Delphi – Partie 1
Le Pascal Objet

Ecrit par Jean-Paul Pruniaux

Objectif

Comprendre le langage Pascal Objet de Delphi et pouvoir écrire des programmes simples (en mode caractère). Ce cours est le premier d'une série de cours destinés à former des programmeurs Delphi.

Niveau requis

L'étudiant doit déjà être un programmeur Pascal (Turbo-Pascal ou Borland Pascal) et être un bon utilisateur d'un système informatique personnel sous *Windows*.

Durée

3 jours.

Droits d'auteur

TABLE DES MATIERES

1. LA PRESENTATION DE DELPHI

Delphi est un environnement de développement qui a été conçu par la société *Borland* (devenue *Inprise*) pour créer des programmes sous *Windows*.

Tout type de programme pour *Windows* peut être conçu avec *Delphi*, qu'il s'agisse de programme de calcul scientifique, techniques ou autres. Cependant il faut signaler que la bibliothèque livrée avec *Delphi* permet aussi de créer facilement des applications de gestion s'appuyant sur des bases de données.

Autant le langage de programmation, que la bibliothèque de composants et l'environnement de développement mettent en œuvre une organisation de la programmation qui s'est largement répandue cette dernière décennie : la *Programmation Orientée Objets* ou *POO*.

A ce stade, considérons que la *POO* est un modèle d'organisation de la programmation dont un atout majeur est une grande *modularité* dans l'écriture des programmes. A cause de cette modularité, les programmes deviennent beaucoup plus faciles à écrire, les problèmes complexes peuvent être découpés en petits problèmes faciles à traiter, les solutions sont alors facilement assemblées, le code devient plus facile à maintenir…

Delphi s'appuie sur un compilateur *Pascal Objet*. Ce compilateur met un œuvre un langage *Pascal* relativement pur (tel que celui de Turbo Pascal ou de Borland Pascal), auquel quelques extensions du langage permettent de mettre en œuvre tous les principes de la *POO*.

La *POO* est le thème majeur de cette partie du cours sur *Delphi*, et ce manuel décrit toutes les parties fondamentales du *Pascal Objet* de *Delphi*. En effet, Delphi est composé de 3 éléments principaux :

- Un compilateur *Pascal Objet*.
- Une bibliothèque de composants écrite en *Pascal Objet*.
- Un environnement de développement graphique lui-même conçu en *Pascal Objet* et utilisant la bibliothèque de composants.

Cette décomposition fait ressortir un fait majeur : pour comprendre Delphi, il faut comprendre la *POO* et le *Pascal Objet*.

D'après notre expérience, la plupart des ouvrages et des formations sur *Delphi* omettent ce fait majeur et ne présentent pas les principes fondamentaux de la *POO* et du *Pascal Objet*. Ou lorsqu'ils les présentent correctement, c'est à la fin de l'ouvrage ou du cours. Ceci établit les raisons de ce manuel : l'importance de construire des bases solides afin de retirer tout le bénéfice d'un environnement de développement comme *Delphi*.

Vérification de la compréhension

Comment expliqueriez-vous simplement ce qu'est la Programmation Orientée Objet ?

Faites un schéma représentant les 3 composants fondamentaux de *Delphi* et leur relation.

2. L'UTILISATION DE DELPHI DANS CE MANUEL

Pour étudier et prendre en main le *Pascal Objet* de *Delphi*, nous ne réaliserons dans ce manuel que des applications en mode texte. En effet, la réalisation de programme pour *Windows* est une tâche relativement complexe sans l'utilisation de bibliothèque fondée sur les principes de la *POO* ; et l'utilisation d'une telle bibliothèque ne peut se concevoir sans comprendre le langage qui permet de la manipuler.

Une application en mode texte se contentera d'afficher des messages dans une fenêtre de texte et permettra éventuellement la saisie de données depuis le clavier dans cette fenêtre.

Cependant nous commencerons par nous faire une idée rapide de la manière de réaliser une application *Windows* afin de comprendre les divers éléments de l'interface de *Delphi*.

2.1 ORIENTATION DANS DELPHI

Commencez par démarrer *Delphi* en sélectionnant dans le menu système de *Windows* la commande suivante : « *Démarrer / Programmes / Borland Delphi4 / Delphi 4* » ou en cliquant sur l'icône adéquate. L'écran doit prendre une apparence qui ressemble à ceci :

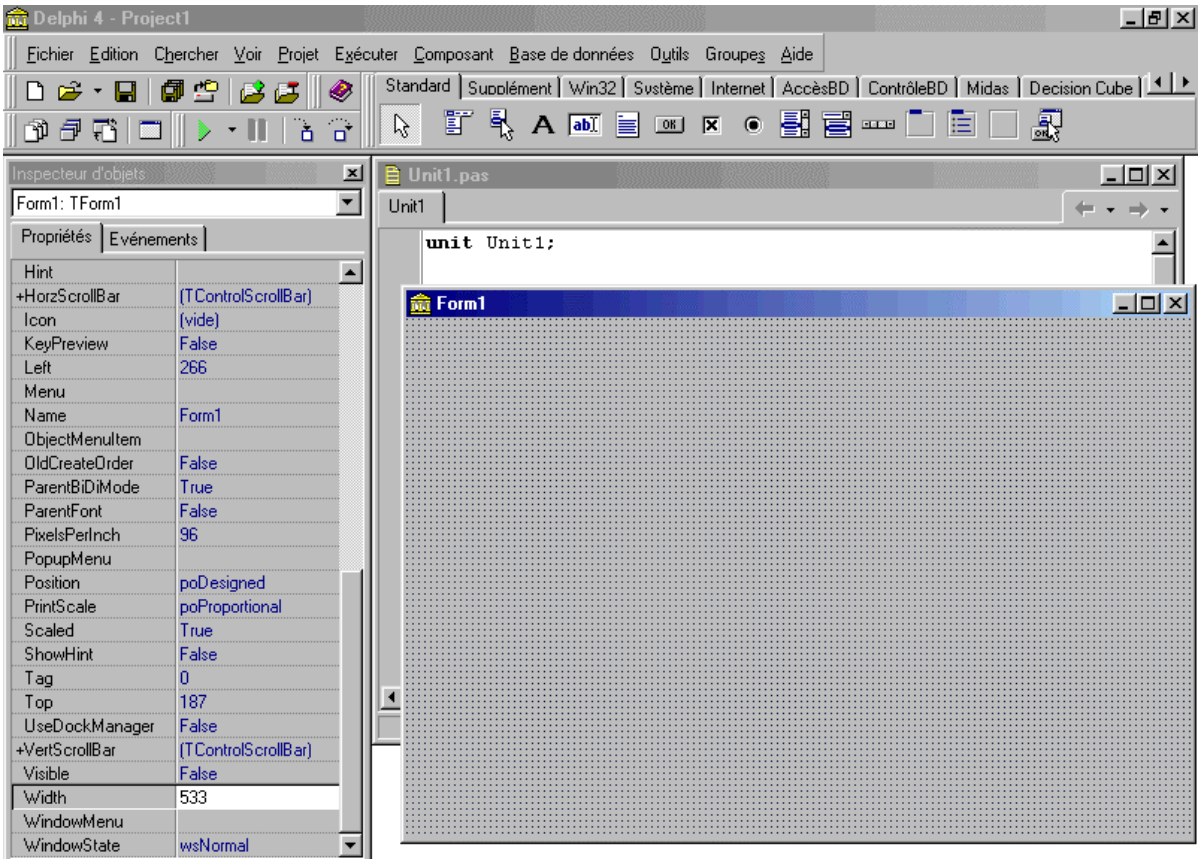


Figure 1. L'environnement de travail de Delphi

Remarque

Lorsque vous démarrez *Delphi*, l'apparence de l'écran peut changer en fonction de la configuration sur votre machine et en fonction des derniers travaux réalisés avec *Delphi*.

Il peut arriver que Delphi vous présente la dernière application en cours de conception — Auquel cas, utilisez la commande « *Fichier / Nouvelle application* » pour démarrer un nouveau projet.

En haut de l'écran se trouve la barre d'outils de Delphi, outils principal de son utilisation. Cette barre d'outils comporte les éléments suivants :

- Un menu rassemblant toutes les commandes possibles.
- Sous le menu à gauche, une barre d'outils avec les icônes servant de raccourci pour les commandes les plus fréquemment utilisés.
- Sous le menu à droite, la *palette de composants*. Celle-ci constitués d'onglets (« *Standard* », « *Suppléments* », « *Win32* »…) permet d'insérer facilement n'importe quel composant graphique de la bibliothèque de Delphi dans une fenêtre de programme.

Remarque

De nombreux menus contextuels, accessibles par le bouton gauche de la souris permettent d'envoyer des commandes particulières à un objet de l'environnement de travail.

Sur la gauche de cet écran se trouve l'*inspecteur d'objet*. Celui-ci permettra de paramétrer n'importe quel composant inséré dans une fenêtre en cours de développement.

Sur la droite de cet écran se trouvent deux fenêtres « *Form1* » et « *Unit1.pas* » :

- La fenêtre actuellement appelée « *Form1* » est destinée à concevoir graphiquement le contenu de la fenêtre principale de la nouvelle application. Dans cette fenêtre vous pouvez insérer des composants à partir de la *palette de composants*.
- Le fenêtre actuellement appelée « *Unit1.pas* » contient des éditeurs de texte destinés à concevoir ou modifier le source du programme et de ses unités.

2.2 LES COMMANDES FONDAMENTALES

2.2.1 La sauvegarde des sources

Avant d'exécuter une application, nous vous engageons à sauvegarder tous les éléments constituant le projet :

Utilisez si nécessaire la commande « *Fichier / Tout enregistrer* » pour enregistrer tous les modules sources du projet et son paramétrage.

Remarque

Afin de faciliter la gestion de l'ordinateur avec lequel vous travaillez pendant cette formation, veuillez créer un répertoire à part et y stocker tous les éléments que vous développez.

Remarque

Par défaut, un projet est constitué de deux éléments principaux :

- Un premier source dont le nom par défaut est « *Unit1* » contenant la définition et le paramétrage de la fenêtre principale — Ce source est en fait constitué de 2 fichiers dont les noms par défaut sont « *Unit1.pas* » (le source *Pascal*[1]) et « *Unit1.dfm* » (le paramétrage de la fenêtre principale avec ses valeurs par défaut[2]) — Ces deux fichiers sont indissociables. Lors de l'enregistrement de ces fichiers, une boîte de dialogue d'enregistrement de fichier standard sous Windows apparaît : choisissez un répertoire et un nom de fichier.
- Un deuxième source dont le nom par défaut est « *Project1.dpr*[3] » contenant le programme principal. Il s'agit d'un source Pascal commençant par l'instruction « `program` ». Lors de l'enregistrement de ces fichiers, une boîte de dialogue d'enregistrement de fichier standard sous Windows apparaît : choisissez un répertoire et un nom de fichier.

Pratique

Avec l'explorateur de *Windows*, créez un répertoire dans lequel stocker tous les exemples qui seront développés dans cette formation.

Sauvegardez les sources d'une nouvelle application dans ce répertoire ; appelez le source de la fenêtre principale « *PremiereFenetre.pas* » et le programme « *PremierProgramme.dpr* ».

2.2.2 La compilation et l'exécution d'une application

Pour compiler et exécuter le programme, utilisez la commande « Exécuter / Exécuter », appuyez sur la touche F9, ou cliquez sur l'icône ci-contre. 

Delphi vérifie alors tous les sources du projet et compile ceux-ci si la date du source est plus récente que la date du module compilé.

Delphi compile alors le source principal de l'application puis édite les liens.

Finalement *Delphi* lance l'application. Sa fenêtre principale s'ouvre et vous pouvez tester le programme.

---

[1] Un source Pascal est un fichier texte contenant des instructions Pascal – Une unité Pascal comporte l'extension « *pas* ».

[2] « *DFM* » signifie « *Delphi ForM* » ou « *Fiche Delphi* » — Un fichier d'extension « *DFM* » contient toutes les valeurs par défaut permettant d'initialiser correctement une fenêtre lors de sa création à l'exécution.

[3] « *DPR* » signifie « *Delphi PRoject* » ou « *Projet Delphi* » — Un fichier d'extension « *DPR* » est un source Pascal contenant le programme principal d'une application *Delphi*.

Remarque

La commande « *Projet / Compiler le projet*[4] » vérifie quels sources ont besoin d'être compilés, puis compile le programme principal et édite les liens (réalisation des deux premières étapes précédentes).

La commande « *Projet / Construire le projet* » compile tous les sources du projet et édite les liens.

Delphi 4 permet d'ouvrir plusieurs projets en même temps ; les commandes « *Projet / Compiler tous les projets* » et « *Projet / Construire tous les projets* » s'appliquent de la même manière à tous les projets chargés.

Remarque

Voici une liste des principaux fichiers gérés par *Delphi* :

- « *.DPR » : projet Delphi — Source Pascal principal d'une application.
- « *.RES » : Ressources[5] du projet.
- « *.PAS » : source Pascal.
- « *.DFM » : Delphi Form : valeurs initiales associées à une fiche.
- « *.DCU » : Delphi Compiled Unit ; version compilée d'un fichier « *.PAS ».
- « *.EXE » : Programme exécutable généré par Delphi. A priori c'est le seul fichier à livrer au client[6].

Pratique

Exécutez l'application que vous venez de sauvegarder — Constatez que vous avez affaire à une fenêtre *Windows* standard, bien que vierge.

Puis fermez cette fenêtre pour revenir à *Delphi*.

2.2.3 Naviguer dans Delphi

Delphi gère un projet au travers de diverses fenêtres, il faut très rapidement savoir passer d'une fenêtre à une autre et notamment faire apparaître la fenêtre dont on a besoin :

La commande « *Voir / Basculer fiche/unité*[7] » permettent de faire l'aller retour entre une fenêtre en cours de construction et le source correspondant.

La commande « *Voir / Inspecteur d'objets*[8] » permettent de faire l'aller retour entre une fenêtre en cours de construction, l'*inspecteur d'objet* et le source correspondant.

---

[4] Cette commande est aussi accessible avec les touches Ctrl F9.

[5] Une ressource correspond à une données ou un ensemble de données gérées par *Windows* (par exemple un menu, une icône…). Les ressources sont intégrées et liées à l'exécutable généré par Delphi.

[6] Dans la mesure où le programme ne fait pas appel à des librairies dynamiques.

[7] Cette commande est aussi accessible avec la touche F12.

La commande « *Voir / Fiche*[9] » montre la liste de toutes les fiches constituant un projet — Sélectionnez la fiche adéquate et validez la boîte de dialogue pour voir cette fiche.

La commande « *Voir / Unité*[10] » montre la liste de tous les sources *Pascal* constituant un projet — Sélectionnez le source adéquat et validez la boîte de dialogue pour voir ce source.

La commande « *Voir / Gestionnaire de projet* » montre dans une nouvelle fenêtre tous les éléments constituant les projets chargés. Ceux-ci sont classés logiquement. Double cliquez sur l'élément vers lequel vous désirez aller. *Delphi* peut stocker un environnement de travail dans un fichier « *Groupe de projets* » (*Project Group*) ; ce groupe de projet peut faire référence à plusieurs projets — Ceci explique le premier item de la fenêtre : « *ProjectGroup1* ».

Pratique

Tester les commandes précédentes ; n'hésitez pas à charger le système d'aide[11] pour obtenir d'avantage d'informations sur ces commandes si vous le souhaitez.

Modifiez le titre de la fenêtre principale grâce à l'inspecteur d'objet. Pour ceci :

- Commencez par mettre cette fenêtre en avant plan.
- Basculez vers l'inspecteur d'objet.
- Dans l'inspecteur d'objet réalisez les opérations suivantes :
  - Cliquez sur l'onglet « *propriété*[12].».
  - Dans la liste des propriétés, trouver le mot « *Caption*[13] » pour modifier le titre de la fenêtre ; puis en face, tapez une nouvelle valeur ; par exemple « *Ma première fenêtre* ».
  - Sauvegardez et exécutez le projet.

[8] Cette commande est aussi accessible avec la touche F11 .

[9] Cette commande est aussi accessible avec les touches ⇧ Shift F12.

[10] Cette commande est aussi accessible avec les touches Ctrl F12.

[11] En appuyant sur la touche F1

[12] Une *propriété* correspond à une caractéristique ou à un paramètre d'un composant.

[13] « *Caption* » peut se traduire par « *Titre* » ou « *Sous Titre* » ou « *Légende* ».

2.2.4 Placer des composants dans une fenêtre

Pour placer des composants dans une fenêtre, commencez par mettre cette fenêtre en avant-plan en utilisant les commandes de la section précédente.

Sélectionnez l'onglet adéquat dans la *palette de composants*, cliquez sur le composant désiré, puis cliquez dans la fiche.

Eventuellement, lorsque vous cliquez dans la fiche vous pouvez tracer le rectangle d'encombrement du composant inséré.

Par exemple pour insérer un bouton :

- Mettez la fenêtre en avant plan.
- Cliquez sur l'onglet « *Standard* » de la *palette de composants*.
- Cliquez sur l'icône  .
- Cliquez en haut à gauche de la fiche et maintenez le bouton de la souris enfoncé.
- Déplacez la souris un peu en bas à droite.
- Relâchez le bouton de la souris.
- Basculez vers l'*inspecteur d'objet*.
- Modifiez la propriété « *Caption* » de ce bouton pour afficher « *Fermer* ».

Remarque

Vous pouvez insérer plusieurs fois le même composant dans une fiche ; pour ceci appuyez sur la touche  en même temps que vous cliquez sur l'icône du composant à insérer. Puis vous cliquez à plusieurs endroit de la fiche.

Lorsque tous les composants sont insérés, cliquez sur l'icône  .

Remarque

Certains composants ne sont visible que lors de la conception de la fiche et ne le seront plus à l'exécution du programme. Il s'agit de composants qui ont été conçu pour paramétrer facilement des fonctionnalités « invisibles ». Nous trouvons par exemple tous les composants de l'onglet « *AccèsBD* » permettant d'accéder aux bases de données.

D'autres composants sont visibles aussi bien en conception qu'à l'exécution. Nous parlons alors de *contrôles*. Nous trouvons par exemple des boutons, des cases à cocher, des zones de saisie…

Une fois qu'un composant a été posé sur une fiche, vous pouvez cliquer sur celui-ci pour le sélectionner. Vous pourrez alors le déplacer ou en modifier la taille grâce aux opérations classiques de travail à la souris sous *Windows*.

Avant de paramétrer un composant avec l'inspecteur d'objet, vous devez le sélectionner.

Pratique

Concevez l'aspect graphique de la fenêtre suivante :

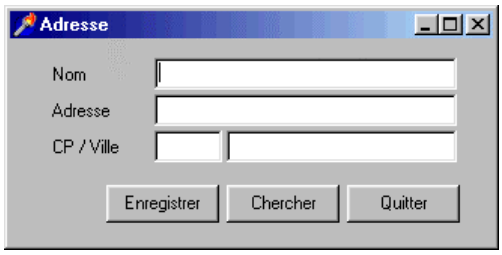


Figure 2. Exemple de fenêtre à concevoir

Pour réaliser cette fenêtre, vous aurez besoin des composants suivants de la palette standard de composants :

Trois boutons accessibles avec l'icône  .

Trois étiquettes (labels) accessible avec l'icône  .

Quatre zones de saisie (Edit) accessibles avec l'icône  [14].

