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 statiques de classe – Méthodes de message

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

CE QU'IL FAUT SAVOIR…

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

METHODES STATIQUES

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

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

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

[Exemple PO-04]

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

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

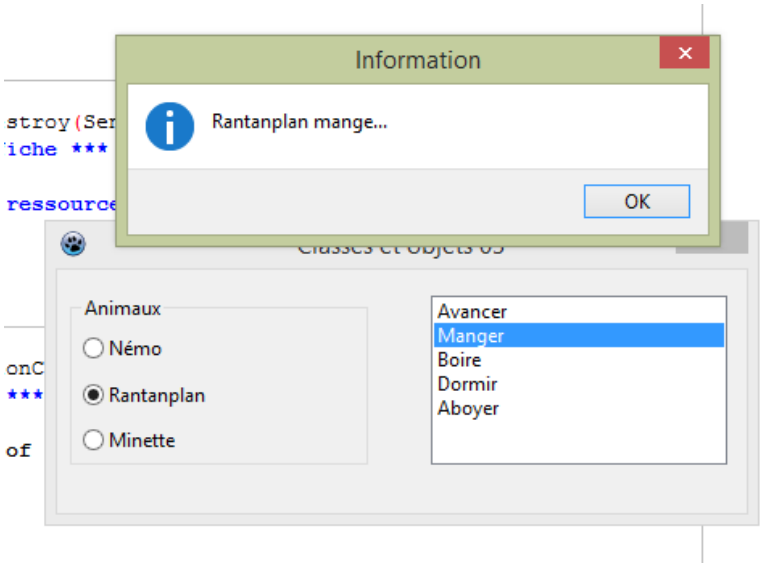
```
TChien = class(TAnimal)
strict private
  fBatard : Boolean ;
  procedure SetBatard;
```

```
public
  procedure Manger; //  <= la méthode est redéfinie
  procedure Aboyer;
  procedure RemuerLaQueue;
  property Batard: Boolean read fBatard write SetBatard;
end;
```

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

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

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



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

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

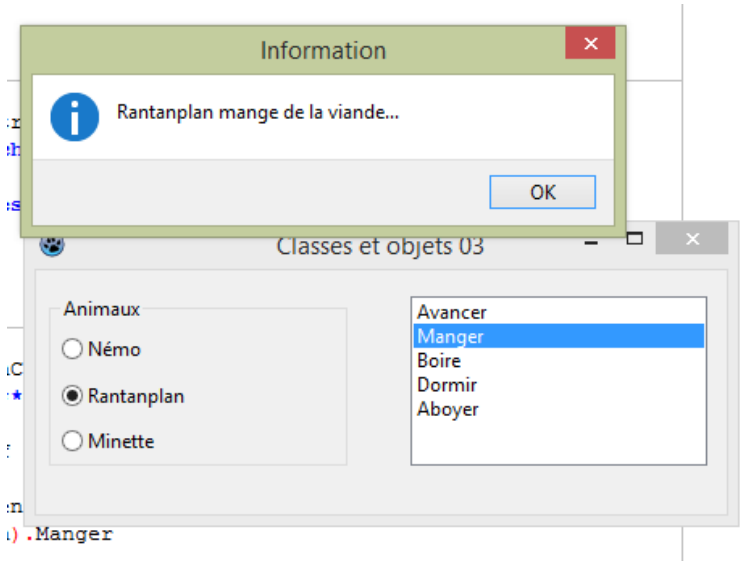
En effet, en écrivant *UnAnimal.Manger*, vous demandez à un animal de manger et non à un chien ! Vous obtenez logiquement ce que sait faire tout animal, à savoir manger, et non la spécialisation de ce que fait un chien carnivore.

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

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

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

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



METHODES VIRTUELLES

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 ? C'est la tâche des *méthodes virtuelles* que vous allez étudier à présent.

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

Contrairement aux méthodes statiques dont le compilateur connaît directement les adresses, les méthodes virtuelles sont accessibles en interne *via* une table[1] d'exécution qui permet de retrouver à l'exécution les adresses de chacune des méthodes dont il a hérité et de celles qu'il a définies lui-même.

[Exemple PO-05]

Pour le programmeur, la déclaration d'une telle méthode se fait par l'ajout du mot *virtual* après sa déclaration. Il est aussi possible d'utiliser *dynamic* qui est strictement équivalent pour **Free Pascal**, mais qui a un sens légèrement différent avec **Delphi**[2].

