# Guide Pascal et Delphi

par Frédéric Beaulieu

Date de publication : 10/04/2000

Dernière mise à jour : 01/04/2008

Bienvenue dans le mille et unième guide pour Pascal et Delphi qui existe en ce bas monde ! La profusion n'étant pas synonyme d'exhaustivité, et ne pouvant satisfaire tout le monde, ce guide se veut d'être un parcours complet depuis presque RIEN (en Pascal, j'entends, voir les pré-requis pour le reste) vers les connaissances nécessaires à un programmeur Delphi (à peu près) digne de ce nom (modestie oblige). Le principal objectif de ce guide est ainsi de vous apprendre à créer des logiciels.

I - Introduction.......... 7
II - Pré-requis.......... 8
III - Pascal ? Delphi ?.......... 9
III-A - Pascal.......... 9
III-B - Delphi.......... 10
III-C - Un peu de vocabulaire.......... 10
IV - Premiers pas, découverte de Delphi.......... 12
IV-A - Installation et lancement de Delphi.......... 12
IV-B - Premier contact avec Delphi.......... 13
IV-C - Utilisateur, Programmeur et les deux à la fois.......... 14
IV-D - Notions de base de la programmation sous Delphi.......... 15
IV-D-1 - La notion de projet.......... 15
IV-D-2 - La notion de composant.......... 17
IV-D-3 - Propriétés et événements.......... 18
IV-E - Premier aperçu de l'interface de Delphi.......... 18
IV-E-1 - La barre de menus.......... 18
IV-E-1-a - Menu Fichier.......... 19
IV-E-1-b - Autres menus.......... 20
IV-E-2 - La barre d'outils.......... 20
IV-E-3 - La palette des composants.......... 20
IV-E-4 - L'inspecteur d'objets.......... 23
IV-E-5 - L'éditeur de code.......... 25
IV-E-6 - Conclusion.......... 25
V - Préliminaires.......... 26
V-A - Données simples dans un programme Pascal.......... 26
V-A-1 - Nombres.......... 26
V-A-1-a - Nombres entiers.......... 26
V-A-1-b - Nombres à virgules.......... 28
V-A-1-c - Opérations sur les nombres.......... 29
V-A-2 - Caractères et chaînes de caractères.......... 30
V-A-3 - Booléens.......... 31
V-A-4 - Types énumérés.......... 33
V-A-5 - Récapitulatif.......... 33
V-B - Utilisations des types simples dans l'inspecteur d'objets.......... 34
V-C - Structure d'un programme Pascal.......... 36
V-C-1 - Structure d'un fichier projet.......... 36
V-C-2 - Structure d'une unité.......... 38
V-D - Constantes et Variables.......... 41
V-D-1 - Préliminaire : les identificateurs.......... 41
V-D-2 - Constantes.......... 42
V-D-3 - Variables.......... 43
V-E - Conclusion.......... 43
VI - Procédures et Fonctions.......... 45
VI-A - Procédures.......... 45
VI-B - Fonctions.......... 48
VI-C - Premières instructions en Pascal.......... 48
VI-C-1 - Affectations.......... 48
VI-C-2 - Appels de procédures et de fonctions.......... 49
VI-D - Manipulations.......... 52
VI-E - Conclusion.......... 58
VII - Types de données avancés de Pascal Objet.......... 59
VII-A - Création de nouveaux types.......... 59
VII-B - Type ordinaux.......... 60
VII-C - Type intervalle.......... 60
VII-D - Compléments sur les types énumérés.......... 61
VII-E - Type ensemble.......... 62
VII-F - Tableaux.......... 64
VII-F-1 - Tableaux à une seule dimension.......... 64