[14] Pour éditer le contenu d'un composant « *Edit* » avec l'inspecteur d'objet, modifiez la propriété « *Text* » et non la propriété « *Caption* ».

2.3 LA REALISATION DE PROGRAMME EN MODE TEXTE AVEC DELPHI

Bien que nous soyons maintenant capables de construire les aspects visuels d'une application avec *Delphi*, nous sommes incapables d'associer quelque programme que ce soit à ces écrans. Aussi devons nous commencer à étudier le *Pascal Objet*. Pour ce faire, nous allons réaliser des exemples et des applications en mode texte.

A partir de la version 2, *Delphi* permet de réaliser des applications en mode texte, c'est à dire dans une fenêtre *DOS*. Pour ceci, nous devons apprendre à paramétrer l'environnement de travail pour réaliser de telles applications.

2.3.1 La réalisation d'application en mode texte avec Delphi 5

Avec Delphi 5, la réalisation d'une application texte peut se faire simplement de la manière suivante :

- Enregistrez tout travail en cours avec « *Fichier / Tout enregistrer* ».
- Lancez la commande « *Fichier / Nouveau…* » qui ouvre la boîte de dialogue suivante :

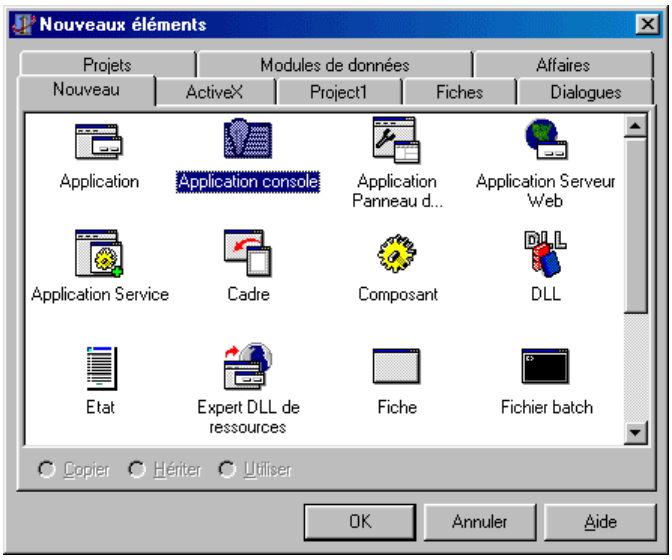


Figure 3. Nouvelle application console

- Choisissez « *Application console* », un nouveau projet est créé, correctement paramétré.

Terminons la création du projet en l'enregistrant :

- Utilisez la commande « *Fichier / Tout enregistrer* ».
- Sélectionnez le dossier dans lequel stocker vos programmes et donner un nom au projet.

Dans l'éditeur de code, vous observez le source suivant :

```
program Premier;
{$APPTYPE CONSOLE}
uses sysutils;

begin
  // Insérer le code utilisateur ici
end.
```

Il s'agit d'un pur source *Pascal* qui porte le nom du projet[15].

La directive de compilation « `$APPTYPE CONSOLE` » indique au compilateur qu'il s'agit d'une application console (voyez éventuellement le cours sur le *Borland Pascal* pour voir ce qu'est une directive de compilation). Cette directive de compilation peut aussi être remplacée par le paramétrage adéquat du projet (Voir la boîte de dialogue « Figure 4. Le paramétrage d'un projet pour la réalisation d'une application en mode texte » ci-après.

A ce stade, la clause « `uses` » n'est pas fondamentale, et le compilateur pourrait s'en passer ; mais elle peut être nécessaire à l'environnement de travail de *Delphi*. En effet dans certains cas, lorsqu'on travaille le programme avec l'environnement graphique celui-ci peut chercher à faire des modifications de code, et cette balise « `uses` » lui est alors nécessaire.

[15] *Delphi* corrige automatiquement le nom associé à la déclaration « `program` » avec le nom du source à chaque fois que vous enregistrez le fichier sous un autre nom.

2.3.2 La réalisation d'application en mode texte avec Delphi 4

Avec Delphi 4, la réalisation d'une application texte demande un peu plus de paramétrage :

- Enregistrez tout travail en cours avec « *Fichier / Tout enregistrer* ».
- Créez une nouvelle application avec « *Fichier / Nouvelle application* ».
- Modifier les paramètres du projet avec la commande « *Projet / Options* » — Une nouvelle boîte de dialogue apparaît — cliquez alors sur l'onglet « *Lieur* » et cochez la case « *Créer une application console* ».

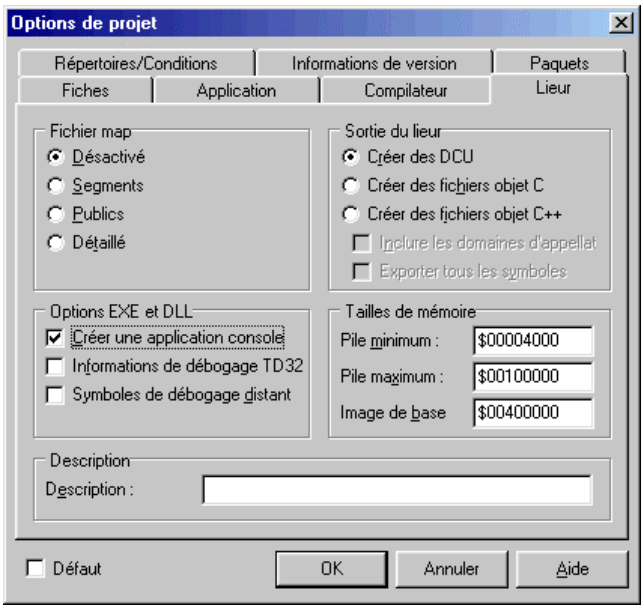


Figure 4. Le paramétrage d'un projet pour la réalisation d'une application en mode texte

Une fenêtre texte est maintenant associée au projet.

Nous allons maintenant supprimer la fenêtre de style *Windows* du projet. Pour ceci :

- Utilisez la commande « *Projet / Retirer du projet* » ou cliquez sur l'icône  .
- Une fenêtre apparaît avec la liste de toutes les unités constituant le projet — A ce stade elle ne doit contenir que l'unité « *Unit1* » (et la fiche « *Form1* »).
- Vérifiez que la ligne correspondant à cette unité soit sélectionnée et validez la boîte de dialogue.
- Si par hasard *Delphi* affiche une boîte de dialogue « *Enregistrez les modifications dans ….pas ?* », répondez « *Non* » — Cela signifie que vous aurez pu changer certains paramètres de la fenêtre que nous retirons (taille ou position par exemple). Changements que nous pouvons ignorer.

Normalement le projet ne doit plus contenir qu'un seul source qui sera le source du programme principal. Enregistrons ce source :

- Utilisez la commande « *Fichier / Tout enregistrer* ».
- Sélectionnez le dossier dans lequel stocker vos programmes et donner un nom au projet.
- Chargez ce source dans l'éditeur de texte (Par exemple avec les touches Ctrl F12 pour avoir la liste des sources).

Examinons maintenant ce source :

```
program Premier;

uses
  Forms;

{$R *.RES}

begin
  Application.Initialize;
  Application.Run;
end.
```

Il s'agit d'un pur source *Pascal* qui porte le nom du projet[16]. Nous allons maintenant retirer de ce source une partie du code non nécessaire qui y a été mis automatiquement par *Delphi*. Le source doit devenir :

```
program Premier;

uses
  Forms;

begin

end.
```

A partir de là nous avons un pur source Pascal à partir duquel nous pouvons concevoir une application en mode texte.

Remarque

Dans ce source, nous avons du conserver l'instruction « `uses Forms ;` ». Cette ligne n'est pas nécessaire pour le compilateur mais elle peut être nécessaire à *Delphi* dans certains cas lorsque l'on travail le programme dans l'environnement graphique.

La déclaration « `program` » doit porter le même nom que le fichier dans lequel est stocké le programme.

Les instructions du programme principal pourront être écrites entre le « `begin` » et le « `end` ».

[16] *Delphi* corrige automatiquement le nom associé à la déclaration « `program` » avec le nom du source à chaque fois que vous enregistrez le fichier sous un autre nom.

2.3.3 Un programme minimum

Dans nos premiers exemples il sera bien souvent nécessaire de terminer le programme principal par une instruction « `ReadLn ;` » afin que le programme attende la frappe de la touche ↵ avant de refermer la fenêtre console.

Le programme devient alors :

```
program Premier;

uses Forms;  // ou // uses sysutils;

begin

  ReadLn;
end.
```

Faisons maintenant le programme minimum qui affiche un message « *Bonjour* » :

```
program Premier;

uses
  Forms;

begin
  WriteLn ('Bonjour');
  ReadLn;
end.
```

Pratique

Réalisez votre premier programme en mode caractère et faites-lui afficher quelque chose.

Aucune fenêtre Windows ne doit s'ouvrir à l'exécution de ce programme.

Retirez l'instruction finale « `ReadLn ;` » et observez ce qu'il se passe à l'exécution.

Ajouter les déclarations et instructions *Pascal* nécessaires au programme afin d'afficher 10 fois « *Bonjour* » avec une boucle « *FOR* ».

2.4 RAPPEL SUR LE DECOUPAGE EN UNITES EN PASCAL

En *Pascal*, les sources sont organisés à partir d'un *programme principal* qui a une structure donné et d'autres sources, les *unités* possédant une autre structure.

Dans sa forme la plus simple un programme principal a la structure suivante :

```
program NomDuProgram;

{ Déclarations }

begin
  { Première Instruction Exécutée }
  { Deuxième Instruction Executée }
  { etc..}
end.
```

Et une unité a la structure suivante :

```
unit NomDeLUnité;

interface[17]

{ Déclarations visibles depuis l'extérieur de l'unité }

implementation[18]

{ Declarations visibles uniquement dans cette unité }
{ Contenu des procédures et des fonctions }

end.
```

Lorsqu'un programme principal utilise dans ses déclarations, ou ses instructions des éléments déclarés dans la partie « `interface` » d'une unité, cette unité doit faire partie de la liste des unités décrites dans la déclaration « `uses` ».

Par exemple avec l'unité « *Affichage.pas* » :

```
unit Affichage;

interface

procedure FaireCoucou;       { Nom de la procédure exportée }

implementation

procedure FaireCoucou;       { Contenu de la procédure exportée }
  begin
  WriteLn ('Coucou');
  end;

end.
```

[17] Le mot clef « *interface* » définit dans le source la section de l'unité qui sera visible à l'extérieur de cette unité. « *Interface* » est un mot anglais qui signifie « point d'interaction entre deux systèmes ». Dans cette section on définit la manière dont on peut communiquer avec l'unité.

[18] Le mot clef « *implementation* » définit dans le source la section « privé » de l'unité. « *Implementation* » est un mot anglais qui signifie « mise en œuvre ».

Le programme principal peut être :

```
program AppelCoucou;
uses Forms, Affichage;        { Etablir le lien avec l'unité }
begin
  FaireCoucou;                { Appel de la procédure }
  ReadLn;
end.
```

De la même manière une unité peut faire référence à une autre unité, soit dans « l'*interface* », soit « l'*implementation* ». Par exemple :

```
unit AffichageComplexe;
interface
procedure Faire10Coucou;      { Nom d'une autre procédure exportée }
implementation
uses Affichage;               { Lien vers l'unité "Affichage"}
procedure Faire10Coucou;      { Contenu de la procédure exportée }
  var
    i : integer;
  begin
  for i := 1 to 10 do
    FaireCoucou;
  end;
end.
```

Voici son utilisation avec un autre programme principal :

```
program Appel10Coucou;
uses Forms, AffichageComplexe;     { Etablir le lien avec l'unité }
begin
  Faire10Coucou;                   { Appel de la procédure }
  ReadLn;
end.
```

Remarque

Lorsque la déclaration « `uses` » est présente dans un programme principal, elle doit obligatoirement suivre la déclaration « `program` ». Aucune autre déclaration ne peut être admise entre « `program` » et « `uses` ».

De la même manière, lorsque la déclaration « `uses` » est présente dans l'interface d'une unité, elle doit obligatoirement suivre la déclaration « `interface` ».

De la même manière, lorsque la déclaration « `uses` » est présente dans l'implementation d'une unité, elle doit obligatoirement suivre la déclaration « `implementation` ».

Une unité peut avoir une déclaration « `uses` » dans sa section « `interface` » et dans sa section « `implementation` ».

Avant d'utiliser une déclaration définie dans la section « `interface` » d'une unité, cette unité doit être déclarée dans l'instruction « `uses` ».

Vérification de la compréhension

Que signifie les termes « *interface* » et « *implementation* » et quels sont leurs rôles dans l'écriture d'une unité ?

Que signifie le terme « *uses* » et quel est son rôle en *Pascal* ?

2.5 LA GESTION DES UNITES AVEC DELPHI

L'environnement de travail de *Delphi* permet d'intégrer de nouvelles unités ou des unités existantes à un projet. Ceci facilite la gestion des sources au sein d'un projet.

2.5.1 La création d'une nouvelle unité

Pour créer une nouvelle unité :

- Utilisez la commande « *Fichier / Nouveau…* » ou cliquez sur l'icône  . Une boîte de dialogue apparaît pour préciser le nouvel élément que nous désirons.
- Cliquez si nécessaire sur l'onglet « *Nouveau* » puis sélectionnez et cliquez sur l'icône « *Unité* » :

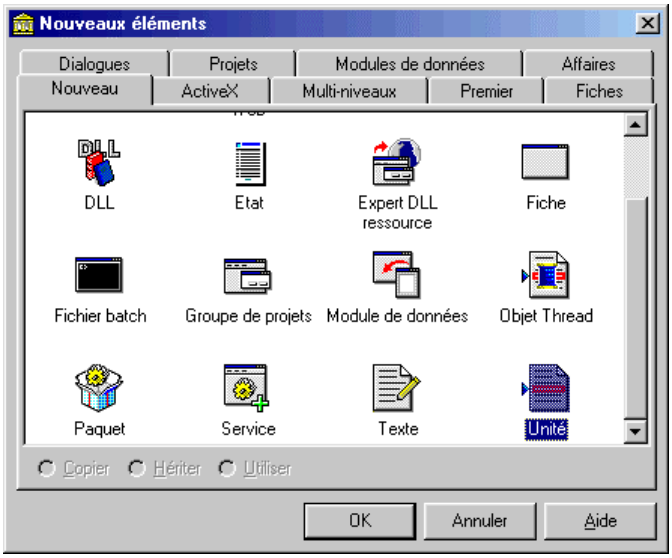


Figure 5. La création d'une nouvelle unité dans un projet.

- Un nouveau source est créé dans l'éditeur de source. Nous vous conseillons de l'enregistrer tout de suite afin de donner un nom cohérent à l'unité.
- Utilisez la commande « *Fichier / Enregistrer* », l'icône  ou les touches Ctrl+'*S*' pour enregistrer le source et lui donner un nom de fichier cohérent.

A ce stade, examinez le source principal pour voir le travail qu'a réalisé *Delphi* :

```
program Deuxieme;

uses
  Forms,
  Affichage in '..\Program Files\Borland\Delphi4\Projects\Affichage.pas',

begin

  ReadLn;
end.
```

Remarque

La déclaration *Pascal* « `uses` » comprend une extension « `in`[19] » permettant de gérer les noms de fichiers longs et l'arborescence sous *Windows*. Ceci est géré automatiquement par *Delphi* et lui permet de retrouver les fichiers concernés sur votre système. Dans toutes les autres déclarations « *uses* », il est inutile de gérer l'extension « `in` ».

Remarque

Lorsque dans un source *Pascal*, vous utilisez les touches Ctrl ↵ alors que le curseur est sur un nom d'unité d'une déclaration « `uses` », *Delphi* charge automatiquement le source concerné ou ouvre la fenêtre dans laquelle cette unité est déjà chargée.

Pratique

Réalisez les exemples de la section *2.4.* Rappel sur le découpage en unités en Pascal en utilisant ces principes.

2.5.2 La réutilisation d'une unité existante

Pour intégrer une unité qui a déjà été écrite à un projet :

- Utilisez la commande « *Projet / Ajouter au projet* », l'icône  ou les touches ⇧ Shift F11 .
- Une boîte d'ouverture de fichier apparaît pour que vous sélectionnez l'unité concernée.

Lorsque vous avez choisi cette unité et avez validé la boîte de dialogue précédente, l'unité fait maintenant partie du projet.

2.6 LA GESTION DES COMMENTAIRES DANS UN SOURCE

Les commentaires traditionnels en *Pascal* se mettent entre accolades — `{ Commentaires }` — ou entre (* et *) — `(* commentaire *)`.

Delphi supporte aussi les commentaires à la mode du « *C++* ». Ceux ci commencent par un double slash — // — et vont jusqu'en fin de ligne. Par exemple :

```
program AppelCoucou;

uses Forms, Affichage;       // Etablir le lien avec l'unité

begin
  FaireCoucou;               // Appel de la procédure
  ReadLn;
end.
```

A ce stade nous savons réaliser des applications *Pascal* en mode texte et gérer un projet constitués de plusieurs sources. Nous allons maintenant étudier les concepts fondamentaux de la *POO* et leur mise en œuvre en *Pascal Objet*.

[19] Il s'agit d'une extension au langage *Pascal* standard.

3. CONCEPTS FONDAMENTAUX DE L'ORIENTATION OBJETS

3.1 QU'EST-CE QU'UN OBJET ?

3.1.1 Définition du terme « *objet* »

Définition théorique : Un objet est constitué :

- D'un *état*
- De *comportements*
- D'une *identité*.

Etat

« *Situation ou manière d'être d'une personne ou d'une chose, plus ou moins durable, mais considéré comme relativement stable, par opposition à une période ou une phase de transformation, de mouvement, d'action* » (Dictionnaire Logos de Bordas).

« *L'état d'un objet englobe toutes les propriétés (habituellement statiques) de l'objet, plus les valeurs courantes (habituellement dynamiques) de chacune de ces propriétés.* » (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Une *propriété* est une caractéristique d'un objet, son état se reflète donc grâce à l'ensemble des caractéristiques de l'objet.

Exemple

Une fenêtre *Windows* peut être caractérisée par sa position (x et y), sa taille (hauteur et largeur), sa couleur de fond...

Une lampe peut être caractérisée par le fait qu'elle soit allumée ou éteinte, qu'elle consomme telle puissance.

Un fichier peut être caractérisé par un nom, son état ouvert/fermé, son mode d'accès (lecture/écriture/accès partagé...), la position du pointeur de fichier.

Comportement

« *Conduite, manière d'agir ou de réagir* » (Dictionnaire Logos de Bordas).

« *Le comportement est la façon dont un objet agit et réagit, en termes de changement de ses états et de circulation de messages* ». (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Exemple

Une fenêtre peut être déplacée (elle suit le mouvement de la souris); elle peut être agrandie (sa taille suit le mouvement de la souris); on peut lui demander de changer de couleur...

Une lampe peut s'allumer ou s'éteindre; en fonction du courant qui passe dedans elle consommera plus ou moins de courant.

Identité

« *L'identité est cette propriété d'un objet qui le distingue de tous les autres objets* ». (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Commentaire

Ne confondez pas la notion d'identité avec celle d'un état permettant d'identifier un objet. Le nom de fichier d'un objet de type fichier ne constitue pas son identité; en effet si on renomme le fichier, on a toujours affaire au même fichier même si le nom n'est pas le même — le nom de fichier fait partie de son état.