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

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

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

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

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

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

```
public
   procedure Manger; override; // <= ligne changée
   procedure Aboyer;
   procedure RemuerLaQueue;
```

- recompilez le projet pour constater que l'avertissement a disparu ;
- modifiez la méthode *Manger* pour qu'elle bénéficie de la méthode de son ancêtre :

```
procedure TChien.Manger;
begin
  inherited Manger; // on hérite de la méthode du parent
```

[1] Cette table porte le nom de VMT : *Virtual Method Table.*
[2] 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.

```
    MessageDlg('... mais principalement de la viande...', mtInformation, [mbOK], 0);
end;
```

On a introduit un mot réservé qui fait appel à la méthode du parent : *inherited*. Si vous lancez l'exécution du programme, vous constatez que choisir « *Rantanplan* » puis « *Manger* » provoque l'affichage de deux boîtes de dialogue successives : la première qui provient de ***TAnimal*** grâce à *inherited* précise que Rantanplan mange tandis que la seconde qui provient directement de ***TChien*** précise que la viande est son aliment principal[3]. Grâce à la table interne construite pour les méthodes virtuelles, le programme a été aiguillé correctement entre les versions de *Manger*.

Il est bien sûr possible de laisser tel quel le comportement d'une méthode virtuelle, tout comme il est possible de modifier une méthode virtuelle que le parent aura ignoré et donc de remonter dans la généalogie. Très souvent, on définit une classe générale qui se spécialise avec ses descendants, sans avoir à tout prévoir avec l'ancêtre le plus générique et tout à redéfinir avec la classe la plus spécialisée[4].


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 spécifiant que cet écrasement est voulu. Pour cela, faites suivre la redéfinition de votre méthode virtuelle par le mot réservé *reintroduce* :

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

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

Étant donné la puissance et la souplesse des méthodes virtuelles, vous vous demanderez peut-être pourquoi elles ne sont pas employées systématiquement : c'est que leur appel est plus lent que celui des méthodes statiques et que la table des méthodes consomme de la mémoire supplémentaire. En fait, utilisez la virtualité dès qu'une des classes qui descendrait de votre classe serait susceptible de spécialiser ou de compléter certaines de ses méthodes. C'est ce que vous avez fait avec la méthode

[3] 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.

[4] 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 !

Manger : elle renvoie à un comportement général, mais sera probablement précisée par les descendants de ***TAnimal***.

Pour résumer :

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

COMPLEMENTS SUR *INHERITED*

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

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

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

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

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

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

Il faut en effet vous dire que vous ne connaissez pas toujours exactement les actions exécutées par tous les ancêtres de votre classe : êtes-vous sûr qu'aucun d'entre eux ne modifiera pour ses propres 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 Destroy ; // on hérite ensuite !
end;
```

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

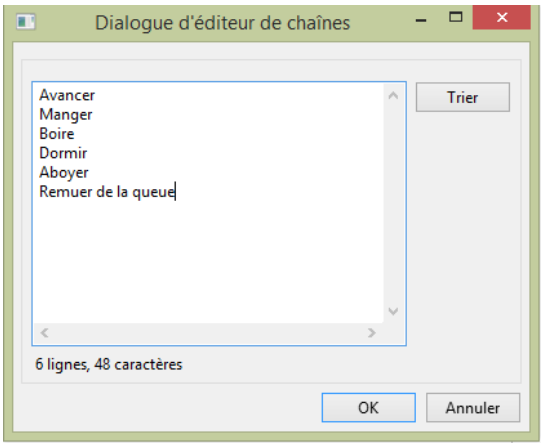
```
procedure TChien.RemuerLaQueue;
begin
 inherited Manger; // <= Manger vient de TAnimal !
 MessageDlg('C''est pourquoi il remue la queue...', mtInformation, [mbOK], 0);
end;
```

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

[Exemple PO-06]

Pour obtenir ce résultat, procédez ainsi :

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



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

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

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

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

… ET CE QU'IL EST UTILE DE SAVOIR

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

METHODES ABSTRAITES

Il peut être utile dans une classe qui servira de moule à d'autres classes plus spécialisées de déclarer une méthode indispensable, 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 le qualifiant *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[5].

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
```

