. 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 l'ancêtre 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 spécialisée[14].


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 de l'ancêtre
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

[13] 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.

[14] 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 !

que leur appel est plus lent que celui des méthodes statiques et que la table des méthodes consomme de la mémoire supplémentaire. En fait, utilisez la virtualité dès qu'une des classes qui descendrait de votre classe serait susceptible de spécialiser ou de compléter certaines de ses méthodes. C'est ce que vous avez fait avec la méthode *Manger* : elle renvoie à un comportement général, mais sera probablement précisée par les descendants de *TAnimal*.

Pour résumer :

- on ajoute *virtual* à la fin de la ligne qui définit une première fois une méthode virtuelle ;
- *dynamic* est strictement équivalent à *virtual* (mais a un sens différent avec Delphi) ;
- on ajoute *override* à la fin de la ligne qui redéfinit une méthode virtuelle dans un de ses descendants ;
- on utilise *inherited* à l'intérieur de la méthode virtuelle redéfinie pour hériter du comportement de son ancêtre ;
- 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 de *inherited* au sein d'une méthode a son importance : si l'on veut modifier le comportement de l'ancêtre, 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 l'ancêtre 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 besoin une propriété que vous voulez initialiser à votre manière ? Dans ce cas, les *Create* hérités annuleraient votre travail !

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

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

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

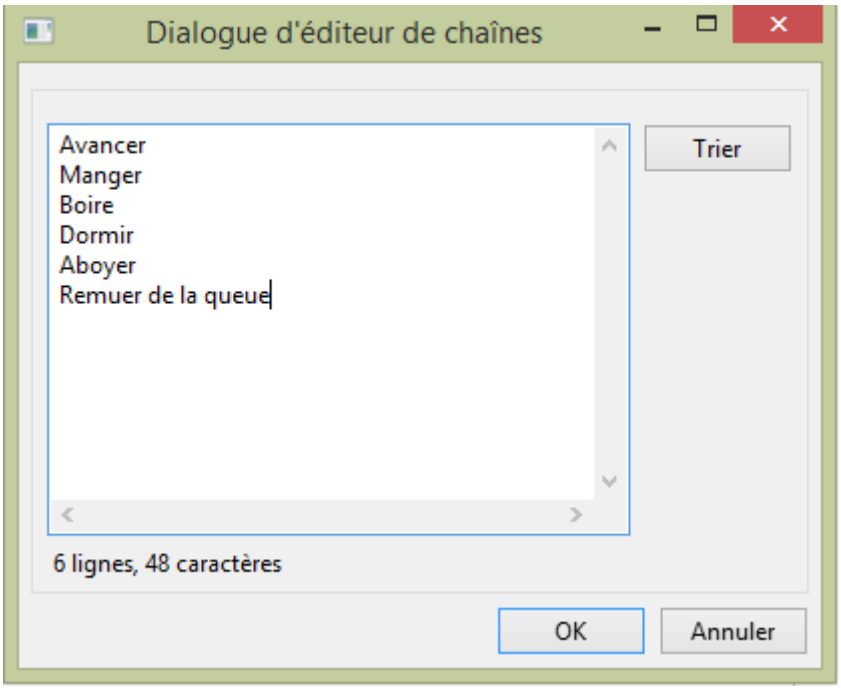
```
procedure TChien.RemuerDeLaQueue;
begin
  inherited Manger; // <= Manger vient de TAnimal !
  MessageDlg('C''est pourquoi il remue de la queue...', mtInformation, [mbOK], 0);
end;
```

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

[Exemple PO-06]

Pour obtenir ce résultat, procédez ainsi :

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



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

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

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

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

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 manipulées[15].