L'identité est ce qui permet de dire « cet objet ».

La meilleure approximation que l'on puisse donner de cette notion d'identité en matière de programmation serait une implémentation mémoire ou un nom de variable.

Commentaires

Pour les personnes ayant manipulé d'autres langages de programmation, la notion d'*état* (et de *propriété*) est à rapprocher de celle de variables (ou de structure); celles-ci pouvant servir à mémoriser l'état de l'objet.

La notion de *comportement* est à rapprocher de celle de procédure, de fonction, de sous-programme — bref à des instructions permettant de décrire ce que doit faire un objet donné dans une circonstance donnée.

La notion d'*identité* peut être rapprochée de celle d'implantation mémoire ou de nom de variable.

Ces rapprochements ne sont là que pour aider à appréhender ces notions, mais doivent être pris avec précaution. Par exemple la hauteur et la largeur d'une fenêtre font bien partie de l'état de celle-ci, mais ces informations ne sont peut-être pas mémorisées dans le programme — le concepteur peut avoir décidé de ne mémoriser que la position du coin supérieur gauche et celle du coin inférieur droit de la fenêtre.

3.1.2 Exemples d'objets

Une Fenêtre		
Hauteur	100	Changer la taille
Largeur	200	Déplacer
Position X	500	Mettre en icône
Position Y	150	Mettre en plein écran
Titre	ABC	Changer titre
Couleur	Rouge	Changer couleur

Une autre fenêtre		
Hauteur	150	Changer la taille
Largeur	300	Déplacer
Position X	400	Mettre en icône
Position Y	450	Mettre en plein écran
Titre	XYZ	Changer titre
Couleur	Bleu	Changer couleur

Figure 6. Exemples d'objets

Le schéma ci-dessus montre deux objets de type fenêtre. « Une fenêtre » et « Une autre fenêtre » correspondent à l'identité ; les colonnes de gauche à l'état ; celles de droite aux comportements.

Notez que les deux objets présentés possèdent une nature similaire, ce sont des fenêtres. Une application est en général amenée à gérer des objets de nature différente.

Remarque

Bien que les objets précédemment évoqués soient de nature graphique et visuelle sur un écran, n'en déduisez pas que tous les objets ont un aspect graphique et visuel. Par exemple, on peut trouver dans un programme des objets de type fichiers (et vous ne les verrez jamais sur un écran).

3.1.3 Nature d'un programme en Programmation Orientée Objets

En Programmation Orientée Objets (ou POO) tous les éléments constitutifs de l'application sont représentés sous forme d'objets :

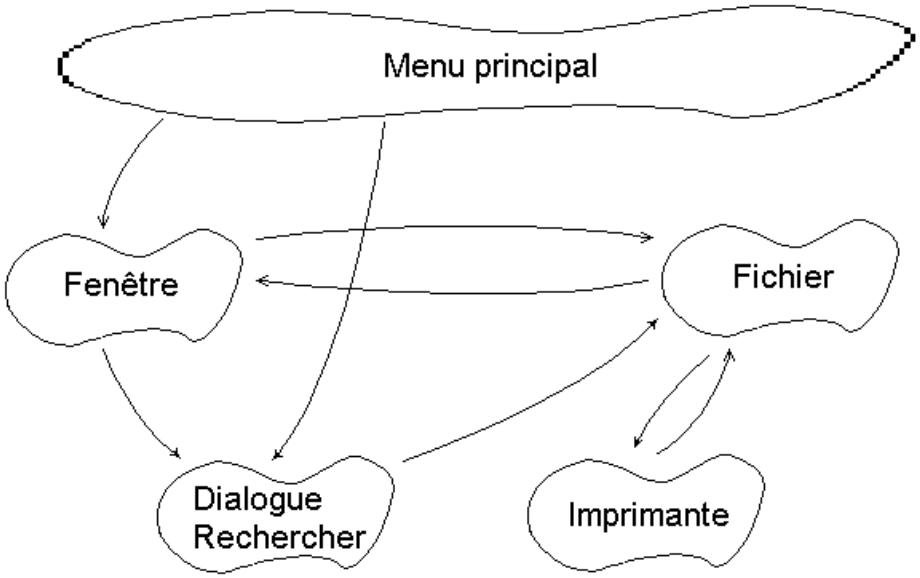


Figure 7. L'organisation d'une application orientée objets

Une application est donc constituée d'un ensemble d'objets que nous faisons collaborer afin d'atteindre un objectif donné.

Remarque

Ceci s'oppose à la forme d'organisation habituellement utilisée avec la programmation structurée — un ensemble de procédures et fonctions hiérarchisées, et en bas de cette pyramide, les données.

Dans le schéma ci-dessus, remarquez que pour faire collaborer les objets entre eux, ceux-ci s'échangent des informations (par exemple pour prévenir un autre objet du fait qu'ils ont changé d'état — on a cliqué sur l'option x du menu) : **Les objets s'envoient des messages**[20].

Un *comportement* est en fait la manière dont un objet réagit à un *message*.

3.1.4 Définition du terme « Instance »

Instance est un terme anglais qui signifie : « *exemple, cas, illustration* » (Traduit du Webster NewWorld Dictionary).

En programmation orientée objet, le terme d'*instance* est utilisé pour désigner un objet particulier ou un objet précis. En fait on pourrait considérer les termes *objet* et *instance* comme étant synonyme — on préférera le terme *objet* lorsqu'on évoque la notion d'objet en général, et le terme d'*instance* lorsqu'on aura affaire à un objet particulier (une fenêtre, un fichier, un contrôle Windows). L'utilisation du terme *instance* évite le côté « truc », « machin » ou « bidule » que le mot *objet* peut évoquer — donc dès qu'on parle de quelque chose de précis dans le cadre de la programmation on préférera ce terme.

Remarque

Le terme d'*instance* donne naissance au verbe *instancier* (créer un objet) et au nom *instanciation* (action de créer un objet — aussi appeler *construction*).

3.2 QU'EST-CE QU'UNE CLASSE?

3.2.1 Définition du terme « Classe »

« *Ensemble d'objets groupés d'après un ou plusieurs caractères communs* » (Dictionnaire Logos de Bordas).

En analyse orientée objets et en *POO*, tous les objets possédant les mêmes caractéristiques sont regroupés en *classes*. Ainsi, nous pouvons trouver la classe des fenêtres, la classe des fichiers, la classe des menus, la classe des options de menu...

3.2.2 Relation entre classes et objets

Les langages comme *Java*, le *Pascal Objet* et le *C++* sont des *langages de classes* — ils demandent, entre autres, à ce que la *classe* d'un objet soit connue et définie avant de pouvoir manipuler un *objet* de la *classe* concernée. En fait la *classe* permet de définir tous les *états* et *comportements* possibles pour les *objets* de cette *classe*.

[20] Le terme de message ici présenté dans le cadre de la *POO*, ne doit pas être confondu avec la notion de message utilisée sous *Windows*. Dans un contexte *Windows*, un *message* est une structure de données que fait circuler le système pour signaler un événement système. La *POO* ne doit pas être confondue avec la programmation événementielle, bien qu'elle simplifie aussi la programmation événementielle.

Remarque

Pour faire un parallèle avec la programmation structurée, une *classe* est l'équivalent d'un *type de données*, une *instance* (un objet) est l'équivalent d'une *variable*.

Exemples

Le schéma, *Figure 6. Exemples d'objets*, montre deux objets, « *Une fenêtre* » et « *Une autre fenêtre* » ayant une nature semblable. Ces objets font partie de la même classe, la classe des « *Fenêtres* ».

Pratique

Parmi les termes ci-dessous quels sont ceux qui illustrent des classes, quels sont ceux qui illustre des objets. Rattachez les objets aux classes auxquelles ils appartiennent. Pour chaque classe trouver des exemples décrivant l'état et les comportements des choses appartenant à ces classes :

« Acteur », « Autant en emporte le vent », « Bateau », « Edgar Poe », « Film », « Le France », « Jules Verne », « Tom Cruse », « 20 000 lieues sous les mers », « Le Titanic », « Ecrivain », « Jean Gabin ».

3.2.3 Le choix d'identificateurs cohérents

Identificateurs

Un *identificateur* est le nom choisi par un programmeur pour identifier un élément dans un programme. Ce nom sert alors à décrire ou manipuler l'élément concerné. Il peut s'agir d'un nom de classe, d'un nom d'objet, d'un état...

Un identificateur peut être constitué de lettres (majuscules ou minuscules, sans accent), de chiffres, et des caractères « *souligné* » ou « *dollar* » (« _ » ou « $ »). Il ne peut commencer par un chiffre.

Le langage Pascal ne faisant pas les distinctions entre majuscules et minuscules, les identificateurs « *Couleur* », « *couleur* » et « *COULEUR* » font référence à la même chose.

Du choix des identificateurs

Notre conseil est que vous vous habituiez à choisir des identificateurs qui soient en harmonie de la théorie de l'objet telle que nous vous l'avons présentée. Ainsi :

- Pour nommer une classe, choisissez un nom assez général qui évoque la nature de la classe (exemple « *Fenêtre* »).
- Pour nommer un objet, choisissez un nom précis qui évoque ce qu'est cet objet (exemple « *FenêtreDeSaisieDesClients* »).
- Pour nommer un état dans un objet, choisissez un nom ou adjectif qui évoque cet état (exemple « *Couleur* », « *Largeur* », « *Visible* », *Actif* »).
- Pour nommer un comportement, choisissez de préférence un verbe (exemple « *Agrandir* », « *Deplacer* », « *Enregistrer* »).

Remarque

Une habitude de programmation veut sous *Delphi* que l'on préfixe les noms de classes de la lette « *T* ». Par exemple « *TFenetre* », « *TFichier* »…

Une autre habitude de programmation veut sous *Delphi* que l'on nomme ses objets en mettant un préfixe qui rappelle la *classe* à laquelle appartient l'*objet*. Cette abréviation est souvent constituée de la première lettre de la classe suivie des consonnes du nom de la classe (En éliminant les consonnes en double). Par exemple « *FntrClient* » pour une fenêtre client, « *FchrFacture* » pour un fichier de factures.

4. L'ECRITURE ET L'UTILISATION D'UNE CLASSE

4.1 LA DECLARATION D'UNE CLASSE

Une *classe* se déclare dans la section « `type` » d'un programme ou d'une unité *Pascal* grâce au mot clef « `class` ». Cette déclaration ressemble à la déclaration d'une *structure Pascal* (*record*), mais peut comporter en plus la déclaration des *fonctions* et des *procédures* pouvant manipuler les objets de cette classe.

Cette déclaration, utilisant le mot « `class` », dans sa forme la plus simple est la suivante :

```
Type
  TnomClasse = class
    // Déclaration des données devant être stockés dans chaque objet
    // Declaration des procédure et des fonctions pouvant manipuler
       // les objets
    end;
```

Prenons par exemple un extrait du source d'une unité gérant des fiches d'adresse :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher;
    procedure Saisir;
    end;

implementation

// Code non écrit

end.
```

Remarque

Les types simples de données les plus fréquemment utilisés par le programmeur *Delphi* sont les suivants :

- *Integer* : valeur entière sur 32 bits.
- *Double* : valeur numérique en virgule flottante double précision.
- *String*[21] : chaîne de caractère de taille quelconque (jusqu'à 2 Go).
- *Char* : Caractère.
- *Boolean* : valeur logique (true ou false).

[21] La déclaration « `string` » a légèrement changé de valeur entre *Delphi* et *Borland Pascal*. La déclaration « `string` » de *Delphi* correspond maintenant à une chaîne de caractères de taille quelconque pouvant aller jusqu'à 2 gigaoctets. *Delphi* en gère automatiquement l'allocation mémoire. Les anciennes chaînes de caractères de *Borland Pascal* sont toujours disponibles avec la déclaration « *ShortString* » ou en modifiant les options de compilation.

La déclaration précédente montre que chaque objet de la classe « *TAdresse* » devra stocker 4 variables de type chaîne de caractères (*string*), « *Nom* », « *Rue* », « *CP* » et « *Ville* » — Ces 4 variables sont aussi des *propriétés* qui déterminent l'*état* d'une adresse[22].

Cette déclaration montre aussi que deux procédures sans paramètres, « *Afficher* » et « *Saisir* » peuvent être utilisées pour manipuler les objets de cette classe.

Définition

Les procédures et fonctions déclarées à l'intérieur d'une classe s'appellent des *méthodes*.

Les méthodes sont utilisées pour modéliser le *comportement* d'un objet.

Remarque

Pour être conforme au modèle objet, l'ensemble des méthodes doit représenter tous les *comportements* de cet objet.

Ainsi un bien meilleur exemple comporterait d'autres méthodes :

```
TAdresse = class
  Nom   : string;
  Rue   : string;
  CP    : string;
  Ville : string;
  procedure Afficher;
  procedure Saisir;
  procedure Demenager (NouvRue, NouvCP, NouvVille : string);
  procedure ChangerNom (NouvNom : string);
  procedure ImprimerEnveloppe;
  procedure Enregistrer (Fichier : Tstream);
  procedure Lire (Fichier : Tstream);
  end;
```

4.2 L'ECRITURE DES METHODES

Le code associé à une *méthode* peut s'écrire au même endroit que celui où on aurait écrit celui d'une procédure ou une fonction *Pascal*. Habituellement celui-ci s'écrit dans la section « *implementation* » d'une unité (celle où a été déclarée la classe)[23].

Comme des *classes* différentes peuvent comporter des *méthodes* de même nom, l'écriture de la méthode doit obligatoirement faire référence au nom de la classe pour laquelle on l'écrit[24]. Pour ceci le nom de la procédure ou de la fonction doit être préfixé du nom de la classe et de l'opérateur *point*, « . ».

[22] Ces variables qui sont stockées dans des objets s'appellent aussi des *variables d'instance*.

[23] La *classe* et les *méthodes* peuvent aussi s'écrire dans la partie déclaration d'un programme Pascal.

[24] De plus l'identificateur choisi pour nommer la méthode peut avoir aussi été employé à d'autres fins à l'extérieur de la classe.

Lorsqu'on écrit une méthode, on doit avoir conscience que celle-ci est destinée à être appliquée à un objet de la classe. Donc tout ce qui a été déclaré dans la classe peut être manipulé dans la méthode. Par exemple si nous faisons référence à l'identificateur « *CP* », il s'agira du « *CP* » de l'objet auquel nous appliquerons la méthode.

Examinons la deuxième partie du source de l'unité de gestion d'adresses :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher;
    procedure Saisir;
    end;

implementation

procedure TAdresse.Afficher;
begin
  WriteLn ('Nom   : ', Nom);
  WriteLn ('Rue   : ', Rue);
  WriteLn ('CP    : ', CP);
  WriteLn ('Ville : ', Ville);
end;

procedure TAdresse.Saisir;
begin
  Write ('Nom   : '); ReadLn (Nom);
  Write ('Rue   : '); ReadLn (Rue);
  Write ('CP    : '); ReadLn (CP);
  Write ('Ville : '); ReadLn (Ville);
end;

end.
```

Dans la partie *implementation*, « `TAdresse.Afficher` » fait référence à la méthode « `Afficher` » qui a été définie dans la classe « `TAdresse` » et « `TAdresse.Saisir` » à la méthode saisir qui a été définie dans cette classe.

Les variables manipulées, « `Nom` », « `Rue` », « `CP` » et « `Ville` » correspondent aux propriétés des objets auxquels ces méthodes seront appliquées car ces identificateurs ont été décrits dans la déclaration de la classe.

Remarque

> Lors de l'écriture d'une méthode, ne cherchez pas la ligne de code qui appelle cette méthode. Nous avons vu beaucoup de programmeurs issus de la programmation structurée et débutant en *POO* le faire ; ce n'est pas un raisonnement objet.
>
> Pensez qu'une méthode représente un des comportements de l'objet. Nous avons donc un objet donné, dans un état donné. Cet objet possède des propriétés et des méthodes. Ainsi dans la méthode « `Afficher` » on fait ce qu'il faut pour que cet objet s'affiche.

Remarque

Avant d'écrire quelque code que ce soit, sachez que Delphi offre des outils facilitant l'écriture de ce code.

Lorsque la déclaration de la classe est terminée, mettez le curseur à l'intérieur de celle-ci (entre le mot « `class` » et le mot « `end` ») et appuyez sur les touches ⇧ Shift Ctrl et '*C*'. Delphi crée automatiquement le squelette des méthodes manquantes dans la partie implémentation.

D'autre part lorsque vous êtes sur une déclaration de méthode (dans une classe) et que vous appuyez sur les touches ⇧ Shift Ctrl ↑ ou ⇧ Shift Ctrl ↓, le curseur se déplace automatiquement vers le code de cette méthode ou remonte du code de la méthode vers la déclaration.

4.3 LES OBJETS DE LA CLASSE

4.3.1 L'instanciation

Nous vous rappelons qu'une *instance* est un objet et que l'*instanciation* est la création d'un objet.

Pour pouvoir *instancier* un objet, nous allons commencer par déclarer une variable qui gardera la trace de cet objet. Pour ceci, revenons dans le programme principal et créons une section « `var` » dans laquelle nous pourrons créer une ou plusieurs variables dont le type correspondra à la classe que nous avons déclarée dans l'unité « `Adresses` ».

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TAdresse;

begin

  ReadLn;
end.
```

Attention

Nous avons maintenant deux variables permettant de gérer des adresses, mais cela ne veut pas dire que nous avons des objets. En effet ceux-ci n'ont pas encore été instanciés.

L'instanciation correspond à une instruction et sera écrite là où on écrit des instructions. En effet la création d'un objet représente une action et des instructions.

En fait, dans le *Pascal Objet* de *Delphi*, un objet est alloué dynamiquement en mémoire, et les variables que nous avons déclarées ne sont que des pointeurs qui permettent de garder la trace de ces objets. Si un autre moyen de garder la trace de ces objets existe, il n'est pas nécessaire de déclarer ces variables.

Toutes les classes possèdent au moins une méthode « `Create` » destinée à créer un objet. Ce type de méthode dont le but est la création d'un objet s'appelle un *constructeur*, elle se charge de l'initialisation correcte de l'objet.

Pour créer un objet, il faut utiliser un constructeur et l'appliquer à la classe avec l'opérateur *point*, « . », par exemple l'instruction « `TAdresse.Create` » crée une nouvelle adresse. Le constructeur retourne l'objet créé.

Notre exemple devient alors :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TAdresse;

begin
  QuelquUn := TAdresse.Create;
  UnAutre  := TAdresse.Create;

  ReadLn;
end.
```

4.3.2 La destruction

Qui dit allocation mémoire, dit libération de la mémoire. De la même manière qui dit *construction* dit aussi destruction.

Le programmeur qui a créé un objet est aussi responsable du fait que cet objet soit ultérieurement détruit. Non seulement la mémoire sera libérée, mais lors de la destruction d'un objet certains traitements particuliers pourront être entrepris afin de libérer d'autres ressources, nettoyer le système, etc…

Tous les objets possèdent une méthode particulière, « `Destroy` » destinée à faire ce travail de libération des ressources : c'est un *destructeur*.

Tous les objets possèdent aussi une méthode « `Free` » qui se charge de vérifier si l'objet existe, puis si tel est le cas appelle le *destructeur* « `Destroy` ».