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

On y reconnaît une méthode statique (*Error*), une méthode virtuelle redéfinie (*DefineProperties*) et une méthode virtuelle simple (*GetCapacity*). Nouveauté : les méthodes *Get* et *GetCount* sont marquées par le mot-clé *abstract* qui indique que ***TStrings*** ne propose pas d'implémentation pour ces méthodes parce qu'elle n'aurait 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;
```

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***, se montrant par conséquent bien plus polyvalente.

Au passage, vous aurez encore vu une manifestation de la puissance du polymorphisme : bien qu'en partie abstraite, ***TStrings*** sera, en certaines circonstances, très 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* et modifiez ainsi la déclaration de la classe ***TAnimal*** :

```
{ TAnimal }

 TAnimal = class
 private
  fNom: string;
  fASoif: Boolean ;
  fAFaim: Boolean ;
  procedure SetNom(const 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*, 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 exécutant 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 de *Copyright* grâce à 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 (VMT) qu'il est alors possible d'examiner.

En revanche, elles peuvent avoir accès aux champs de classe, propriétés de classe et méthodes de classe : comme les autres membres d'une classe indépendants de l'instanciation, leur déclaration commence toujours par le mot *class*.

Par exemple, une variable de classe sera déclarée ainsi :

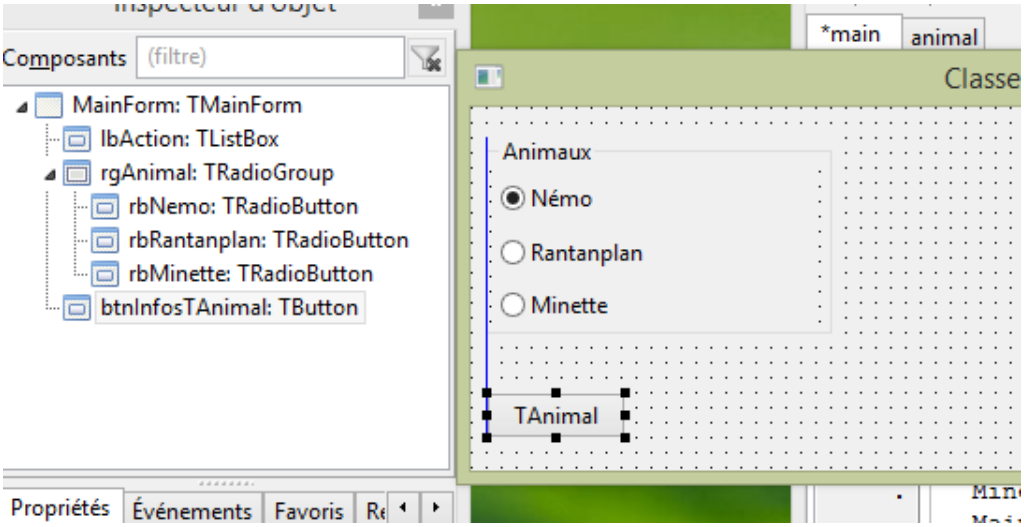
```
class var MyVar : Integer ;
```


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

[Exemple PO-08]

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

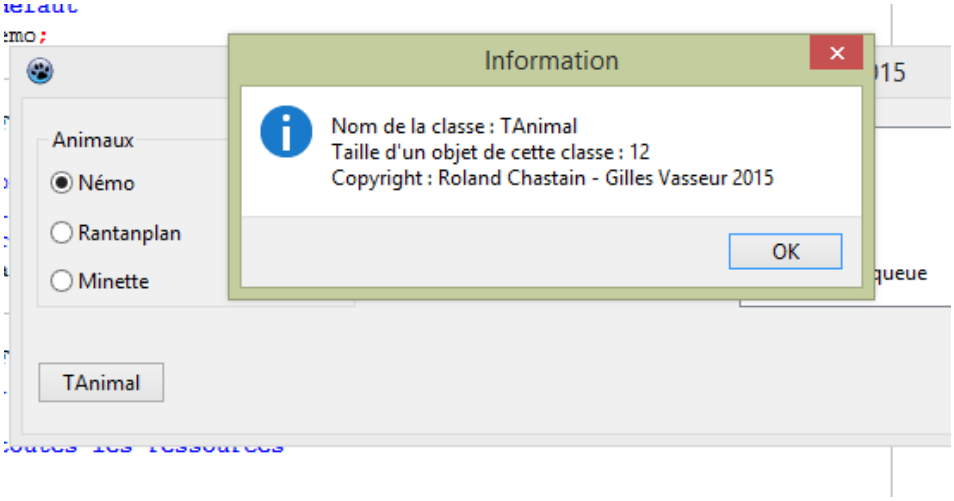
- ajoutez un bouton à la fiche principale, renommez-le *btnInfosTAnimal* et changez sa légende en « *TAnimal* » ;



