# POO à gogo : Les méthodes

**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*.

## Méthodes 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 :

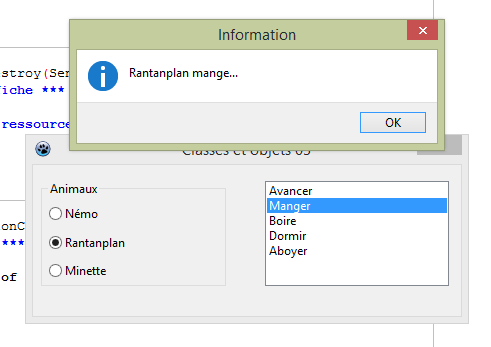
procedure TChien.Manger;

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

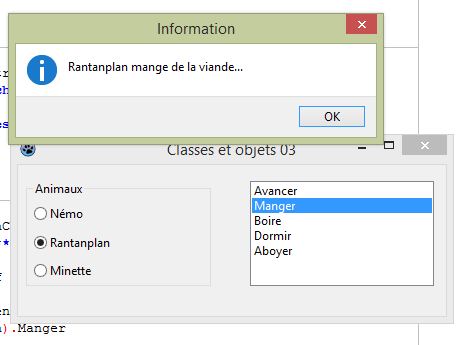
(UnAnimal as TChien).Manger

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 ?

## Méthodes 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[[1]](#footnote-1) 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[[2]](#footnote-2).

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

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[[3]](#footnote-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 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[[4]](#footnote-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 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 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.

## Compléments 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);

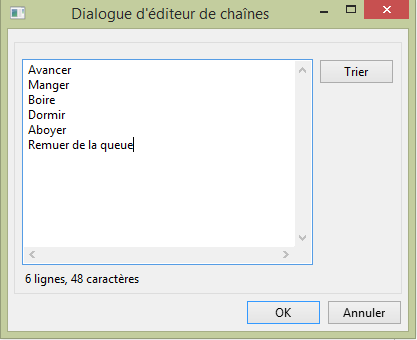
end;

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.

## Méthodes 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[[5]](#footnote-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

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;

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 !

## Méthodes 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.

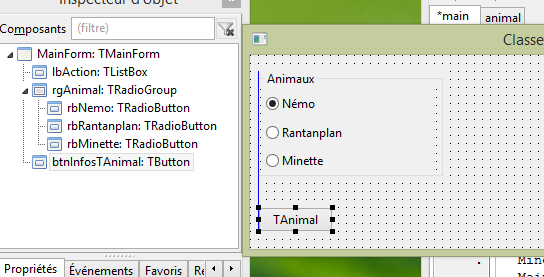
Leurs limites découlent de leur définition même : comme elles sont indépendantes de l’instanciation, elles ne peuvent pas avoir accès aux champs, propriétés et méthodes ordinaires de la classe à laquelle elles appartiennent. De plus, depuis une méthode de classe, *Self* pointe non pas vers l’instance 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 +

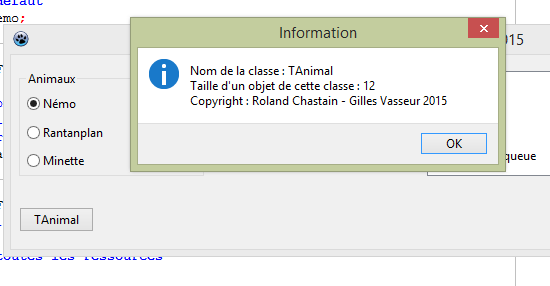
#13#10'Taille d''un objet de cette classe : ' + IntToStr(TAnimal.InstanceSize) +

#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;

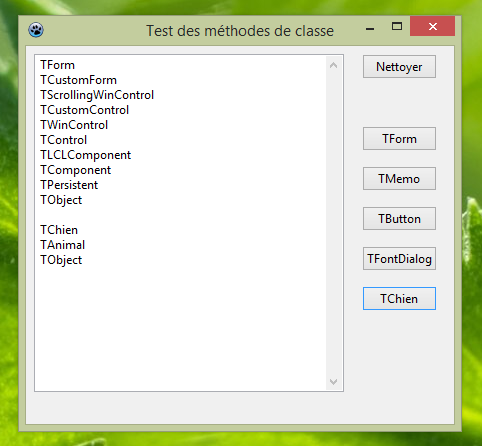
mmoDisplay.Lines.Add('');

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*.

## Méthodes de classe statiques

## Méthodes de message

1. Cette table porte le nom de VMT : *Virtual Method Table*. [↑](#footnote-ref-1)
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. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
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 ! [↑](#footnote-ref-4)
5. Une des classes les plus utilisées, *TStringList*, a pour ancêtre *TStrings*. [↑](#footnote-ref-5)