Il suffit d'appliquer une de ces méthodes à un objet par l'intermédiaire de l'opérateur *point*, « . » pour détruire l'objet. Ainsi les deux manières de détruire un objet sont les suivantes :

```
Objet.Free;    // Manière sure de détruire un objet
ou
Objet.Destroy; // Destruction d'un objet existant
```

Notre exemple devient alors :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TAdresse;

begin
  QuelquUn := TAdresse.Create;
  UnAutre  := TAdresse.Create;

  QuelquUn.Free;
  UnAutre .Free;
  ReadLn;
end.
```

4.3.3 La manipulation d'un objet

Entre sa construction et sa destruction le contenu d'un objet peut être manipulé par l'opérateur *point*, « . ».

Une donnée stockée dans un objet peut être manipulée comme n'importe quelle variable avec la syntaxe « `Objet.Propriété` ». Une méthode appartenant à un objet peut être appliquée à cet objet avec la syntaxe « `Objet.Methode` » ou « `Objet.Methode (Parametres)` ».

Le programme peut alors devenir :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TAdresse;

begin
  QuelquUn := TAdresse.Create;
  UnAutre  := TAdresse.Create;

  QuelquUn.Nom   := 'Durand';
  QuelquUn.Rue   := '17, rue Collette';
  QuelquUn.CP    := '75017';
  QuelquUn.Ville := 'Paris';
  QuelquUn.Afficher;

  UnAutre.Saisir;
  UnAutre.Afficher;

  WriteLn ('Vous venez de manipuler l''adresse de ', UnAutre.Nom);
  WriteLn ('Cette personne habite ', UnAutre.Ville);

  QuelquUn.Free;
  UnAutre .Free;
  ReadLn;
end.
```

Pratique

Réalisez l'exemple précédent.

Créez une classe simple destinée à gérer des factures. Ces factures se caractérisent par un libellé (texte), un montant hors taxe (double précision) et un taux de TVA (double), avec une méthode de saisie et une méthode d'affichage, et testez la dans le programme principal.

4.3.4 Le regroupement des instructions

Lorsque plusieurs instructions font références aux propriétés ou aux méthodes d'un même objet, on peut utiliser l'opérateur « `with` ». Celui-ci fonctionne de la même manière que le « `with` » des *records Pascal*. Ainsi le programme principal précédent peut devenir :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TAdresse;

begin
  QuelquUn := TAdresse.Create;
  UnAutre  := TAdresse.Create;

  With QuelquUn do
    begin
    Nom   := 'Durand';
    Rue   := '17, rue Collette';
    CP    := '75017';
    Ville := 'Paris';
    Afficher;
    end;


  With UnAutre do
    begin
    Saisir;
    Afficher;

    WriteLn ('Vous venez de manipuler l''adresse de ',Nom);
    WriteLn ('Cette personne habite ',Ville);
    end;

  QuelquUn.Free;
  UnAutre .Free;
  ReadLn;
end.
```

Dans certains cas on peut même se passer de la déclaration de la variable servant à garder la trace de l'objet :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

begin
  With TAdresse.Create do
    begin
    Nom   := 'Durand';
    Rue   := '17, rue Collette';
    CP    := '75017';
    Ville := 'Paris';
    Afficher;
    Free;
    end;

  ReadLn;
end.
```

Pratique

Revoyez votre précédent exercice avec des « `with` ».

4.4 DES INFORMATIONS COMPLEMENTAIRES POUR L'ECRITURE DE CLASSES

4.4.1 Des méthodes avec des paramètres

Tout comme des *procédures* et des *fonctions* Pascal peuvent en avoir, des *méthodes* peuvent gérer des *paramètres*[25].

Exemple

Nous allons ajouter à la classe *TAdresse* une méthode « `Initialiser` » permettant d'initialiser un objet en une seule instruction :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Initialiser (UnNom, UneRue, UnCP, UneVille : string);
end;

implementation

procedure TAdresse.Initialiser (UnNom, UneRue, UnCP,
  UneVille : string);
begin
  Nom   := UnNom;
  Rue   := UneRue;
  CP    := UnCp;
  Ville := UneVille
end;

end.
```

Les paramètres sont ensuite passé à la méthode comme ils le seraient à n'importe quelle procédure *Pascal* :

```
program GereAdresses;

uses
  Forms,
  Adresses in 'Adresses.pas';

Var
  QuelquUn : TAdresse;

begin
  QuelquUn := TAdresse.Create;
  QuelquUn.Initialiser ('Durand', '17, rue Collette', '75017',
'Paris');
  QuelquUn.Afficher;
  QuelquUn.Free;
ReadLn;
end.
```

Pratique

Tester la méthode « `Initialiser` », puis créez les méthodes « `Demenager` » et « `ChangerDeNom` ».

[25] Pour de plus amples explications sur l'utilisation des paramètres, nous vous renvoyons à la formation *Pascal*.

4.4.2 La valeur de retour d'une fonction

En *Pascal* traditionnel, la valeur de retour d'une fonction est assignée en affectant une valeur au nom de la fonction. Par exemple :

```
function Pourcent (Valeur, Base : double) : double;
begin
  Pourcent := Valeur * 100 / Base;
end;
```

Avec le *Pascal* de *Delphi*, une variable implicite, « `Result` », est déclarée automatiquement par le compilateur afin de pouvoir manipuler la valeur de retour. Par exemple :

```
function Pourcent (Valeur, Base : double) : double;
begin
  Result := Valeur * 100 / Base;
end;
```

Ceci offre les avantages suivants :

- Cette variable offre un nom générique pratique disponible dans toues les fonctions.
- On peut manipuler cette variable comme toute autre variable sans rencontrer de problème de réentrance :

```
Result := Result + 1;     // Augmente la valeur de retour d'une unité

Pourcent := Pourcent + 1; // Appel récurent à la fonction
                          // puis augmentation du résultat d'une unité
```

4.4.3 Une méthode peut en utiliser une autre

Avec les déclarations que nous avons vues, tout ce qui est déclaré dans une classe peut être manipulé dans le code d'une méthode. Une méthode peut donc utiliser une autre méthode.

Exemple

```
unit Factures;

interface

type
  TFacture = class
    Libelle : string;
    HT      : double;
    TauxTVA : double;
    function TVA : double;
    function TTC : double;
    procedure Afficher;
    end;
```

```
implementation

function TFacture.TVA : double;
begin
  Result := HT * TauxTVA / 100;
end;

function TFacture.TTC : double;
begin
  Result := HT + TVA;
end;

procedure TFacture.Afficher;
begin
  WriteLn ('Libelle   : ', Libelle);
  WriteLn ('HT        : ', HT:10:2, ' F');
  WriteLn ('Taux TVA  : ', TauxTVA:5:2, '%');
  WriteLn ('TVA       : ', TVA:10:2, ' F');
  WriteLn ('TTC       : ', TTC:10:2, ' F');
end;

end.
```

Pratique

Réalisez et testez l'exemple précédent.

Considérons que nous travaillons pour une société qui fabrique des plaques en fontes. Dans un premier temps, nous allons concevoir une classe, « `TPlaqueCirculaire` », pour gérer des plaques en fonte circulaire (Les variables d'instance sont : le nom de référence, le diamètre en mm, l'épaisseur en mm et le poids en kilogramme par litre). Concevez les méthodes pour calculer la surface de la plaque[26], son volume[27] et son poids[28], concevez une autre méthode pour afficher les caractéristiques d'une plaque, et une autre pour les saisir — Testez toutes ces méthodes.

4.4.4 L'objet courant

Les méthodes que nous avons écrites sont destinées à être appliquées à des objets par l'intermédiaire de l'opérateur *point*, « . ». Implicitement dans une méthode nous manipulons l'objet auquel cette méthode est appliquée, cependant dans certaines circonstances, on peut avoir besoin de connaître explicitement l'objet auquel la méthode est appliquée.

Toutes les méthodes pouvant être appliquées à un objet possèdent un paramètre implicite, « `self` » qui identifie l'objet auquel est appliquée la méthode.

[26] Surface = Π x D^2 / 4

[27] Volume = Surface x Epaisseur

[28] Poids = Volume x Densité — Attention à la concordance des unités, nous avons une densité en kg par litre, et un litre = 1 dm^3.

Exemple

Le code ci-dessous ne peut marcher car le nom des paramètres masque le nom des variables d'instance

```
unit Factures;

interface

type
  TFacture = class
    Libelle : string;
    HT      : double;
    TauxTVA : double;
    Procedure Intialiser (Libelle : string; HT, TauxTVA : double);
    end;

implementation

procedure TFacture.Intialiser (Libelle : string;
HT, TauxTVA : double);
begin
  Libelle := Libelle;   // ne marche pas
  HT      := HT;        // ne marche pas
  TauxTVA := TauxTVA;   // ne marche pas
end;

end.
```

Si on tient à écrire la méthode de cette manière, il faut passer par le paramètre « `self` » :

```
procedure TFacture.Intialiser (Libelle : string;
HT, TauxTVA : double);
begin
  Self.Libelle := Libelle;
  Self.HT      := HT;
  Self.TauxTVA := TauxTVA;
end;
```

Pratique

Vérifier que le premier exemple ne produit pas le résultat souhaité.

Corrigez-le avec « `self` » et vérifiez qu'il produit le résultat souhaité.

4.4.5 La gestion des boucles

Le *Pascal* de *Delphi* offre deux nouvelles instructions qui permettent de gérer la sortie et la reprise des boucles (boucle « `for` », « `while` », « `repeat`… `until` »).

« `break` » : le reste de la boucle est ignoré et on sort de la boucle. Par exemple :

```
function LireTexte : string;
  var
    Mot : string;
  Begin
  Result := '';
  While true do
    Begin
    ReadLn (Mot);
    if Mot = 'STOP' then
      break;                // Sortie de la boucle
                            // et donc de la fonction
    Result := Result + ' ' + Mot;
    end;
  end;
```

« `continue` » : le reste de la boucle est ignoré et on reprend la boucle. Par exemple :

```
procedure ListerNombre;
  var
    i : integer;
  begin
  for i := 1 to 100 do
    begin
    if i mod 10 = 0 then
      continue;             // les chiffres muiltiples de 10
                            // sont ignorés
    // traitement des autres nombres
    end;
  end;
```

4.4.6 Un objet peut faire référence à un autre objet

Un objet peut faire référence à un autre objet. Par exemple :

```
unit Factures;

interface

uses Plaques;

type
  TFacture = class
    Produit : TPlaqueCirculaire;
    HT      : double;
    TauxTVA : double;
    procedure Afficher;
    end;

implementation

procedure TFacture.Afficher;
  begin
  WriteLn ('Article facture');
  Produit.Afficher;
  WriteLn ('HT ',       HT:10:2);
  WriteLn ('Taux TVA ', TauxTVA:5:2);
  end;

end.
```

Lors de l'utilisation de ces objets, il faudra s'assurer que l'objet « `Produit` » soit correctement initialisé, et il faudra utiliser l'opérateur *point*, « . » autant de fois que nécessaires pour accéder aux caractéristiques de l'objet référencé :

```
program Gestion;

uses
  Forms,
  Factures in 'Factures.pas',
  Plaques in 'Plaques.pas';

Var
  Article : TPlaqueCirculaire;
  Facture : TFacture;

begin
  Article := TPlaqueCirculaire.Create;
  Facture := TFacture.Create;

  Article.Initialiser (' Fonte 1,20m x 20mm', 1200, 20);
  Facture.Produit := Article;
  Facture.HT      := 1000;
  Facture.TauxTVA := 20.6;

  WriteLn (
    'Nom du produit facturé ',
     Facture.Produit.Nom
     );

  Facture.Free;
  Article.Free;
  ReadLn;
end.
```

Remarque

Nous vous rappelons que sous *Delphi*, les variables de type objet sont en fait des pointeurs sur des blocs de mémoire contenant les caractéristiques des objets (contenant les variables d'instance). Donc, après l'instruction « `Facture.Produit := Article;` » les variables « `Facture.Produit` » et « `Article` » font référence au même objet.

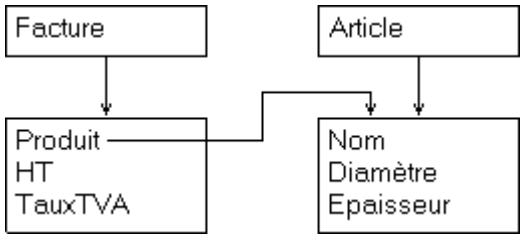


Figure 8. Les variables sont des références vers des objets

Pratique

Après avoir testé le code précédent, ajoutez une ou plusieurs instructions qui modifient le contenu de l'objet « `Article` » après l'affectation « `Facture.Produit := Article;` », puis affichez la facture. Que se passe-t-il ?

4.4.7 Tester l'existence d'un objet

Les variables de type objet étant des références vers des objets, nous ne sommes jamais sûrs qu'une variable fasse référence à un objet existant.

Remarque

Comme les variables de type objet sont des pointeurs on peut leur affecter la valeur « `nil` » pour signaler que cette variable ne fait référence à aucun objet. Par exemple :

```
Facture.Free;     // destruction réelle de l'objet
Facture := nil;   // Facture ne fait référence à aucun objet
```

Dans le *Pascal* de *Delphi* on ne peut tester directement la valeur d'un pointeur par rapport à la valeur « `nil` », il faut passer par la fonction « `Assigned(Pointeur)` » qui renvoie « `true` » ou « `false` » suivant que le pointeur est ou non affecté.

Exemple

L'instruction « `Produit.Afficher;` » dans l'exemple précédent est dangereuse car à ce stade nous n'avons aucune certitude quant au fait que « `Produit` » fasse référence à un article existant.

```
procedure TFacture.Afficher;
  begin
  Produit.Afficher;                  // instruction dangereuse
  end;
```

Pour sécuriser le code, il faudrait écrire la méthode comme suit :

```
procedure TFacture.Afficher;
  begin
  If Assigned (Produit) then Produit.Afficher
  Else WriteLn ('Aucun produit n''est lie à cette facture');
  end;
```

5. L'ECRITURE DE CONSTRUCTEURS ET DE DESTRUCTEURS

5.1 DEFINITIONS

Les *constructeurs* et les *destructeurs* sont des méthodes vitales dans la gestion du cycle de vie d'un objet.

Constructeur

Un *constructeur* est une méthode responsable de l'initialisation complète et totale d'un objet.

Entre autres choses, un constructeur alloue la mémoire nécessaire à l'objet.

Remarque

Une classe peut posséder plusieurs *constructeurs* ; il s'agira d'autant de manières différentes d'initialiser un objet.

Un constructeur peut ou non comporter des paramètres.

Toutes les classes possèdent un *constructeur* par défaut : « `Create` » qui se contente d'allouer la mémoire et d'initialiser celle-ci à 0.

Destructeur

Un *destructeur* est une méthode responsable de tout le travail de « nettoyage » nécessaire à la libération d'un objet.

Entre autres choses, un destructeur libère la mémoire nécessaire à l'objet.

Remarque

Habituellement les objets ne possèdent qu'un *destructeur* (bien qu'il soit possible d'en écrire plusieurs).

Tous les objets possèdent un destructeur par défaut : « `Destroy` ».

Remarque importante

Un programmeur qui construit un objet est implicitement responsable de la destruction de cet objet.

Dans les exemples que nous avons écrits jusqu'à présent, l'omission de la destruction ne portait pas à conséquence, par contre si vous omettez de détruire les objets que vous avez construit dans une application *Windows*, vous risquez de vite épuiser les ressources système.

Voici quelques principes simples qui président à l'écriture des constructeurs, des destructeurs et des classes :

- En écrivant un *constructeur*, examinez toutes les variables déclarées dans la classe et décidez si oui ou non elles doivent être initialisées.
- En écrivant un *destructeur*, examinez tout ce qui a été fait dans les constructeurs et décidez de ce que vous devez faire dans le destructeur.
- En ajoutant de nouvelles déclarations de variables dans une classe, examinez tous les constructeurs et tous les destructeurs associés à cette classe et décidez de que vous devez faire.

Par exemple :

- Si un objet ouvre un fichier dans son constructeur, il faudra penser à le fermer dans le destructeur.
- Si un objet alloue de la mémoire pour un travail particulier, il faudra penser à la libérer dans le destructeur.
- Si un objet instancie un autre objet dans le constructeur, il faudra probablement le détruire dans le destructeur (dans ce cas, le premier objet est considéré comme le propriétaire de l'objet contenu).

5.2 L'ECRITURE D'UN CONSTRUCTEUR

Un *constructeur* s'écrit comme n'importe quelle autre méthode en utilisant le mot clef « `constructor` » au lieu du mot clef « `procedure` » ou « `function` ».

Exemple

```
unit Factures;

interface

type
  TFacture = class
    Libelle : string;
    HT      : double;
    TauxTVA : double;
    constructor Create (UnLibelle : string; UnHT, UnTauxTva : double);
    constructor CreateEmpty;
    end;

implementation

constructor TFacture.Create (UnLibelle : string;
UnHT, UnTauxTva : double);
  begin
  Libelle := UnLibelle;
  HT      := UnHT;
  TauxTva := UnTauxTVA;
  end;

constructor TFacture.CreateEmpty;
  begin
  Libelle := '';
  HT      := 0;
  TauxTva := 20.6;
  end;

end.
```

Remarque

Bien que nous ayons attiré votre attention sur le fait qu'un constructeur alloue la mémoire, vous n'avez aucune précaution à prendre en tant que programmeur, cela est réalisé automatiquement par le compilateur quand il détecte que nous avons affaire à un constructeur.

Dans cet exemple, le constructeur par défaut, « `Create` » a été remplacé par un autre constructeur « `Create` » possédant 3 paramètres.

Lorsqu'un constructeur possède des paramètres, il faut que ceux-ci soient renseignés lors de l'instanciation. Les deux manières de créer une facture sont maintenant les suivantes :

```
program Gestion;

uses
  Forms,
  Factures in 'Factures.pas',

Var
  F1 : TFacture;
  F2 : TFacture;

begin
  F1 := TFacture.Create ('Plaque', 1000, 20.6);
  F2 := CreateEmpty;

  // etc...

  F1.Free;
  F2.Free;
  ReadLn;
end.
```

Pratique

Revoyez les classes « `TAdresse` » et « `TPlaqueCirculaire` » afin de leur créer un constructeur « `Create` » qui permette de passer tous les paramètres nécessaires à l'initialisation des variables d'instances. Testez-les.

5.3 L'ECRITURE D'UN DESTRUCTEUR

Un *destructeur* s'écrit comme n'importe quelle autre méthode en utilisant le mot clef « `destructor` » au lieu du mot clef « `procedure` », « `function` » ou « `constructor` ».

Exemple

Par malheur le remplacement du *destructeur* par défaut, « `Destroy` », par un *destructeur* de notre cru fait référence à une notion de *Pascal Objet* que nous n'avons pas encore vue. Aussi dans les exemples qui suivent, nous créerons un autre *destructeur*, « `Suppress` ». D'un point de vue logique, vous n'avez alors plus le droit de détruire des objets en utilisant le destructeur « `Destroy` » ou la méthode « `Free` » (Bien que le compilateur nous y autoriserait).

Considérons maintenant qu'un objet de la classe facture contienne un objet de la classe d'adresse. Et considérons que la facture est la propriétaire de l'adresse. Lorsque l'on crée une facture on crée une adresse, lorsqu'on détruit la facture on détruit l'adresse :

```
unit Factures;

interface

uses Adresses;

type
  TFacture = class
    Client  : TAdresse;
    HT      : double;
    TauxTVA : double;
    constructor Create;
    destructor  Suppress;
    end;

implementation

constructor TFacture.Create;
  begin
  Client  := TAdresse.Create;
  HT      := 0;
  TauxTVA := 20.6;
  end;

destructor TFacture.Suppress;
  begin
  Client.Free;
  end;

end.
```

L'utilisation de ces classes peut alors être :

```
program Gestion;

uses
  Forms,
  Adresses in 'Adresses.pas',
  Factures in 'Factures.pas';

Var
  Fact : TFacture;

begin
  Fact := TFacture.Create;
  Fact.Client.Nom   := 'Durand';
  Fact.Client.Ville := 'Paris;
  Fact.HT           := 1000;
  // etc...