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

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

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



Mais où les méthodes de classe *ClassName* et *IntanceSize* ont-elles été déclarées ? Elles proviennent de l'ancêtre ***TObject*** qui les définit par conséquent pour toutes les classes. Vous pourrez donc vous amuser à remplacer ***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. Cette méthode de classe est elle aussi définie par ***TObject***. 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 pour l'affichage un composant ***TMemo*** nommé *mmoDisplay*, la méthode d'exploration pourra ressembler à ceci :

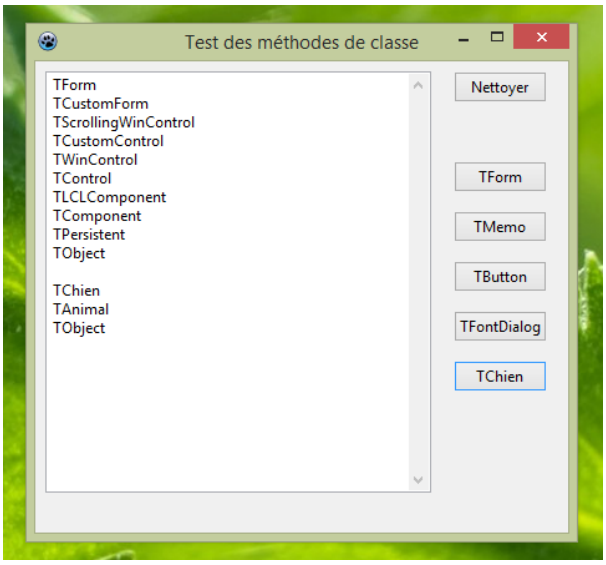
```
procedure TMainForm.Display(AClass: TClass);
begin
  repeat
    mmoDisplay.Lines.Add(AClass.ClassName);
```

```
    AClass := AClass.ClassParent;
  until AClass = nil;
  mmoDisplay.Lines.Add('');
end;
```

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

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

Voici un affichage obtenu par ce programme :



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

Voici le listing complet de cet exemple :

```
unit main;

{$mode objfpc}{$H+}

interface

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

type

  { TMainForm }
```

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

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

uses
  animal; // unité pour le traitement des animaux

{ TMainForm }

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

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

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

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

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

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

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

end.
```

METHODES STATIQUES DE CLASSE

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

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

[Exemple PO-10]

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

En voici le listing complet :

```
unit main;

{$mode objfpc}{$H+}

interface

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

type

  { TMyClass }

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

  { TMainForm }

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

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

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

procedure TMainForm.btnOKClick(Sender: TObject);
// *** affichage de la valeur ***
begin
```

```
  mmoDisplay.Lines.Add(TMyClass.GetMyValue); // on affiche
end;

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

{ TMyClass }

class procedure TMyClass.SetMyValue(const AValue: string);
// *** la valeur est mise à jour ***
begin
  fMyValue := Upcase(AValue); // en majuscules
end;

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

end.
```

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

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

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

Leur comportement est aussi atypique :

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

[Exemple PO-11]

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

```
unit main;

{$mode objfpc}{$H+}

interface

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

type

  { TMyClass }

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

  { TMainForm }

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

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMyClass }

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

class destructor TMyClass.Destroy;
```

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

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

{ TMainForm }

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

initialization
 MessageDlg('Initialization : ' + TMyClass.Access.MyFunct, mtInformation, [mbOK], 0);
finalization
 MessageDlg('Finalization', mtInformation, [mbOK], 0);
end.
```

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

Le mécanisme de l'ensemble est celui-ci :

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

METHODES DE MESSAGES

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

reste cependant intéressant de les connaître pour résoudre certains problèmes avec une économie de moyens remarquable.

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

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

S'il est possible de gérer les messages de l'OS (c'est ce que fait sans cesse **Free Pascal** avec ses bibliothèques), vous serez ici invité à générer vos propres messages.