Voici un court extrait de son interface :

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

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

Les descendants de *TStrings* procèderont 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;
```

[15] Une des classes les plus utilisées, *TStringList*, a pour ancêtre *TStrings*.

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 :

- rouvrez 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 : *Rantaplan*, *Nemo* ou *Minette*. Les méthodes de classe obéissent en effet aux mêmes règles de portée et d'héritage que les méthodes ordinaires. Elles peuvent être virtuelles et donc redéfinies.

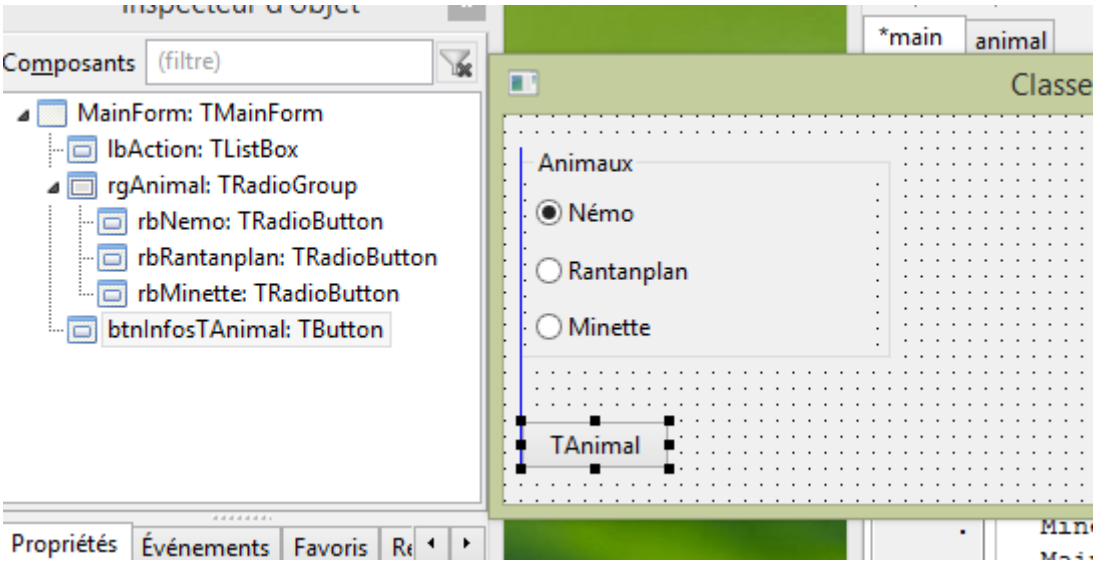
Leurs limites découlent de leur définition même : comme elles sont indépendantes de l'instanciation, elles ne peuvent pas avoir accès aux champs, propriétés et méthodes ordinaires de la classe à laquelle elles appartiennent. De plus, depuis une méthode de classe, *Self* pointe non pas vers l'instance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