  Fact.Suppress;
  ReadLn;
end.
```

Remarque

Dans l'exemple de la section *4.4.6.* Un objet peut faire référence à un autre objet, page 36, nous disions qu'un objet faisait *référence* à un autre objet. En effet vu la manière dont était organisé le code, les objets pouvaient être considérés comme indépendants l'un de l'autre.

Dans le code que nous venons d'écrire il existe une relation étroite entre la facture et l'adresse. Nous disons ici que la facture contient l'adresse ou qu'elle en est le propriétaire. Cette relation entre objets s'appelle une relation de *contenance*.

Dans un cas comme dans le code précédent, lorsqu'un objet manipule un autre objet dont il n'est pas le propriétaire nous parlons d'une relation d'*utilisation*.

Pratique

Après avoir réalisé l'exemple précédent, ajoutez un destructeur « Suppress » dans la classe d'adresse qui se contente d'afficher le message « Destruction de l'adresse de … » (précisant le nom de la personne dont on détruit l'adresse, et vérifiez que la destruction de la facture affiche ce message.

Reprenez la classe de facture afin d'inclure dans chacun de ses objets un produit (une plaque en fonte). Construisez et détruisez la facture avec les mêmes principes.

5.4 COMPLEMENT SUR L'ECRITURE DE METHODES

5.4.1 Des méthodes différentes peuvent porter le même nom

Signalons ici une faculté du *Pascal Objet* de *Delphi 4* qui permet d'assigner le même nom à des méthodes différentes moyennant qu'elles possèdent des paramètres différents (*Rédéfinition* à la mode du C++). Pour que ceci soit possible, les méthodes doivent être accompagnées du mot clef « `overload` ».

Ceci peut nous permettre d'écrire différents *constructeurs* portant tous le même nom. Par exemple :

```
type
  TFacture = class
    Libelle : string;
    HT      : double;
    TauxTVA : double;
    constructor Create (UnLibelle : string;
                UnHT, UnTauxTva : double); overload;
    constructor Create; overload;
    end;
```

5.4.2 Les valeurs par défaut

Depuis *Delphi 4*, les procédures, les fonctions et toutes les méthodes peuvent comporter des paramètres ayant des valeurs par défaut.

Il s'agit obligatoirement des derniers paramètres.

La valeur par défaut est précisée avec un signe *égale*, « = », après la déclaration du paramètre.

Exemple

```
unit Factures;

interface

type
  TFacture = class
    Libelle : string;
    HT      : double;
    TauxTVA : double;
    constructor Create (UnLibelle : string;
                        UnHT : double;
                        UnTauxTva : double = 20.6);
    end;

implementation

constructor TFacture.Create (UnLibelle : string;
UnHT:double; UnTauxTva : double);
  begin
  Libelle := UnLibelle;
  HT      := UnHT;
  TauxTva := UnTauxTVA;
  end;

end.
```

Une facture peut maintenant être instanciée de deux manières différentes « `TFacture.Create ('Fonte', 1000, 18.6)` » ou « `TFacture.Create ('Fonte', 1000)` ». Dans ce deuxième cas, le compilateur passe à la méthode la valeur 20.6 pour le troisième paramètre.

6. L'ENCAPSULATION

6.1 LA PRESENTATION DE L'ENCAPSULATION

Définition

L'*encapsulation* consiste à séparer la conception d'une classe de la manière dont on l'utilise ou dont on utilise ses objets.

Avec la notion d'*encapsulation* on ne permet pas à l'utilisateur d'une classe de manipuler des données ou du code qui ont concerne la manière dont la classe est réalisée.

Autrement dit, la classe définit une interface qui permet de dialoguer avec la classe ou ses objets, alors que la manière dont sont traités les problèmes est masquée à l'utilisateur.

Remarque

Le terme d'encapsulation vient du mot « capsule »

En fait en *Pascal* la manière dont sont organisées les unités réalise déjà une certaine forme d'*encapsulation*. La section « `interface` » monte tout ce qu'on peut utiliser dans une unité alors que la partie « `implementation` » ne concerne que le concepteur de l'unité.

Cette forme d'*encapsulation* est appréciable mais, comme le montreront les exemples que nous verrons dans ce chapitre, est insuffisante pour le programmeur orienté objet.

6.2 LES ATTRIBUTS D'ENCAPSULATION

Dans une classe on peut ouvrir des sections de différente portée grâce aux mots clefs suivants :

- « `public` » : toutes les déclarations dans la classe qui suivent le mot « `public` » sont publiques, c'est à dire qu'elles sont accessibles à n'importe quel source qui peut voir la déclaration de la classe.
- « `private` » : toutes les déclarations dans la classe qui suivent le mot « `private` » ne sont visibles que du source en cours.
- « `protected` » : cet attribut sera décrit ultérieurement.
- « `published` » : si la classe que nous décrivons est un composant qui peut être installé dans la palette de composant, non seulement les déclarations qui suivent seront publiques mais en plus les propriétés seront visibles dans l'inspecteur d'objet.

Par défaut les déclarations faites dans une classe sont publiques, c'est la raison pour laquelle nous n'avons jamais eu à nous préoccuper de la visibilité de nos déclarations.

Remarque

On peut ouvrir plusieurs section de type « `public` », « `private` », « `protected` » ou « `published` » dans une classe.

Exemple

Considérons une classe de fenêtres. Chaque fenêtre possède une position, une taille et une couleur de fond. Chaque fenêtre possède aussi une méthode permettant de la redessiner. On pourrait aussi ajouter des méthodes permettant de déplacer la fenêtre, la redimensionner, en changer la couleur. Dans note exemple, ne prenons en considération que la gestion du code de couleur ; si nous en restons là avec le code ci-dessous, la classe va montrer quelques faiblesses :

```
unit Fenetres;

interface

type
  TFenetre = class
    //...
    Couleur    : integer;
    //...
    procedure Redessiner;
    procedure ChangerCouleur (c : integer);
    //...
    end;

implementation

procedure TFenetre.Redessiner;
begin
// code de dessin
end;

procedure TFenetre.ChangerCouleur (c : integer);
begin
  Couleur := c;
  Redessiner;
end;

end.
```

Illustrons maintenant le problème que l'on peut rencontrer à l'utilisation de cette classe :

```
program Dessiner;

uses
  Forms,
  Fenetres in 'Fenetres.pas',

Var
  Fen : TFenetre;

begin
  Fen := TFenetre.Create;

  Fen.ChangerCouleur (ClGreen);  // parfait
                                 // la fenêtre se dessine en vert

  //...
  Fen.Coul := ClRed;             // problème!
                                 // Le contenu de l'objet n'est plus
                                 // cohérent avec son apparence.

  //...

  Fen.Free;
  ReadLn;
end.
```

Maintenant corrigeons la classe avec la notion d'encapsulation. Nous allons déclarer la couleur comme étant privée. Le compilateur refuse alors de compiler « `Fen.Coul := ClRed ;` » et remonte une erreur. L'utilisateur de la classe est donc obligé de passer par la méthode « `ChangerCouleur` » s'il souhaite modifier la couleur de la fenêtre ; ce faisant la fenêtre sera aussi redessinée à l'écran avec la bonne couleur.

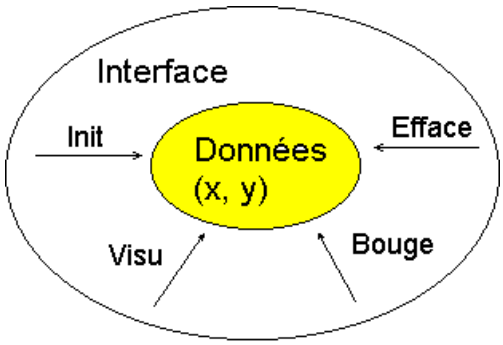


Figure 9. Schéma illustrant l'encapsulation

Le schéma ci-dessus illustre l'*encapsulation* des données d'un objet graphique situé à des coordonnées x, y données.

La couche extérieure, l'*interface*, permet de communiquer avec l'objet. Le cœur est inaccessible de l'utilisateur. Mais que se passe-t-il si l'utilisateur de la classe a besoin d'interroger l'objet pour lui demander quelle est sa couleur ? Il ne peut plus consulter la variable « Fen.Coul » car le compilateur remonterait une erreur. C'est alors la responsabilité du concepteur de la classe de fournir un moyen d'accéder à cette valeur ; par exemple par l'intermédiaire d'une fonction qui renvoie la valeur de la couleur.

Le code de la classe devient alors :

```
unit Fenetres;

interface

type
  TFenetre = class
    private
      //...
      Couleur    : integer;
    public
      procedure Redessiner;
      procedure ChangerCouleur (c : integer);
      function LireCouleur : integer;
      //...
    end;
```

```
implementation

procedure TFenetre.Redessiner;
begin
// code de dessin
end;

procedure TFenetre.ChangerCouleur (c : integer);
begin
  Couleur := c;
  Redessiner;
end;

function TFenetre.LireCouleur : integer;
begin
  Result := Coul;
end;
end.
```

L'utilisateur s'il veut changer la couleur doit maintenant faire « `Fen.ChangerCouleur (ClRed) ;` » et s'il a besoin de consulter la couleur il doit faire quelque chose du genre « `c := Fen.LireCouleur ;` ». Comme il n'a plus accès à la variable d'instance « `Coul` », nous n'avons plus de problème de cohérence à l'utilisation des objets de cette classe.

Remarque

Les règles d'habitude veulent que la méthode permettant de modifier la variable d'instance soit bâtie sur le même nom que cette variable d'instance et soit préfixé du verbe anglais « *set* » (mettre ou initialiser).

Ces règles veulent aussi que la méthode chargée de lire la variable d'instance soit préfixée du verbe anglais « *get* » (obtenir).

Nous vous conseillons d'obéir à ces règles d'habitude qui, d'une part, comme vous le verrez offrent certains avantages, et qui, d'autre part, facilitent la lecture d'un source écrit par une autre personne.

La classe de notre exemple devient alors :

```
type
  TFenetre = class
    private
      //...
      Couleur    : integer;
    public
      procedure Redessiner;
      procedure SetCouleur (c : integer);
      function GetCouleur : integer;
      //...
    end;
```

Pratique

Concevez une classe de facture qui gère la notion de HT, de TVA et de TTC sachant que la règle comptable TTC = HT+TVA soit toujours vérifiée, que l'utilisateur de la classe essaye de modifier le HT, la TVA ou le TTC. Permettez aussi à l'utilisateur de la classe de consulter le HT, la TVA et le TTC de la facture. Testez complètement cette classe. Si besoin est, mémorisez aussi le taux de TVA dans la classe.

6.3 DES PROPRIETES « VIRTUELLES »

Le *Pascal* de *Delphi* permet de simuler l'existence de variables d'instance (de *propriétés*) à partir d'élément dans une classe permettant de lire ou d'écrire une valeur dans un objet.

La déclaration dans la classe s'appuyant sur le mot clef « `property` » obéit à la syntaxe suivante :

```
property Nom : TypePropriete read CommentLire write CommentEcrire;
```

Dans cette déclaration « `CommentLire` » représente une expression Pascal fournissant une valeur du type donné et « `CommentEcrire` » fournit un moyen d'écrire dans la propriété.

La partie « `read` » est facultative si la partie « `write` » est présente et vice versa.

Exemple

Reprenons la classe précédente et utilisons ces principes :

```
type
  TFenetre = class
    private
      //...
      FCouleur    : integer;
      procedure SetCouleur (c : integer);
      function GetCouleur : integer;
    public
      procedure Redessiner;
      property Couleur : integer read GetCouleur write SetCouleur;
      //...
    end;
```

Remarque

Les règles d'habitude veulent que la variable d'instance soit préfixée de la lettre « *F* » (comme « *Field* » – « *Champ* »).

Notez aussi que nous avons transféré dans la section private les méthodes « `SetCouleur` » et « `GetCouleur` ». En effet celles-ci ne servent plus à rien quand l'utilisateur peut manipuler la couleur par l'intermédiaire de la propriété « `Couleur` ».

En fait la méthode « GetCouleur » ne sert plus à rien et nous pouvons simplifier la classe comme suit :

```
type
  TFenetre = class
    private
      //...
      FCouleur    : integer;
      procedure SetCouleur (c : integer);
    public
      procedure Redessiner;
      property Couleur : integer read FCouleur write SetCouleur;
      //...
    end;
```

La modification de la couleur de la fenêtre peut maintenant se faire avec l'instruction « `Fen.Couleur := ClGreen ;` ». Par le truchement de la propriété, la méthode « `SetCouleur` » est systématiquement appelée et la fenêtre est redessinée avec la bonne couleur.

Pratique

Reprenez la classe de facture de l'exemple précédent et faites 4 propriétés HT, TauxTVA, TVA et TTC.

Pensez à déclarer chaque méthode de type « `set` » ou « `get` » dans la partie privée.

7. L'HERITAGE

7.1 LA PRESENTATION DE L'HERITAGE

Définitions

L'*héritage* consiste à construire une nouvelle classe à partir d'une classe existante.

La classe de départ est appelée *ancêtre*, *superclasse* ou classe mère. Après plusieurs *héritages* successifs, le terme d'*ancêtre* désigne n'importe quelle classe située en amont d'une sous-classe.

La nouvelle classe s'appelle une *sous-classe* (ou classe fille).

La *sous-classe* récupère toutes les définitions faites dans la classe *ancêtre*. On peut alors modifier ou compléter ces définitions.

Lorsqu'on modifie des définitions existantes on parle de *surcharge*.

La manière dont on précise une relation d'héritage en Pascal Objet consiste à préciser le nom de la classe ancêtre entre parenthèses après le mot clef « `class` ».

Exemple

Considérons notre classe d'adresse :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher;
    procedure Saisir;
    procedure Demenager (NouvRue, NouvCP, NouvVille : string);
    end;

implementation

// ...

end.
```

Dans notre application nous pouvons aussi avoir besoin d'adresse qui possèdent un numéro de téléphone :

```
unit Telephones;

interface

uses Adresses;

type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure Afficher;
    procedure Saisir;
    procedure ChangerTel (NouvTel : string);
    end;

implementation

// ...

end.
```

Comme la classe « *téléphone* » hérite de la classe « *adresse* », un objet de type « *téléphone* » possède les propriétés suivantes : « *Nom* », « *Rue* », « *CP* », « *Ville* » et « *Tel* ».

Un objet « *téléphone* » hérite aussi de la méthode « *Déménager* ».

Par contre les méthodes « *Afficher* » et « *Saisir* » ont dû être *surchargées* car il faudra gérer l'affichage et la saisie du numéro de téléphone. Ce qu'il se passe ici est que la nouvelle définition de ces méthodes cache la définition des méthodes de l'*ancêtre*.

Nous avons ensuite ajouté une nouvelle méthode, « *ChangerTel* » pour gérer le changement de numéro de téléphone.

Tout ceci peut se représenter dans le schéma suivant :

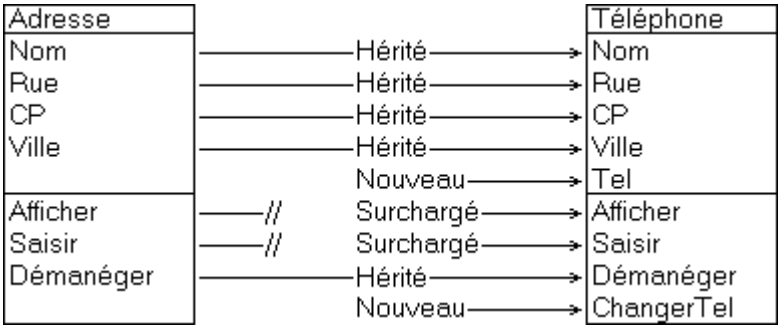


Figure 10. L'héritage

7.2 LA SURCHARGE

L'écriture d'une déclaration dans la *sous-classe* portant le même nom qu'une déclaration dans un *ancêtre* (surcharge) masque la déclaration faite dans l'*ancêtre*. Cependant la déclaration faite originellement dans l'ancêtre existe toujours bien qu'elle ne soit plus visible.

Ainsi dans l'exemple suivant :

```
type

  TAncetre = class
    Numero : string;
    Texte  : string;
    end;

  TDescendant = class (TAncetre)
    Numero : integer;
    end;
```

Les objets de la classe « *descendant* » possèdent 3 variables d'instance : « *Numéro* », « *Texte* » et « *Numéro* ». Le premier numéro est de type texte alors que le deuxième est de type entier.

Cependant dans l'écriture des méthodes ou dans la manipulation d'un descendant seules les variables « *Texte* » (de type texte) et le deuxième « *Numéro* » (de type entier) sont visibles.

Lorsqu'on a besoin de manipuler la déclaration de l'ancêtre qui a été masqué, il faut préfixer l'*identificateur* du mot clef « `inherited` » (« *hérité* ») ; par exemple dans une méthode de la classe « *descendant* », « `inherited Numero` » fera référence au premier numéro de type texte.

Les méthodes surchargées

Dans notre exemple il a fallu *surcharger* les méthodes « `Afficher` » et « `Saisir` » car l'affichage et la saisie doivent intégrer la gestion du nouveau champ « `Tel` ».

Cependant, nous n'allons pas réécrire complètement ces méthodes mais récupérer ce qui a été fait dans l'ancêtre grâce à « `inherited` ».

```
unit Telephones;

interface

uses Adresses;

type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure Afficher;
    procedure Saisir;
    procedure ChangerTel (NouvTel : string);
    end;

implementation

procedure TTelephone.Afficher;
  begin
  inherited Afficher;
  WriteLn ('Tél   : ', Tel);
  end;

procedure TTelephone.Saisir;
  begin
  inherited Saisir;
  Write ('Tél   : '); ReadLn (Tel);
  end;

procedure TTelephone.ChangerTel (NouvTel : string);
  begin
  Tel := NouveTel;
  end;

end.
```

Certaines personnes ont qualifié ce principe de programmation de « *programmation par différence* ».

Remarque

L'écriture d'une méthode surchargée fait relativement souvent référence à la méthode de l'ancêtre, mais ceci n'est pas systématique.

Par exemple si nous concevons une famille d'objets graphiques et si nous faisons hériter un cercle d'un carré, la méthode de dessin du cercle ne peut appeler la méthode de dessin du carré car ces figures géométriques ne se dessinent pas de la même manière.

Pratique

Concevez et testez une classe « *téléphone* » à partir de votre dernière version de la classe « d'*adresse* » — Surcharger toutes les méthodes adéquates (y inclus le/les constructeurs si nécessaire).

Concevez et testez une classe de « *couronne en fonte* » à partie de la classe des « *plaques circulaires* ». Il s'agit de couronne dont le centre est percé, nous gérons ceci en ajoutant la propriété « diamètre du cercle intérieur ».