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

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

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

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

LM_User est prédéfinie afin que l'utilisateur ne choisisse qu'une valeur non déjà utilisée par le système.

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

[Exemple PO-12]

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

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  LMessages; // unité pour les messages

const
  M_CHANGEDMESSAGE = LM_User + 1; // message de changement
  M_LOSTMESSAGE = LM_User + 2; // message perdu

type

  { TMainForm }

  TMainForm = class(TForm)
    btnLost: TButton;
    edtDummy: TEdit;
    mmoDisplay: TMemo;
    procedure btnLostClick(Sender: TObject);
    procedure edtDummyChange(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
    procedure Changed(var Msg: TLMessage); message M_CHANGEDMESSAGE;
    procedure DefaultHandler(var AMessage); override;
  end;


var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

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

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

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

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

end.
```

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

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

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

SURCHARGE DE METHODES

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

Si sa valeur de retour et/ou ses paramètres sont différents de ceux de son ancêtre, la nouvelle méthode coexistera avec la méthode héritée. Vous pourrez par conséquent faire appel aux deux : l'implémentation activée sera celle qui correspondra

aux paramètres invoqués. Ce sont vos méthodes qui seront polymorphiques puisqu'elles s'adapteront à vos souhaits


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

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

[Exemple PO-13]

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

```
type

 { TAddition }

 TAddition = class
  function AddEnChiffres(Nombre1, Nombre2: Integer): string;
  function AddVirtEnChiffres(Nombre1, Nombre2: Integer): string; virtual;
 end;

 { TAdditionPlus }

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

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

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

```
{ TAddition }

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

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

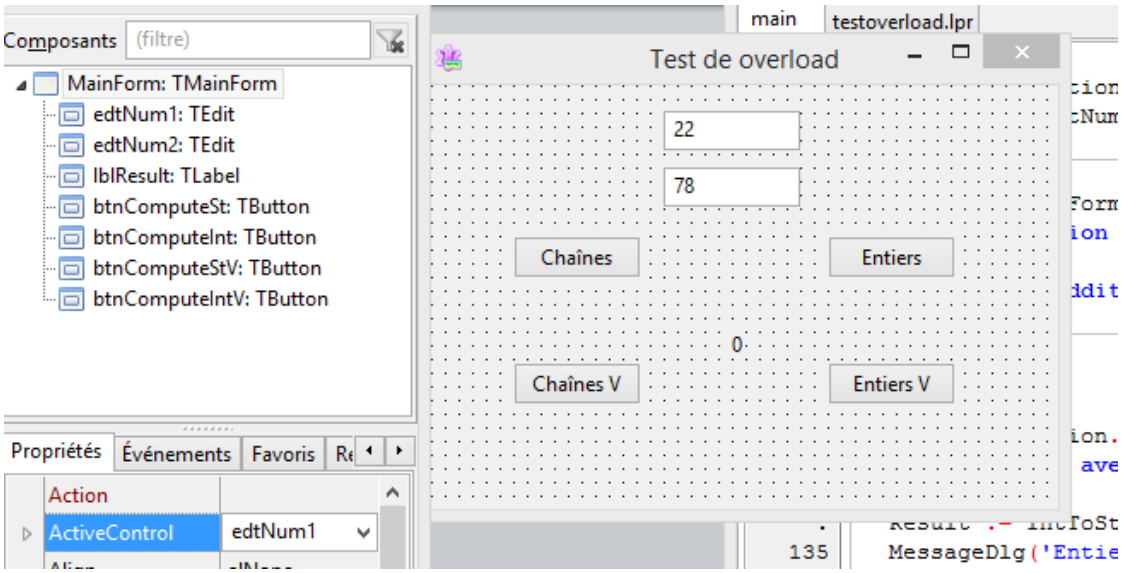
```
// *** addition avec méthode virtuelle ***
begin
  Result := IntToStr(Nombre1 + Nombre2);
  MessageDlg('Entiers (méthode virtuelle)...', 'Addition d''entiers effectuée',
    mtInformation, [mbOK], 0);
end;

{ TAdditionPlus }

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

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

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



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

```
implementation

{$R *.lfm}

{ TMainForm }

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

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

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

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

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

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

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

BILAN

Dans ce chapitre, vous avez appris à :

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