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 des langages fondés sur 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 – Notion de constructeur – Notion de destructeur – Premiers gains de la POO – *Principes et techniques de la POO* : Encapsulation – Notion de portée – Notion d'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

En regroupant les instructions au sein de modules appelés *fonctions* et *procédures*, la *Programmation Structurée* a permis une meilleure lisibilité et par conséquent une maintenance améliorée des programmes. 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 pour 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(const 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 imaginer des techniques pour reconnaître les animaux, fournir leur état et en décrire le

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

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(const 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. Vous n'aurez besoin de rien de plus : chaque animal particulier sera reconnu grâce à la variable dédiée et seules ses caractéristiques seront prises en compte, sans intervention de votre part.

Par la suite, il s'agit de donner chair à cette classe en renseignant chacun de ses éléments.

CHAMPS, METHODES ET PROPRIETES

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

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

Une *propriété* est avant tout un moyen d'accéder à un champ : par exemple, *fNom* 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 troisième chapitre sur la POO.

[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}


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*, 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, votre 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(const 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(const 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.
```

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, à ce stade, n'est pas utile à votre compréhension.

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 routine 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* ;
- 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 boîte sans vie.

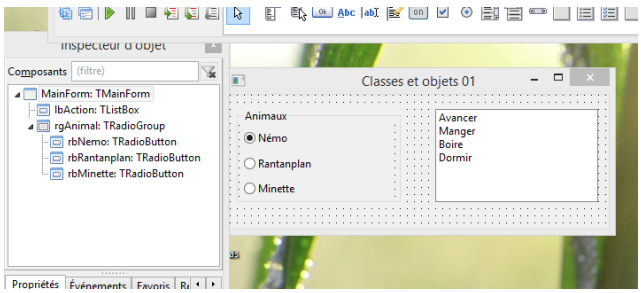
LES OBJETS

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*, 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 l'unité *MainForm*, définissez quatre variables de type ***TAnimal*** : Nemo, Rantanplan, Minette et UnAnimal :

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

Vous avez ainsi déclaré trois animaux grâce à trois variables du type ***TAnimal***. 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 seulement 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 :

- double-cliquez tour à tour sur chacun des composants voulus de telle manière que **Lazarus** crée pour vous le squelette des méthodes[6] ;

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

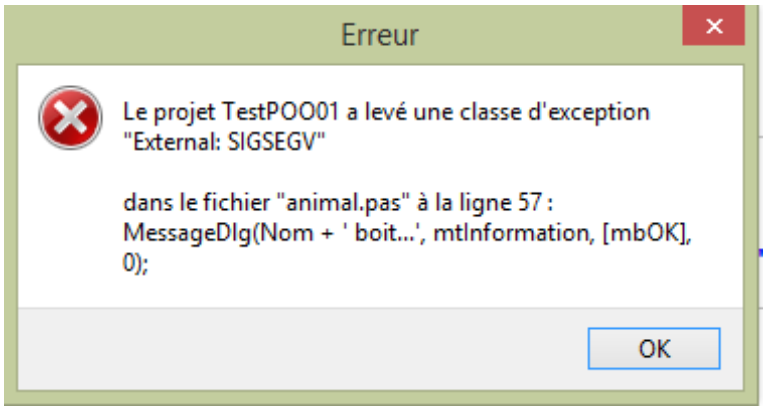
[6] Les méthodes *OnClick* sont en effet les méthodes créées par défaut pour de nombreux composants.

- complétez le corps des méthodes comme suit :

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

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

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

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

NOTION DE 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* ». Cette erreur survient quand vous tentez d'accéder à des emplacements de mémoire qui ne vous sont pas réservés.

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 gérer[7].

L'instanciation de ***TAnimal*** prendra 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 peut être 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*[8], 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 ainsi 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 le suivant :

- 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 pour tout l'objet, initialise cette zone de mémoire et renvoie un pointeur vers elle.

[7] Vous verrez par la suite que cette obligation ne s'applique pas pour un objet dont le propriétaire est défini (voir page 140).
[8] 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.

 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…[9]

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 référence[10].

NOTION DE 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. Par conséquent, 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. Il en découle qu'on définit la méthode *Destroy*, mais qu'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;
```

[9] Pour rappel, la valeur nulle d'un pointeur est fournie par la constante *nil*.
[10] Pour des exemples d'utilisation de *self*, voir le chapitre sur la LCL.