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

[Exemple PO-08]

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

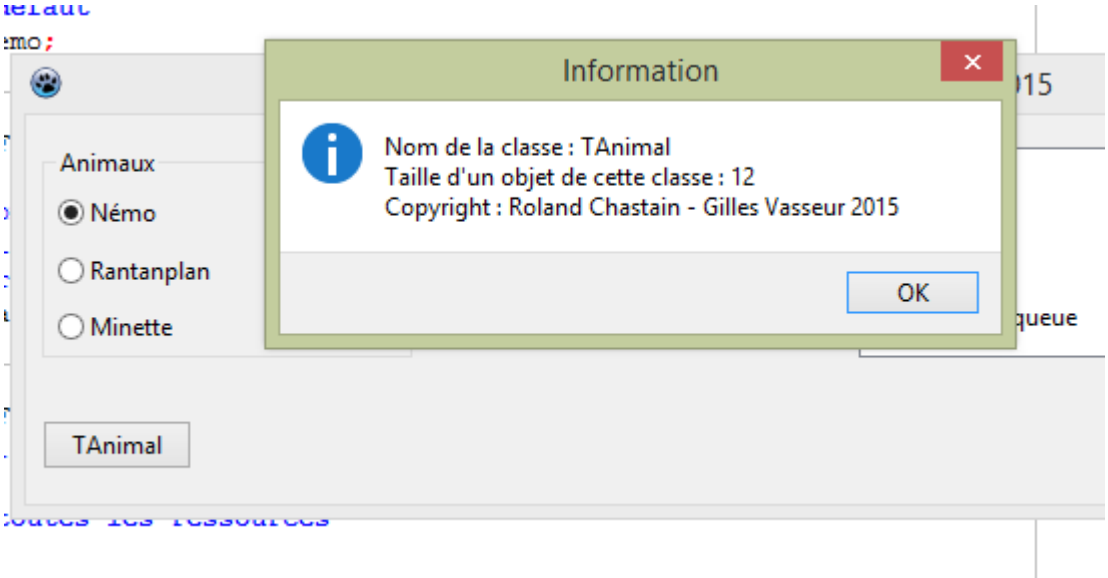
- ajoutez un bouton à la fiche principale, renommez-le *btnInfosTAnimal* et changez sa légende en *TAnimal* ;



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

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

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



Mais où les méthodes de classe *ClassName* et *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]

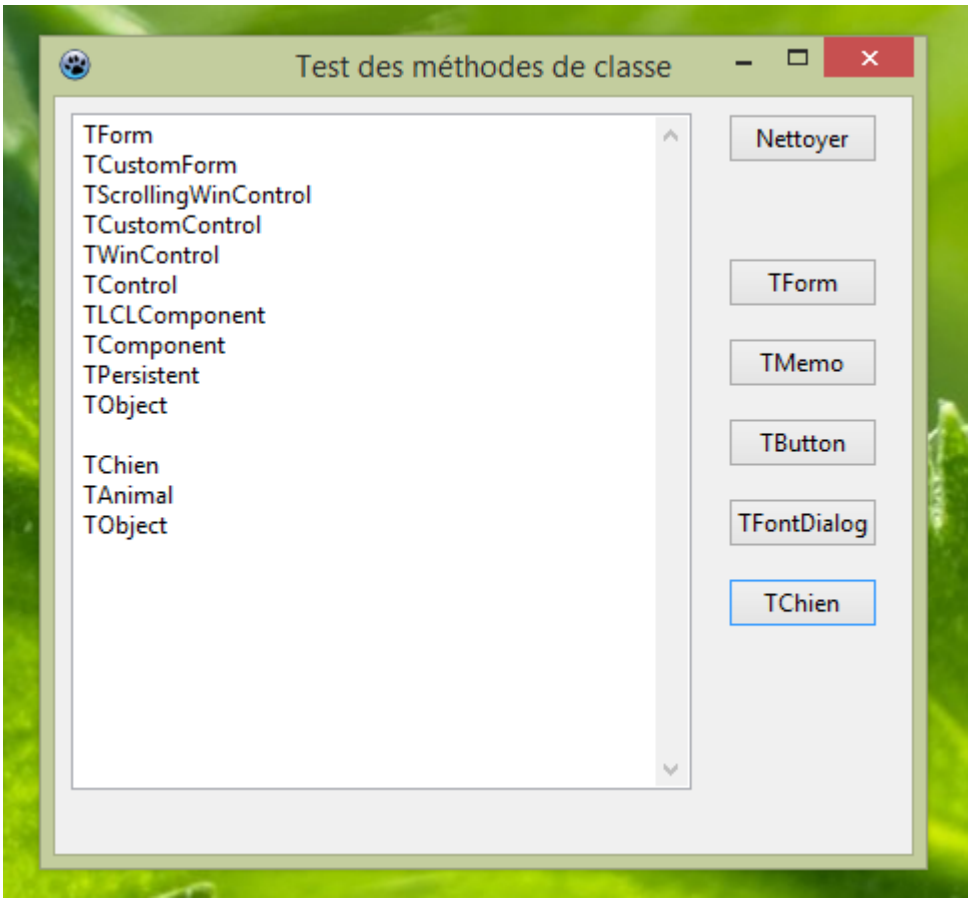
En utilisant un composant *TMemo* nommé *mmoDisplay*, la méthode d'exploration pourra ressembler à ceci :

```
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 donné par ce programme dont le code source complet est présent dans le répertoire de l'exemple :



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*.

METHODES DE CLASSE STATIQUES

METHODES DE MESSAGE