7.3 L'ATTRIBUT D'ENCAPSULATION « PROTECTED »

Dans la section *6.2.* Les attributs d'encapsulation, page 44, nous avons présenté les attribut d'*encapsulation* « `public` », « `private` » et « `published` », il en existe un quatrième, « `protected` ».

Les informations déclarées dans une section « `protected` » ne sont visibles que depuis les *méthodes* de la *classe* et de ses *sous-classes*.

L'auteur de ce manuel préfère déclarer une section « `protected` » plutôt qu'une section « `private` », car il est en général à la fois l'auteur de ses classes et de ses sous-classes.

7.4 LA RELATION D'HERITAGE

7.4.1 La classe « TObject »

Toutes les classes héritent de la classe « *TObject* ». Ainsi lorsqu'une nouvelle classe est créée sans que l'on précise d'ancêtre, un héritage est fait à partir de la classe « *TObject* ». C'est du « *TObject* » que la nouvelle classe tire ses méthodes « *Create* », « *Free* », « *Destroy* ». Dans la classe « *TObject* » se trouve bien d'autres méthodes qui permettent de manipuler facilement un objet. (Consultez éventuellement l'aide à la rubrique « *TObject* »).

7.4.2 Les hiérarchies de classes et les diagrammes d'héritage

Une classe peut hériter d'une classe qui elle-même peut hériter d'une autre classe, et ainsi de suite. Nous avons donc une *hiérarchie* de classes que l'on peut représenter sous la forme d'un *diagramme d'héritage* :

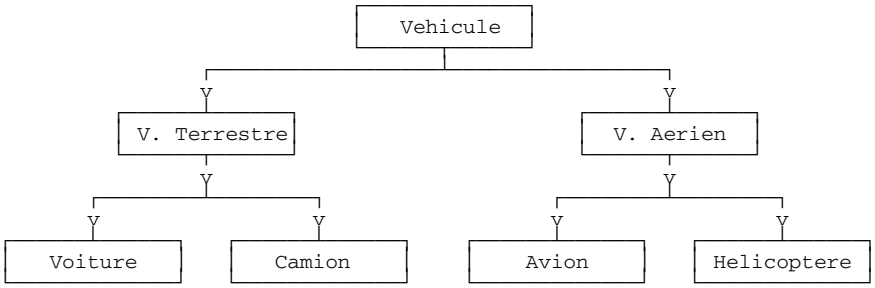


Figure 11. Un diagramme d'héritage

En examinant n'importe quelle hiérarchie, vous devrez remarquer que :

- Les classes du sommet de la *hiérarchie* sont plus simples, comprenant moins de propriétés et de méthodes — Elles sont plus générales.
- Les classes du bas de la *hiérarchie* sont plus précises et entraînent plus de détails.

Nous avons ici affaire à un véritable processus de classification où les objets sont regroupés par caractéristiques communes et générales. Puis la classification affine la définition des objets.

Reprenons la définition d'une classe : « *Ensemble d'objets groupés d'après un ou plusieurs caractères communs* » (Dictionnaire Logos de Bordas).

Nous pouvons ici faire un parallèle avec l'organisation en ensembles : L'ensemble des « véhicules » contient l'ensemble des « véhicules terrestres » et l'ensemble des « véhicules aériens » (nous avons deux sous-ensembles), et chacun de ces sous-ensembles contient lui-même deux sous-ensembles.

Une partie du travail de l'analyste consiste à faire ce travail de classification. Notez que les premières classes que l'analyste perçoit ne sont pas toujours les classes du sommet de la hiérarchie. La démarche pourra être inverse : à partir de notion comme « cuisinière », « machine à laver », « réfrigérateur » l'analyste peut penser à une notion plus générale : « appareil électroménager ». Cette démarche s'appelle la *généralisation*.

Ou au contraire l'analyste a déjà identifié une notion comme « figures géométriques » classe elle-même constitués de « points », de « rectangles », de « cercles »… puis il découvre une nouvelle figure géométrique, le « rond » (sorte de cercle plein). Un « rond » peut hériter d'un « cercle ». Cette démarche au contraire est la *spécialisation*.

7.4.3 Ce que représente la relation d'héritage

Les paragraphes précédents donne déjà une bonne idée de ce que représente la notion d'*héritage* en terme d'analyse. Nous préciserons ceci avec la déclaration suivante :

> **L'héritage représente une relation « Sorte de »**

Ainsi :

- Une « voiture » est une « sorte de » « véhicule terrestre » et par conséquent c'est aussi une « sorte de » « véhicule ».
- Un « avion » est une « sorte de » « véhicule aérien » et par conséquent c'est aussi une « sorte de » « véhicule ».

Ceci peut aussi se comprendre de part le fait qu'une sous-classe hérite de tout ce qui a été défini dans son ancêtre.

Pratique

Classifiez les notions suivantes qui toutes concernent une gestion d'emploi du temps et dessinez la *hiérarchie* :

- Rendez-vous avec un client.
- Contact téléphonique avec un client.
- Elément d'emploi du temps.
- Tâche à faire (sans notion de client).
- Produit à livrer à un client.

Notez que pour concevoir cette *hiérarchie* vous pouvez être amenés à penser à des classes intermédiaires qui ne sont pas décrites dans la liste ci-dessus.

8. LE POLYMORPHISME

Définition

Le *polymorphisme* est l'aptitude à assumer plusieurs formes.

Dans ce chapitre nous verrons comment objets et méthodes peuvent assumer plusieurs formes.

8.1 LA REGLE DE COMPATIBILITE DE TYPE

La règle de *compatibilité de type* est une conséquence de l'*héritage*.

Elle repose sur le fait qu'une *sous-classe* possède toutes les caractéristiques de ses *ancêtres*. De la même manière une instance d'une *sous-classe* possède toutes les caractéristiques d'une instance d'une classe *ancêtre*.

Exemple

Soit la classe d'adresse suivante :

```
unit Adresses;
interface
type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher;
    procedure Saisir;
    end;
implementation
// ...
end.
```

Soit la sous-classe suivante gérant des numéros de téléphone :

```
unit Telephones;
interface
uses Adresses;
type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure Afficher;
    procedure Saisir;
end;
implementation
// ...
end.
```

Le programme principal peut parfaitement contenir les instructions suivantes :

```
program Gestion;

uses
  Forms,
  Adresses in 'Adresses.pas',
  Telephones in 'Telephones.pas';

Var
  QuelquUn : TAdresse;
  UnAutre  : TTelephone;

Begin
  UnAutre := TTelephone.Create;

  UnAutre.Saisir;
  QuelquUn := UnAutre;
  QuelquUn.Afficher;

  UnAutre.Free;
  ReadLn;
end.
```

Par contre l'instruction ci-après est illégale :

```
UnAutre := QuelquUn; // Illégal
```

En effet un objet de la classe « *téléphone* » est une « *sorte d'adresse* » ; ce qui permet d'écrire « `QuelquUn := UnAutre;` » — Par contre une « *adresse* » n'est pas une « *sorte de téléphone* » ; ce qui rend illégal « `UnAutre := QuelquUn;` ».

Notez qu'à l'exécution de « `QuelquUn.Afficher ;` », seule la partie « *Adresse* » de l'objet est vu, et seuls sont affichés les champs « *Nom* », « *Rue* », « *CP* » et « *Ville* ».

En fait on pourrait schématiser l'instruction « `QuelquUn := UnAutre;` » de la manière suivante :

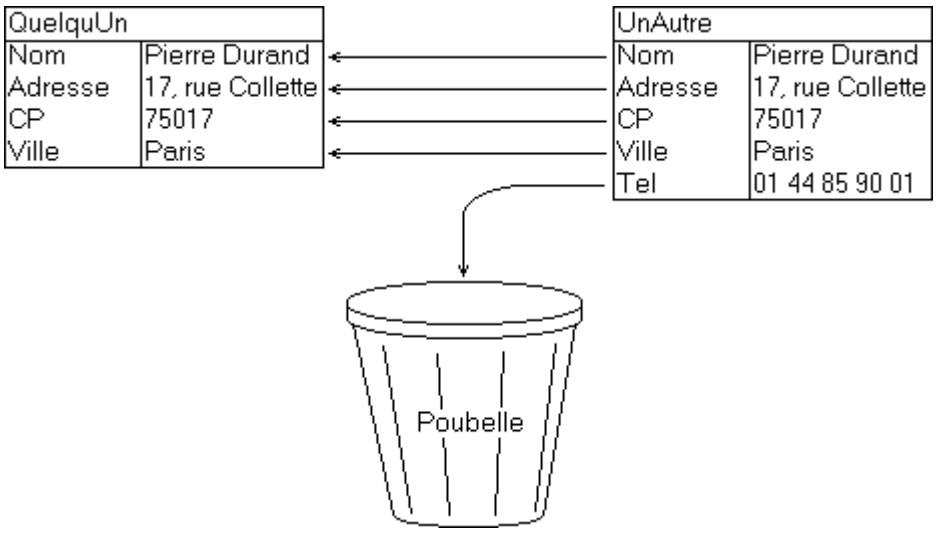


Figure 12. La règle de compatibilité de type

Remarque importante

Ce schéma n'est qu'une vue de l'esprit pour illustrer la règle de compatibilité de type. En effet comme nous vous l'avons déjà signalé, les variables de type objet sont des pointeurs vers des objets.

L'instruction « `QuelquUn := UnAutre;` » ne copie pas un objet dans un autre, mais affecte au pointeur « `QuelquUn` » la même valeur qu'au pointeur « `UnAutre` ». Donc les deux variables objets font référence au même objet, bien que le deuxième pointeur, « `QuelquUn` » ne permet de voir que la partie « *adresse* » de cet objet.

Remarque

Les instructions suivantes sont aussi parfaitement licites :

```
program Gestion;

uses
  Forms,
  Adresses in 'Adresses.pas',
  Telephones in 'Telephones.pas';

Var
  QuelquUn : TAdresse;

Begin
  QuelquUn := TTelephone.Create;

  QuelquUn.Saisir;
  QuelquUn.Afficher;

  QuelquUn.Free;
  ReadLn;
end.
```

En effet « `TTelephone.Create` » est une *instanciation* qui fabrique et retourne un objet de la classe « *téléphone* ». On garde la trace de cet objet au travers d'une variable de la classe « *adresse* ».

Pratique

Avec les classes que vous avez réalisées, « `TAdresse` » et « `TTelephone` » d'une part et « `TPlaqueCirculaire` » et « `TCouronne` » d'autre part, vérifier la règle de compatibilité de type ; tester les instructions légales et les instructions illégales afin de vérifier que le langage obéit bien aux principes de la règle de compatibilité de type.

8.2 DE LA COMPATIBILITE DE TYPE AU POLYMORPHISME

Considérons l'exemple suivant où nous avons créé une procédure destinée à afficher le contenu d'une adresse avant d'en effectuer la saisie :

```
program Gestion;

uses
  Forms,
  Adresses in 'Adresses.pas',
  Telephones in 'Telephones.pas';

Procedure Editer (QuelqueChose : TAdresse);
  Begin
  QuelqueChose.Afficher;
  QuelqueChose.Saisir;
  end;

Var
  QuelquUn : TTelephone;

Begin
  QuelquUn := TTelephone.Create;

  Editer (QuelquUn);

  QuelquUn.Free;
  ReadLn;
end.
```

Cet exemple comme les exemples précédents est syntaxiquement correct, se compile et s'exécute. Cependant comme précédemment la partie « *adresse* » de la fiche contenant un numéro de téléphone est vu.

Ceci s'explique par le fait que le compilateur reconnaît le « `QuelqueChose` » comme étant de la classe « *Adresse* ». Aussi utilise-t-il systématiquement les méthodes « `TAdresse.Afficher` » et « `TAdresse.Saisir` ».

Plutôt que d'utiliser systématiquement ces méthodes « `TAdresse.Afficher` » et « `TAdresse.Saisir` » nous préférerions qu'il utilise les méthodes de la classe à laquelle appartient réellement l'objet concerné. Ainsi comme l'objet instancié est de la classe « téléphone », nous préférerions que dans ce cas il utilise les méthodes « `TTelephone.Afficher` » et « `TTelephone.Saisir` ».

Autrement dit, il faut que le choix des méthodes à utiliser soit fait à l'exécution et non à la compilation (la technique sous-jacente qui sera mise en œuvre s'appelle du reste « *édition de lien dynamique* »).

Une autre manière de poser le problème consisterait à dire que nous souhaitons que l'objet connaisse ses méthodes « `Afficher` » et « `Saisir` ». Le programme pourra alors retrouver les bonnes méthodes « `Afficher` » et « `Saisir` ».

Ceci se réalise au moyen des *méthodes virtuelles* :

Les méthodes virtuelles

Dans une classe lorsqu'une *méthode* est déclarée avec l'attribut « `virtual ;` » (après la déclaration de la méthode), elle peut être surchargée dans les sous-classes avec l'attribut « `override`[29] ; ».

L'objet garde alors la trace de ses *méthodes virtuelles*, et le compilateur appellera toujours la méthode liée à l'objet.

Donc dans notre exemple si les méthodes « `Afficher` » et « `Saisir` » sont *virtuelles* et *surchargées* correctement, la procédure « `Editer` » édite correctement l'objet, que ce soit une « *adresse* » ou une « *adresse avec un numéro de téléphone* ».

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher; virtual;
    procedure Saisir; virtual;
    end;

implementation

// ...

end.
```

```
unit Telephones;

interface

uses Adresses;

type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure Afficher; override;
    procedure Saisir; override;
end;

implementation

// ...

end.
```

Conclusion

Les instructions « `QuelqueChose.Afficher;` » et « `QuelqueChose.Saisir;` » sont donc capable d'assumer plusieurs formes.

Ces *méthodes* sont *polymorphes*, l'appel de ces *méthodes* génère du *polymorphisme* et l'objet « `QuelqueChose` » est *polymorphe*.

[29] Avec le *Pascal Objet* de *Delphi*, le programmeur doit montrer explicitement qu'il veut surcharger et remplacer la *méthode virtuelle* par une nouvelle méthode. D'où le mot clef « `override` ».

Pratique

Testez l'exemple précédent.

Considérez les classes « *PlaqueCirculaire* » et « *Couronne* » et déterminez les méthodes qui devrait être virtuelles ; vérifiez-le avec un petit programme.

Remarque

Les classes suivantes illustrent aussi le *polymorphisme* :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure Afficher; virtual;
    procedure Saisir; virtual;
    procedure Editer;
    end;

implementation

// ...
procedure TAdresse.Editer;
  begin
  Afficher;
  Saisir;
  end;

end.

unit Telephones;

interface

uses Adresses;

type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure Afficher; override;
    procedure Saisir; override;
end;

implementation

// ...

end.
```

Nous attirons votre attention sur ces classes car nous avons souvent vu les programmeurs débutant déclarer la méthode « `Editer` » comme étant *virtuelle*. Ceci n'est pas nécessaire et alourdit inutilement le code. Le *polymorphisme* n'est pas sur l'appel à « `Editer` », mais sur l'appel à « `Afficher` » et sur l'appel à « `Saisir` ». Un indice qui doit vous aider dans la bonne décision est que la méthode « `Editer` » n'a pas besoin d'être surchargée.

Pratique

Concevez une classe de « *factures avec remise* » héritant de la classe des « *factures* ».

La classe de « *factures* » contient les variables d'instance « *FHT* » et « *FTauxTVA* », elle possède les propriétés « *HT* », « *TauxTVA* », « *TVA* », et « *TTC* ». Les règles comptables sont les suivantes :

- TVA = TauxTVA x HT / 100
- TTC = HT + TVA

La classe « *facture avec remise* » contient les variables d'instance précédentes ainsi qu'un pourcentage de remise, « *FTauxRemise* ». Les propriétés sont alors les suivantes « *HT* », « *TauxRemise* », « *Remise* », « *HTRemisé* », « *TauxTVA* », « *TVA* » et « *TTC* ». Les règles comptables sont les suivantes :

- Remise = HT x TauxRemise / 100
- HTRemisé = HT – Remise
- TVA = TauxTVA x HTRemise / 100
- TTC = HTRemisé + TVA

8.3 LA METHODE « FREE » ET LE DESTRUCTEUR « DESTROY »

Dans la classe « `TObject` » le *destructeur* « `Destroy` » est déclaré avec l'attribut « `virtual` ».

La conséquence de ceci est qu'il faut penser à mettre l'attribut « `override` » lorsqu'on *surcharge* le *destructeur* « `destroy` ».

Comme le montreront les exemples qui suivent, déclarer le *destructeur* comme étant *virtuel* offre de nombreux avantages (*polymorphisme* à la destruction).

L'un de ceux-ci résident dans la méthode « `Free` » qui est déclarée dans le « `TObject` » :

```
procedure TObject.Free;
  begin
  if Assigned (self) then
    destroy;
  end;
```

Comme le montre ce code, « `Free` » teste si l'objet auquel on applique cette méthode (« `self` ») existe, et si tel est le cas, le détruit.

Pratique

Reprenez les exemples précédents où nous avons écrit un destructeur « `Suppress` » et transformez-les pour surcharger le destructeur « `Destroy` ».

8.4 TESTER L'APPARTENANCE A UNE CLASSE

L'opérateur « IS »

L'opérateur « `is` » permet de tester si un objet appartient à une classe donnée. Ceci peut s'avérer utile pour appeler ultérieurement des méthodes propres à cette classe.

L'opérateur « AS »

L'opérateur « `as` » permet de considérer temporairement un objet comme étant d'une autre classe que ce que la déclaration n'indique en prime abord.

Exemple

```
If QuelquUn is TTelephone then
  With QuelquUn as TTelephone do
    ChangerNumeroTelephone ('01 44 85 90 01');
```

Ou encore

```
If QuelquUn is TTelephone then
  (QuelquUn as TTelephone).ChangerNumeroTelephone ('...');
```

La conversion de type à la mode du C++

On peut aussi considérer une variable Pascal comme étant d'un autre type grâce à la syntaxe « `AutreType (Valeur)` ».

Exemple

```
If QuelquUn is TTelephone then
  Ttelephone(QuelquUn).ChangerNumeroTelephone ('...');
```

Différence entre « AS » et la conversion de type

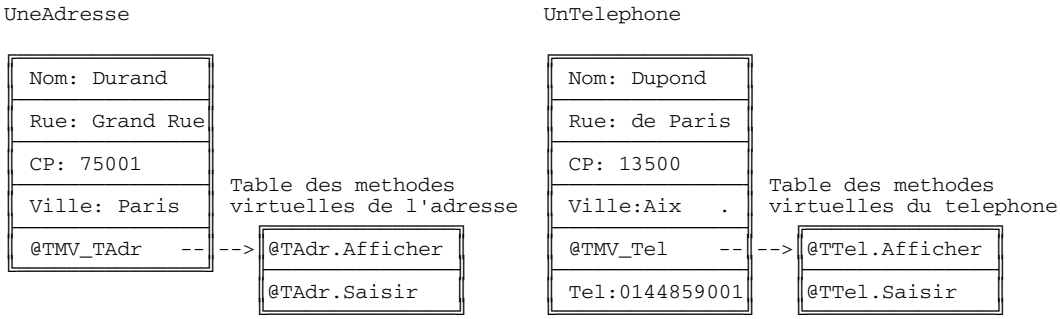
Avec l'opérateur « AS » si l'objet ciblé n'est pas de la classe concernée, le programme génère une erreur (exception) qui peut être analysée et contrôlée.