- 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 ainsi que les actions à effectuer sont reconnus.

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** : une fiche est un objet sur lequel vous déposez d'autres objets comme des étiquettes, des éditeurs, des images, tous des instances de classes prédéfinies par l'EDI. 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[11]. Dans l'exemple sur 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 de bouts de codes entremêlés.

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 que vous ayez pu les utiliser sans avoir eu à les définir ?

[11] 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. Pour davantage d'informations voir le chapitre sur la création des composants (voir page 144).

PRINCIPES ET TECHNIQUES DE LA POO

ENCAPSULATION

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

```
TAnimal = class // c'est bien une classe
strict private // indique que ce qui suit n'est pas visible à l'extérieur de la classe
  fNom: string; // un champ de type chaîne
  fASoif : Boolean ; // deux champs booléens
  fAFaim : Boolean ;
  procedure SetNom(AValue : string); // détermine la valeur d'un champ via une méthode
public // indique que ce qui suit est visible à l'extérieur de la classe
  procedure Avancer ; // des méthodes…
  procedure Manger;
  procedure Boire;
  procedure Dormir;
  // les propriétés permettant 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 des champs, 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 rendra 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 de ce fait à l'utiliser ?

Lazarus définit six niveaux de portée :

- *strict private* : l'élément n'est visible 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, 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 d'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 directement.
- 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[12].
- 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**.


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 dans tout le programme, elles sont souvent sources d'erreurs parfois difficiles à détecter et à corriger.

NOTION D'HERITAGE

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

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


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 ***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]

[12] Voir le chapitre suivant sur les méthodes (page 71).

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*, 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* 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 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. Ce comportement est identique à celui attendu dans le monde réel : vous savez qu'un chien est toujours un animal, mais rien ne vous assure qu'un animal soit forcément un chien !

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 et 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*:

```
(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
```

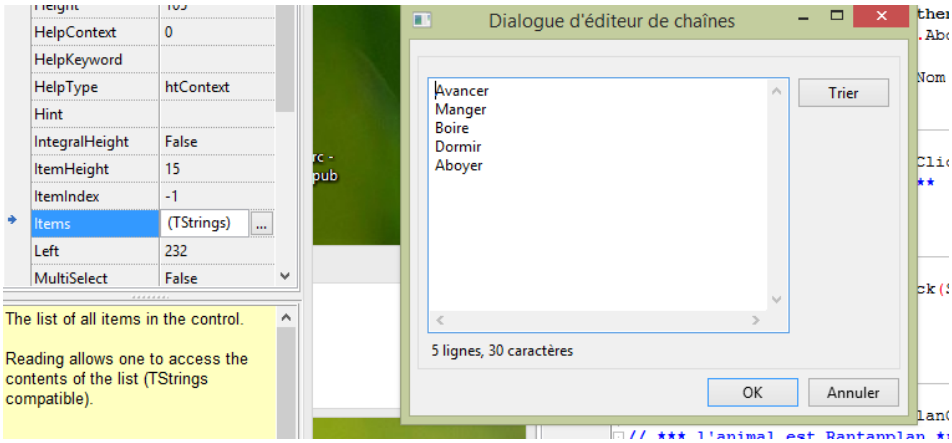
Le déclenchement possible d'une erreur avec *as* conduit à l'accompagner la plupart du temps d'un test préalable avec *is* :

```
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 avez 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.