Avec une conversion de type style C++, si l'objet ciblé n'est pas du type concerné, le résultat est complètement aléatoire. (Dans l'exemple ci-dessus comme l'objet a été testé par rapport à la classe on peut utiliser sans risque une conversion de type style C++).

8.5 COMMENT LE COMPILATEUR TRAITE-T-IL LES METHODES VIRTUELLES

Dès qu'une *classe* contient des *méthodes virtuelles*, le compilateur alloue dans l'objet un pointeur qui fera référence à une table contenant les adresse de toutes les *méthodes virtuelles* de cette *classe* — La *table des méthodes virtuelles* (*TMV* ou *Virtual Method Table*, *VMT* en anglais).

Le schéma ci-dessous illustre cette organisation en mémoire.



A la génération du code, lorsque le compilateur détecte un appel à une méthode virtuelle, il génère le code suivant :

- Examen du pointeur vers la *TMV*.
- Obtention de l'adresse adéquate dans cette table
- Exécution de la méthode à cette adresse.

Examinez alors ce qu'il se passe avec ces principes lorsqu'on exécute la procédure « Editer » pour une « adresse », puis lorsqu'on exécute cette même procédure pour une « adresse avec téléphone ».

Cette technique est appelée « *édition de lien dynamique* ».

8.6 EXERCICE RECAPITULATIF

Nous allons concevoir une nouvelle classe pour gérer un facturier. Ce facturier pourra contenir des factures avec remises et des factures sans remises, telles que réalisées dans le dernier exercice. Le nombre maximum de facture que pourra stocker un facturier est arbitrairement fixé à 1000.

Essentiellement un « facturier » encapsulera un tableau de 1000 pointeurs vers des « factures » et un nombre indiquant combien de factures sont réellement stockées dans ce tableau :

```
unit Facturier;

interface

uses Factures, FacturesRemisees;

const
  MAX_FACTURE = 1000;

type
  TFacturier = class
    protected
      Factures : array [1..MAX_FACTURE] of TFacture;
      Nombre : integer;
    public
      // A écrire
    end;

implementation

// A ecrire

end.
```

Les comportements de ce facturier sont les suivants :

- Construction d'un facturier : aucune facture présente.
- Destruction d'un facturier : détruire toutes les factures qui pourraient y avoir été stockées.
- Une méthode de type « `procedure Editer ;` » doit afficher un menu qui comporte les options suivantes :
  - Ajout d'une facture (Prévoir la saisie de la facture avant son insertion).
  - Ajout d'une facture avec remise (Prévoir la saisie de la facture avant son insertion).
  - Liste de toutes les factures
  - Modification d'une facture (demander le numéro de la facture avant)
  - Voir une facture (demander le numéro de la facture avant)
  - TTC global : additionne tous les TTC
  - TVA globale : additionne toutes les TVA
  - Quitter

- Pour ce faire vous devrez créer autant de méthodes élémentaires que nécessaire. Par exemple :
  - Une méthode « `procedure Ajouter (F : TFacture) ;` » doit permettre d'ajouter une facture à la liste.
  - Une méthode « `function Retrouver (i : integer) : TFacture ;` » doit permettre de retrouver une facture de numéro donné.
  - Une méthode « `function TotalTTC : double` » doit calculer le total TTC de toutes les factures présentes.

Le programme principal devra être le suivant :

```
program GererFactures;

uses
  Forms,
  Factures in 'Factures.pas',
  FacturesRemisees in 'FacturesRemisees.pas',
  Facturiers in 'Facturiers.pas';

Var
  Facturier : Tfacturier;

begin
  Facturier := Tfacturier.Create;
  Facturier.Editer;
  Facturier.Free;
end.
```

9. LES CLASSES ABSTRAITES

En français le terme « d'*abstraction* » signifie : « *dégager une idée* ».

En fait le processus de *généralisation* va nous amener tout naturellement à la notion de *classe abstraite*.

Considérons dans une application que nous voulons gérer des périphériques de type divers et que nous voulions créer des *classes* pour les gérer — Voici la liste des périphériques concernés :

- Ecran
- Imprimante
- Fichiers sur un disque dur
- Liaison série

Tous ces périphériques (ou presque) vont posséder une *méthode* du type « Lire des données » et une autre du type « Ecrire des données », et il semble naturel de vouloir rassembler ces objets dans une même hiérarchie.

Pour construire la *hiérarchie*, le principe de *généralisation* nous amène tout naturellement à envisager une cinquième *classe* qui servira d'*ancêtre* commun, par exemple la classe « *Périphérique* » :

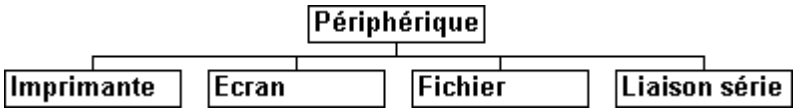


Figure 13. Exemple de classe abstraite

Le problème est que pour cette classe de périphérique, nous ne pouvons écrire les méthodes « *Lire* » et « *Ecrire* », et elles devront toutes être *surchargées*.

La *classe* « périphérique » est une *classe abstraite*, les méthodes « Lire » et écrire sont des *méthodes abstraites* aussi appelées *méthodes virtuelles pures*.

Définitions

Une *méthode abstraite* ou (*virtuelle pure*) est une *méthode* dont on ne peut écrire le code. Elle devra obligatoirement être *surchargée* dans une *sous-classe*.

Une *classe abstraite* est une classe qui ne peut être *instanciée*. Une *classe* qui contient des *méthodes abstraites* doit toujours être déclarée *abstraite* elle aussi.

Syntaxe

Une *méthode abstraite* se déclare avec le mot clef « `abstract` » et un point virgule « ; » après la déclaration « `virtual;` » associée à la *méthode*.

Exemple

Déclarons la classe de périphérique précédente tout en lui adjoignant une gestion d'erreur qui sera héritée par tous les périphériques :

```
unit Periphériques;

interface

type
  TPeripherique = class
    protected
      Ferreur : integer;
    public
      procedure Ecrire (Car : char); virtual; abstract;
      function Lire : Char; virtual; abstract;
      function Erreur : integer;
    end;

implementation

function Tperipherique.Erreur : integer;
  begin
  Result := FErreur;
  FErreur := 0;
  End;

end.
```

Remarque

L'instruction « `TPeripherique.Create` » est illicite car la classe possède des méthodes *virtuelles pures* et est donc *abstraite*.

Par contre la déclaration « `var TerminalParDefaut : TPeripherique` » est valide. On peut alors imaginer qu'un module d'initialisation instancie le périphérique par défaut adéquat « `TerminalParDefaut := TEcran.Create (…) ;` » et n'importe quelle autre partie du programme pourra accéder à cet écran par exemple « `TerminalParDefaut.Ecrire ('C') ;` » — Le polymorphisme est à l'œuvre.

Ce sera souvent une bonne idée de mettre une classe abstraite au sommet de vos *hiérarchies*. Par exemple dans une application *Windows*, vous serez amenés à créer des fenêtres et des boîtes de dialogues diverses et variées. Chacune d'entre elle sera une instance d'une *classe* particulière. Il serait bon que toutes ces *classes* héritent de la même *classe abstraite* vous donnant par là un point d'entrée vous permettant de travailler sur toutes les fenêtres d'un coup. Une application peut par exemple installer un mécanisme de traduction des libellés dans cette classe abstraite et toutes les fenêtres en héritent.

Enrichissons la classe de périphérique pour lire et écrire du texte :

```
unit Periphériques;

interface

type
  TPeripherique = class
    protected
      Ferreur : integer;
    public
      procedure Ecrire (Car : char); virtual; abstract;
      function Lire : Char; virtual; abstract;
      procedure EcireTexte (Texte : string);
      function LireTexte : string;
      //...
    end;

implementation

procedure TPeripherique.EcrireTexte (Texte : string);
  var
    i : integer;
  begin
  for i := 1 to Length (Texte) do
    Ecrire (Texte [i]);
  Ecrire (#$0d); // retour chariot
  Ecrire (#$0a); // nouvelle ligne
  end;

function LireTexte : string;
  var
    Entree : Char;
  begin
  Result := '';
  While true do
    Begin
    Entree := Lire;
    case Entree of
      #$0d : break; // retour chariot
      #$0a : ;      // nouvelle ligne
      else Result := Result + Entree;
      end;
    end;
  end;

end.
```

Nous retrouvons le *polymorphisme* à l'œuvre avec « `PeripheriqueParDefaut.Ecrire ('Bonjour') ;` ».

Dans les sous-classes seules les méthodes « `Lire` » et « `Ecrire` » sont à définir pour que cela marche.

Revenons à nos classes d'adresse et de téléphone. Nous pouvons leur adjoindre des méthodes leur permettant de s'écrire sur n'importe quel périphérique :

```
unit Adresses;

interface

type
  TAdresse = class
    Nom   : string;
    Rue   : string;
    CP    : string;
    Ville : string;
    procedure EcrireSur (Sortie : TPeripherique); virtual;
    //...
    end;

implementation

procedure TAdresse.EcrireSur (Sortie : TPeripherique);
  begin
  Sortie.EcrireTexte (Nom);
  Sortie.EcrireTexte (Rue);
  Sortie.EcrireTexte (CP);
  Sortie.Ecrire (Ville);
  end;

//...

end.


unit Telephones;

interface

uses Adresses;

type
  TTelephone = class (TAdresse)
    Tel   : string;
    procedure EcrireSur (Sortie : TPeripherique); virtual;
    //...
    end;

implementation

procedure TTelephone.EcrireSur (Sortie : TPeripherique);
  begin
  inherited Ecrire (Sortie);
  Sortie.Ecrire (Tel);
  end;

//...

end.
```

Nous retrouvons alors le *polymorphisme* à l'œuvre dans toute sa splendeur :

```
program Premier;

//...

var
  Periph : TPeripherique;
  Adr    : TAdresse;

begin
//...

Adr.EcrireSur (Periph);

//...
end.
```

Quelle que soit la nature de l'adresse et du périphérique, quelles que soient les manières dont ils ont été instanciés, cela fonctionne grâce au *polymorphisme*.

10. LES VARIABLES ET LES METHODES DE CLASSE

Variable de classe

En terme objet, une *variable de classe* est une variable qui sera partagée par tous les objets d'une même classe.

Supposez que nous voulions compter tous les objets de type facture qu'une application crée. Ce compteur ne devra pas être stocké dans chaque facture, mais être commun à toutes les factures.

Le *Pascal Objet* de *Delphi* n'offre pas de mécanisme syntaxique permettant de déclarer des *variables de classe*. Cependant celles-ci peuvent être aisément simulées en les déclarant dans la partie privée d'une unité. On pourra compter et décompter les factures dans le *constructeur* et le *destructeur*. La classe des factures devient donc :

```
unit Factures;

interface

type
  TFacture = class
    protected
      Libelle : string;
      HT      : double;
      TauxTVA : double;
    public
      constructor Create;
      destructor Destroy; override;
      //...
    end;

implementation

var
  Compteur : integer;

constructeur TFacture.Create;
begin
  Compteur := Compteur + 1;
end;

destructor TFacture.Destroy;
begin
  Compteur := Compteur -1;
  Inherited Destroy;
end;

//...

initialization
  Compteur := 0;
end.
```

Remarque

A la fin d'une *unité*, vous pouvez ajouter une section « `initialization` » destiné à initialiser l'*unité*. Le code de cette section sera exécuté avant le programme principal.

De la même manière à la fin d'une *unité*, vous pouvez ajouter une section « `finalization` » destiné à gérer la destruction de l'*unité*. Le code de cette section sera exécuté après le programme principal.

Méthode de classe

Une *méthode de classe* est une *méthode* qui sera partagée par tous les objets d'une même classe. Une *méthode de classe* se déclare avec le préfixe « `class` ». Il est inutile de connaître un objet particulier pour manipuler une *méthode de classe*, on peut en demander l'exécution avec « `NomDeClasse.NomDeMéthode ;` ».

Reprenons notre exemple, il nous faut un moyen pour consulter le compteur que nous venons de créer en tant que *variable de classe*. Ceci peut se faire avec une méthode de classe, « `Combien` » :

```
unit Factures;

interface

type
  TFacture = class
    protected
      Libelle : string;
      HT      : double;
      TauxTVA : double;
    public
      constructor Create;
      destructor Destroy; override;
      class function Combien : integer;
      //...
    end;

implementation

var
  Compteur : integer;

constructeur TFacture.Create;
begin
  Compteur := Compteur + 1;
end;

destructor TFacture.Destroy;
begin
  Compteur := Compteur -1;
  Inherited Destroy;
end;

class function TFacture.Combien : integer;
  begin
  Result := Compteur;
  end;

//...

initialization
  Compteur := 0;
end.
```

L'expression « `TFacture.Combien` » retourne maintenant le nombre de facture instanciées.

Remarque

Une *méthode de classe* ne peut manipuler les *variables et les méthodes d'instance*. En effet il faut un objet particulier pour manipuler ces entités.

L'objet implicite « `self` » est inaccessible dans une *méthode de classe* pour les mêmes raisons.

Pratique

Revoyez le facturier qui a été écrit dans un exercice précédent :

- Ajouter le compteur de facture avec les principes précédents.
- Ajouter dans le menu une commande pour afficher le nombre de facture obtenu par ce principe.
- Après la destruction du facturier afficher le compteur de facture et vérifiez que celui-ci soit retombé à 0.

11. LA GESTION DES EXCEPTIONS

La gestion des *exceptions* est ce qui vous permettra de créer des applications sûres. Une *exception* n'est pas une anomalie système, mais quelque chose que vous considéreriez, vous ou le concepteur d'un objet, comme étant une anomalie — Par exemple vous *instanciez* un objet, et il n'y a pas assez de mémoire pour réaliser cette opération.

Définition

Une exception est un événement ou une condition qui, lorsqu'il se produit, interrompt l'exécution d'un programme. Une exception est aussi un objet qui contient des informations sur l'erreur survenue.

Exemple d'exceptions

Voici un exemple de liste de classes d'exception telles que vues avec le *scruteur d'objets*[30] :

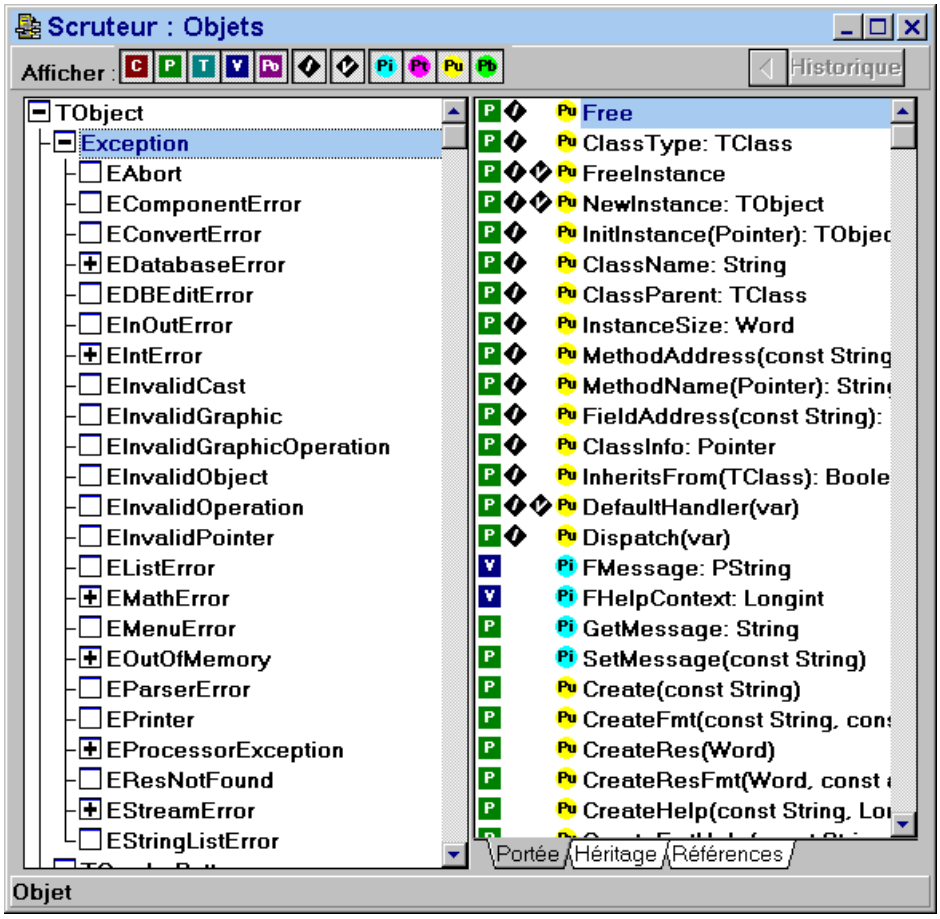


Figure 14. Liste d'exceptions vues avec le scruteur d'objets.

[30] Le *scruteur d'objet* permet de voir l'anatomie de toutes les classes liées à un programme.

11.1 DECLARATION D'UN TYPE D'OBJET EXCEPTION

Comme les *exceptions* sont des objets, définir un nouveau type d'exception est aussi simple que déclarer une nouvelle *classe*. Bien que vous puissiez faire de n'importe quelle instance d'objet une exception, les gestionnaires d'exception standard ne traitent que les exceptions qui descendent de la classe **EXCEPTION**.

Il peut donc être judicieux de dériver tout nouveau type d'exception de cette *classe* ou de l'une des autres *exceptions* standard.

Ainsi, comme le programme peut déjà incorporer du code pour gérer les exceptions standard, votre nouvelle exception sera prise en compte correctement et traitée.

Exemple

```
type
  EMyException = class(Exception);
```

11.2 DECLENCHER UNE EXCEPTION

Pour indiquer une condition d'erreur dans une application, vous pouvez provoquer une *exception* qui intègre la *construction* d'une *instance* d'une classe d'*Exception* et interrompre le programme avec le mot réservé **RAISE**.

Pour provoquer une exception, appelez le mot réservé **RAISE**, suivi d'une instance d'un objet exception.

Quand un gestionnaire d'exception traite l'exception, il se termine en détruisant l'instance de l'exception, ce qui fait que vous n'avez jamais à vous en charger.

Exemples

A partir de la déclaration suivante,

```
type
  EPasswordInvalid = class(Exception);
```

Vous pouvez à tout moment provoquer une exception de "mot de passe incorrect" en appelant **RAISE** avec une instance de **EPASSWORDINVALID**, comme cela :

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Ce mot de passe n'est pas le bon');
```

Choix de l'adresse de l'exception

Provoquer une exception place la variable **ERRORADDR** dans l'unité **SYSTEM**, et lui donne l'adresse à laquelle l'application a provoqué l'exception. Vous pouvez utiliser **ERRORADDR** dans vos gestionnaires d'exception, par exemple, pour indiquer à l'utilisateur où s'est produit l'erreur. Vous pouvez aussi spécifier une valeur pour **ERRORADDR** lorsque vous provoquez une exception.

Pour spécifier une adresse d'exception, ajoutez le mot réservé **AT** après l'exception, suivi d'une expression d'adresse.

11.3 LE TEST ET LE TRAITEMENT DES EXCEPTIONS

Il existe deux méthodes pour intercepter et/ou traiter une exception :

- Protection du code et exécution d'un code de nettoyage (**TRY** ... **FINALLY**).
- Protection du code et si une exception se produit, déroutage du programme vers un code de traitement de l'exception (**TRY** ... **EXCEPT**).

11.3.1 Le bloc try ... finally

Syntaxe

```
try
   Instructions
finally
   Code de nettoyage
end ;
```

Les instructions de la partie **FINALLY** d'un bloc **TRY** ... **FINALLY** sont toujours exécutées, même s'il se produit une *exception*.

Les instructions d'un bloc **TRY** ... **FINALLY** sont exécutées normalement, sauf s'il se produit une *exception* ; dans ce cas, les instructions de la partie **FINALLY** sont exécutées. Une fois l'instruction `end` atteinte, l'exception se propage à nouveau. Le bloc **TRY** ... **FINALLY** ne gère pas l'*exception* en tant que tel mais permet de s'assurer qu'un groupe d'instruction sera exécuté même s'il y a un problème ce qui permet de protéger du code vital, par exemple la destruction d'un objet :

```
Adresse := TAdresse.Create;
try

   //...

finally
   Adresse.Free;
end ;
```

Ainsi nous sommes sûrs que l'appel du destructeur sera fait. Nous pouvons aussi combiner ceci avec un bloc « `with` » :

```
With TFacture.Create do
try
   HT := 1000;
   TauxTVA := 20.6
   Afficher;
   HT := HT / 0;              // provoquera une exception
   Afficher;                  // ne sera pas exécuté
finally
   Free;                      // sera executée
end ;
// l'exception se propage vers le prochain bloc try
```

Remarque

Dans l'exemple ci-dessus, remarquez que nous pouvons manipuler un objet sans passer par une variable. En effet le constructeur « `TFacture.Create` » instancie un objet et retourne l'adresse de l'objet instancié. Cet objet est ensuite manipulé grâce au « `with` » qui pilote le bloc « `try … finally` ». Le « `finally` » permet donc de s'assurer que l'objet qui a été instancié soit détruit.

11.3.2 Le bloc try...except

Un bloc qui gère des exceptions est un bloc try ... except.

Syntaxe

```
try
   Instructions
except
   on InstanceException : ClasseException do begin ... end ;
   on InstanceException : ClasseException do begin ... end ;
   on InstanceException : ClasseException do begin ... end ;
   else begin ... end ;
end ;
```

Où **ClasseException** est une classe d'identification d'exception (Voir 11.1, *Déclaration d'un type d'objet exception*) et **InstanceException** un paramètre propre au gestionnaire d'exception permettant de récupérer l'instance d'exception qui a été générée.

Dans la partie **try** du bloc, les instructions s'exécutent dans l'ordre normal, sauf s'il se produit une *exception* ; dans ce cas, l'exécution passe directement à la partie **except**. Si aucune *exception* ne se produit, le bloc se termine sans utiliser les parties **except** ou **else**.

La partie **except** est une liste d'exceptions spécifiques et des réponses qui y sont faites, chacune d'elles étant une instruction **on** ... **do**. Si aucune des instructions **on** ... **do** ne s'applique à l'*exception* active, c'est le gestionnaire d'exception par défaut de la partie **else** qui s'exécute. Lorsqu'un gestionnaire (spécifique ou par défaut) gère l'exception, le bloc se termine.

L'exécution ne reprend pas dans le bloc à la suite d'une exception.

Exemple

L'exemple suivant définit un gestionnaire d'exception pour la division par zéro de façon à obtenir un résultat par défaut :

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems;
  except
    on EDivByZero do Result := 0;
  end;
end;
```

Ceci vous évite de tester la présence d'une valeur zéro à chaque appel à la fonction. Voici une fonction équivalente qui ne tire pas parti des exceptions :

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then
    Result := Sum div NumberOfItems
  else Result := 0;
end;
```

12. D'AUTRES EXTENSIONS AU LANGAGE

Les points essentiels du *Pascal Objet* de *Delphi* ont été abordés. Ce chapitre se contente de lister un certain nombre de particularités du *Pascal Objet*. Il se présente comme une liste de références diverses.

12.1 LES TYPES DE DONNEES

12.1.1 Les chaînes de caractères

La structure des chaînes de caractères, « `string` », a changé depuis Borland Pascal — Les chaînes sont maintenant de taille quelconque, peuvent stocker jusqu'à 2 Go et sont allouée dynamiquement et automatiquement par le programme.

Pour connaître la taille d'une chaîne de caractères, utilisez la fonction « `Lenght` » (consultez l'aide) ; par exemple « `Length ('Coucou')` » retourne « `6` ».

Pour modifier la taille d'une chaîne de caractère, vous pouvez utiliser la fonction « `SetLength` » (consultez l'aide).

Les anciennes chaînes de caractères peuvent être manipulées avec la déclaration « `ShortString` » ou en modifiant une option de compilation.

12.1.2 La date et l'heure

Nous trouvons un nouveau type de données, « `TDateTime` » qui permet de manipuler des informations temporelles.

Un « `TDateTime` » est compatible avec un nombre en virgule flottante ; la partie entière détermine le nombre de jours écoulés depuis le « *31 décembre 1899* », la partie décimale correspond à une fraction de journée (donc « 0.5 » correspond à « *12 heures* »).

La bibliothèque « *SysUtils* » contient des utilitaires permettant de formater, déformater et convertir facilement des données temporelles.

12.1.3 Les « Variant »

Le type « `Variant` » permet de manipuler des informations non typées. On peut donc y stocker un nombre, une chaîne de caractères, un objet…

Ce type est à l'opposé de l'esprit du langage *Pascal* qui est au contraire fortement typé et nous vous le déconseillons fortement ; il a cependant été rendu nécessaire pour pouvoir manipuler certaines fonctions de *Windows*.

Dans un usage courant il est à proscrire.

12.1.4 Les tableaux ouverts

Les paramètres de procédure ou de fonction peuvent être des *tableaux ouverts*, c'est à dire des tableaux dont l'indice maximum n'est pas précisé (l'indice minimum vaut « 0 »). Ceci permet d'écrire des procédures et des fonctions de traitement de tableaux sans considérations de taille.

La déclaration du paramètre est « `Tableau : array of QuelqueChose` » ; « `QuelqueChose` » étant un type connu.

L'indice maximum du tableau peut être connu avec la fonction « `High` ».

Exemple

```
function Somme(const A: array of Real): Real;

var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Somme := S;
end;
```

Les constantes de type « tableaux ouverts »

On peut passer une expression littérale comme paramètre à une procédure attendant un tableau ouvert. Le tableau est décrit entre *crochets*, « `[...]` ».

Ainsi la procédure précédente peut-elle être appelée de la manière suivante :

```
Total := Somme ([10, 20.5, 15.2, 17.3]);
```

12.1.5 Les tableaux dynamiques

Un tableau peut être déclaré sans indice avec la déclaration « `Tableau : array of QuelqueChose` » ; « `QuelqueChose` » étant un type connu. Il s'agit d'un *tableau dynamique*.

Vous devrez ensuite initialiser la taille du tableau par une instruction utilisant la procédure « `SetLength` », puis vous pourrez changer la taille du tableau avec cette même procédure.

L'indice minimum vaut « 0 », l'indice maximum peut être connu par la fonction « `High` ».

12.2 LES FONCTIONS UTILITAIRES DE « SYSUTILS »

L'unité « *SysUtils* » contient de nombreux utilitaires, notamment en ce qui concerne la conversion de données. Signalons entre autres :

- « `IntToStr` » : fonction convertissant un entier en chaîne de caractères ; par exemple « `IntToStr (10)` » donne « `'10'` ».
- « `StrToInt` » : fonction convertissant une chaîne de caractères en entiers ; par exemple « `StrToInt ('10')` » donne « `10` ».
- « `DateToStr` » : fonction convertissant une valeur « `TDateTime` » en chaîne de caractères ; par exemple « `DateToStr (1)` » donne « `'01/01/1900'` ».
- « `StrToDate` » : fonction convertissant une chaîne de caractères en une valeur « `TDateTime` » ; par exemple « `StrToDate ('31/12/1899')` » donne « `1` ».
- De la même manière nous trouvons « `StrToDateTime` », « `StrToTime` », « `TimeToStr` », « `DateTimeToStr` »…

12.3 GESTION DES CLASSES ET DES OBJETS

12.3.1 Les pointeurs sur une méthode d'un objet

Tout comme les déclarations « `procedure` » et « `function` » du *Pascal* permettent de déclarer des pointeurs sur des procédures ou des fonctions, les déclarations « `procedure of object` » et « `function of object` » permettent de déclarer des pointeurs sur des *méthodes* — Consultez l'aide.

12.3.2 Les pointeurs sur une classe

La déclaration « `class of …` » permet de déclarer des pointeurs sur des *classes*.

De puissantes possibilités de *polymorphisme* existent derrière cette notion (on peut affecter un pointeur vers une sous-classe de la classe déclarée et bénéficier ainsi du *polymorphisme* — Il sera alors souvent judicieux de déclarer les constructeurs « *virtual* »).

Consultez l'aide.

13. GLOSSAIRE

Abstraction En français le terme « d'*abstraction* » signifie : « *dégager une idée* ».L'abstraction peut reposer sur l'observation ; par exemple on observe un groupe de 3 allumettes, un groupe de 3 arbres, un groupe de 3 voitures… et l'on finit par dégager l'idée de la quantité 3. Le terme « *abstrait* » s'oppose au terme « *concret* » qui lui indique une réalité physique et palpable.

Abstrait Une *méthode abstraite* (ou *méthode virtuelle pure*) est une *méthode* dont on ne peut écrire le code. Elle devra obligatoirement être *redéfinie* dans une *sous-classe*.

Une *classe abstraite* est une classe qui ne peut être *instanciée*. Une *classe* qui contient des *méthodes abstraites* est toujours *abstraite*. Une *classe abstraite* est destinée à être l'*ancêtre* commun de plusieurs *classes* possédant des caractéristiques similaires. Notez que le *Pascal Objet* de *Delphi* ne permet pas de formaliser la notion de classe abstraite ; d'autre langage permettent de signaler qu'une classe est abstraite (mais pas *Delphi*).

Ancêtre Qualifie toute *classe* située en amont après un *héritage* ou une succession d'*héritage*.

Classe « *Ensemble d'objets groupés d'après un ou plusieurs caractères communs* » (Dictionnaire Logos de Bordas).

Classe abstraite Voir *Abstrait*.

Compatibilité de type Principe selon lequel une classe est compatible avec ses *ancêtres*.

Comportement « *Le comportement est la façon dont un objet agit et réagit, en termes de changement de ses états et de circulation de messages* ». (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Constructeur Méthode appelée lors de la création d'un objet afin d'en assurer l'initialisation complète et totale.

Construction Voir *Instanciation*.

Contenance (relation entre objets) Relation qui existe entre deux objets lorsque l'un de ces objets est le propriétaire de l'autre. Lorsque l'objet propriétaire est détruit, tous les objets contenus sont aussi détruit. Par exemple une « *voiture* » est constituée d'une « *carrosserie* », d'un « *moteur* », de quatre « *roues* »… La voiture est alors le conteneur des autres objets.

Cycle de vie d'un objet Un objet est *construit* (*instancié*), puis on le manipule (il reçoit des *messages* des autres objets), puis il est *détruit*. Le programmeur qui *instancie* un objet et responsable de sa *destruction* future.

Destruction Opération par laquelle un objet arrive à la fin de son existence dans le système. Toutes les opérations de libération de ressource sont alors effectuées (entre autres la libération de la mémoire allouée à l'objet). Toutes les opérations destinées à rendre cohérente la fin de l'utilisation de cet objet sont entreprises.

Destructeur Méthode appelée lors de la destruction d'un objet afin de restituer un contexte de travail propre.

Encapsulation Distinction dans une classe entre la partie concernant l'utilisation d'un objet ou d'une classe et la partie concernant les choix du programmeur de la classe quant à la mise en œuvre des fonctionnalités de la classe. L'encapsulation revient à considérer que dans une classe il y a une interface ou partie publique (comment on utilise la classe) et une partie propre au programmeur qui ne concerne que ces choix sur la manière dont on fournit la partie publique (partie privée ou protégée).

Etat « *L'état d'un objet englobe toutes les propriétés (habituellement statiques) de l'objet, plus les valeurs courantes (habituellement dynamiques) de chacune de ces propriétés.* » (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Généralisation Démarche de l'analyste objet qui part de classes données pour trouver une classe plus générale. Cette classe plus générale devient l'ancêtre des classes de départ. Par exemple à partir des classes « Fichier », « Imprimante », « Liaison série »… L'analyste va définir la classe « Périphérique ». Cette classe de « périphérique » va permettre de définir les états et les comportements communs aux « fichiers », « imprimantes » et « liaisons séries ». Voir aussi « *Abstraction* » et « *Spécialisation* ».

Héritage Définition d'une nouvelle *classe* à partir d'une *classe* existante. La nouvelle *classe hérite* (récupère) de toutes les caractéristiques (*propriétés* et *méthodes*) de la *classe* existante. Le programmeur pourra lors enrichir la *classe* avec de nouvelles *propriétés* ou *méthodes* ou la modifier en redéfinissant des *méthodes* existantes.

Hiérarchie Ensemble des *classes*, à partir d'une *classe ancêtre*, comprenant toutes les *sous-classes* construites par *héritage* à partir de cet *ancêtre*.

Identificateur Nom choisi par un programmeur pour identifier un élément dans un programme. Ce nom sert alors à décrire ou manipuler l'élément concerné. Il peut s'agir d'un nom de classe, d'un nom d'objet...

Identité « *L'identité est cette propriété d'un objet qui le distingue de tous les autres objets* ». (Conception orientée objets et applications — Grady Booch, édité par Addison Wesley).

Instance Terme anglais qui signifie : « *exemple, cas, illustration* » (Traduit du Webster NewWorld Dictionary). En programmation orientée objet, le terme d'*instance* est utilisé pour désigner un objet particulier ou un objet précis. En fait, on pourrait considérer les termes *objet* et *instance* comme étant synonyme.

Instanciation Création d'un objet.

Instancier Créer un objet.

Interface En programmation orientée objet, le terme d'*interface* désigne l'ensemble des membres (*propriétés* et *méthodes*) définis dans une *classe*, accessible de puis l'extérieur de la *classe* afin de communiquer avec les objets de la *classe* ou avec la *classe*.

Méthode (ou fonction membre) Suite d'instructions exécutées par un objet ou une classe en réponse à un *message*.

Méthode de classe *Méthode* commune à tous les objets d'une classe qui peut aussi être manipulée sans connaître un objet particulier dans la classe — Voir aussi « *Propriété de classe* ».

Méthode d'instance *Méthode* pouvant être appliquée à un objet.

Méthode virtuelle Lorsqu'une *méthode* est déclarée *virtuelle* (grâce au mot clef « `virtual` »), chaque objet de la classe en garde la trace. Ceci permet de *surcharger* cette méthode après un ou plusieurs héritages et de personnaliser cette méthode plus bas dans la *hiérarchie* (chaque objet des *sous-classes* gardant la trace de ses propres *méthodes virtuelles*). Ceci permet de déclencher à un niveau *abstrait* (au sommet de la *hiérarchie*) des opérations qui définies en bas de la *hiérarchie*.

Méthode virtuelle pure Voir *Abstrait*.

Message 1/ Sollicitation envoyée vers un objet pour lui demander de faire quelque chose ou pour le prévenir qu'un événement vient de se produire. L'objet répond à un message en exécutant du programme (Voir méthode).

2/ Sous *Windows*, information que le système envoie à un programme pour lui signaler qu'un événement vient de se produire (exemple : frappe d'une touche sur le clavier, déplacement de la souris...).

Objet Un objet est constitué d'un *état*, de *comportements* et d'une *identité*.

Pascal Langage de programmation créé en 1969 par un professeur de *Zurich* (*N.Wirth*) afin de donner de bonnes habitudes aux programmeurs. Ce langage s'avère être aussi un puissant outil de programmation permettant de réaliser des programmes performants.

Pascal Objet Langage *Pascal* auquel quelques extensions (essentiellement des déclarations) permettent de mettre en œuvre tous les principes de la *Programmation Orientée Objets* (*POO*)

Polymorphisme Aptitude à assumer plusieurs formes.

POO *Programmation Orientée Objets*.

Programmation Orientée Objets En première approche la *Programmation Orientée Objets* ou *POO* est une manière d'organiser la création de programmes en fonction d'un principe de modularité. A cause de cette modularité, les programmes deviennent beaucoup plus faciles à écrire, les problèmes complexes peuvent être découpés en petits problèmes faciles à traiter, les solutions sont alors facilement assemblées, le code devient plus facile à maintenir…

Propriété (Attribut, variable d'instance ou donnée membre) Donnée mémorisée à l'intérieur d'un *objet* afin de définir son *état*. Ces données peuvent être des valeurs (nombre ou texte) ou des références à d'autres objets. Notez aussi que le Pascal Objet permet de simuler l'existence de propriétés virtuelles grâce à la déclaration « `property` ».

Redéfinition Conception d'une nouvelle *méthode* portant le même nom qu'une *méthode* existante au sein d'une même *classe* (ou d'un même niveau de visibilité). Les *méthodes* doivent posséder des paramètres de type ou de noms différents. La *redéfinition* utilise le mot clef « `overload ;` ». Dans ce manuel nous avons introduit un terme différent, « *redéfinition* », là où d'autres ouvrages utilisent le terme de « *surcharge* ». Nous avons fait ceci dans le but d'éviter les confusions car la « *surcharge* » signifie aussi autre chose (voir ce terme).

Sous-classe Qualifie une *classe* construite à partir d'une autre *classe* grâce à un *héritage*.

Spécialisation Démarche de l'analyste objet inverse de la *généralisation*. La nouvelle classe va permettre de définir l'état et le comportement d'objets plus spécifiques et spécialisés. Par exemple l'analyste a déjà défini une classe de « périphérique » et une classe de « fichier ». L'analyse l'amène à concevoir aussi des « fichiers séquentiels » voire des « fichiers séquentiels indexés » — C'est la spécialisation. Voir aussi « *Généralisation* ».

Superclasse Classe ancêtre — Voir Ancêtre.

Surcharge Le remplacement d'une *méthode* d'une classe *ancêtre* par une *méthode* de même nom (ou la redéclaration d'une *variable d'instance*) dans une *sous-classe* après un *héritage*.

Utilisation (relation entre objets) Relation qui existe entre deux objets lorsqu'un de ces objets peut voir l'autre (sans en être le propriétaire — Voir *contenance*). L'objet qui en voir un autre peut alors lui envoyer des *messages*. Un objet peut en utiliser un autre parce qu'il a été passé comme paramètre à une de ses méthodes ou parce que cet autre objet et global (et est donc visible par tous les objets de l'application).

Unité Une *unité* est un module de programmation ou un module de programme écrit en *Pascal*. Un programme *Pascal* est construit à partir d'un programme principal et éventuellement d'une ou plusieurs unités.

Variable de classe Variable commune à tous les objets d'une même *classe*. Le *Pascal Objet* de *Delphi* ne permet pas de déclarer de *variable de classe*, mais celles-ci peuvent être facilement simulées en déclarant une variable dans la partie « `implementation` » d'une *unité*.

Variable d'instance Variable existant au sein d'un *objet* (d'une *instance*).

Virtuel En français « *virtuel* » signifie « *qui semble exister bien qu'il n'existe pas* ». Un exemple est l'image que renvoie un miroir, il semble qu'il y ait un objet *virtuel* derrière la paroi du miroir. Voir *abstrait*, *méthode virtuelle* et *méthode virtuelle pure*.

14. INDEX