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

VII-F-2 - Tableaux à plusieurs dimensions........ 65
VII-F-3 - Notions avancées sur les tableaux........ 66
VII-F-4 - Tableaux de taille dynamique........ 67
VII-G - Enregistrements........ 69
VII-G-1 - Vue d'ensemble sur les enregistrements........ 69
VII-G-2 - Manipulation avancée des enregistrements........ 72
VII-H - Types et paramètres de procédures et fonctions........ 73
VII-I - Conclusion et retour sur terre........ 73
VIII - Structures de programmation en Pascal........ 74
VIII-A - Structures conditionnelles........ 74
VIII-A-1 - Blocs 'if'........ 74
VIII-A-2 - Blocs 'case'........ 79
VIII.B - Structures itératives........ 81
VIII-B-1 - Blocs 'for'........ 82
VIII-B-2 - Blocs 'while'........ 84
VIII-B-3 - Blocs 'repeat'........ 89
VIII-B-4 - Contrôle avancé des boucles........ 91
IX - Manipulation des composants........ 93
IX-A - Introduction........ 93
IX-B - Aperçu de la structure interne d'un composant........ 94
IX-C - Manipulation des propriétés d'un composant........ 95
IX-C-1 - Utilisation de Delphi........ 95
IX-C-2 - Utilisation du code source........ 98
IX-D - Manipulation des méthodes d'un composant........ 102
IX-E - Evénements........ 105
IX-E-1 - Style de programmation événementiel........ 105
IX-E-2 - Utilisation des événements........ 106
IX-F - Composants invisibles........ 108
IX-G - Imbrication des composants........ 109
IX-H - Types de propriétés évolués........ 111
IX-H-1 - Propriétés de type objet........ 111
IX-H-2 - Propriétés de type tableau, propriété par défaut........ 111
IX-H-3 - Propriétés de type référence........ 113
IX-I - Conclusion........ 113
X - Découverte des composants les plus utilisés - 1ère partie........ 114
X-A - La fiche : Composant "Form"........ 114
X-A-1 - Manipulation Guidée........ 120
X-B - Référence des composants........ 122
X-B-1 - Composant "MainMenu"........ 122
X-B-2 - Composant "TPopupMenu"........ 123
X-B-3 - Composant "Label"........ 124
X-B-4 - Composant "Edit"........ 127
X-B-5 - Composant "Memo"........ 130
X-B-6 - Composant "Button"........ 132
X-B-7 - Composant "CheckBox"........ 136
X-B-7-a - Manipulation guidée........ 137
X-B-8 - Composant "RadioButton"........ 140
X-B-9 - Composant "ListBox"........ 141
X-B-10 - Composant "ComboBox"........ 143
X-B-11 - Composant "GroupBox"........ 145
X-B-12 - Composant "Panel"........ 146
X-B-13 - Composant "Bevel"........ 148
X-B-14 - Composant "ImageList"........ 150
X-C - Mini-projet........ 150
XI - Pointeurs........ 151
XI-A - Présentation de la notion de pointeur........ 151
XI-A-1 - nil........ 152
XI-A-2 - Eléments pointés par un pointeur........ 153



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

XI-A-3 - Gestion de l'élément pointé par un pointeur.......... 154
XI-A-4 - Utilisation de l'élément pointé par un pointeur.......... 156
XI-B - Etude de quelques types de pointeurs.......... 156
XI-B-1 - Pointeurs de tableaux.......... 156
XI-B-2 - Pointeurs d'enregistrements (record).......... 157
XI-C - Transtypage.......... 159
XI-D - Conclusion.......... 160
XII - Objets.......... 161
XII-A - Définitions des notions d'objet et de classe.......... 161
XII-A-1 - Objet.......... 161
XII-A-2 - Classe.......... 162
XII-B - Utilisation des objets.......... 162
XII-B-1 - Construction et destruction.......... 162
XII-B-2 - Manipulation des objets.......... 163
XII-B-2-a - Exercice résolu.......... 165
XII-C - Notions avancées sur les classes.......... 168
XII-C-1 - Hiérarchie des classes.......... 168
XII-C-1-a - Concept général.......... 168
XII-C-1-b - Classes descendantes de TForm.......... 171
XII-C-2 - Ajout d'éléments dans une classe.......... 172
XII-C-2-a - Sections privées et publiques d'une classe.......... 173
XII-C-2-b - Ajout de méthodes et de variables.......... 173
XII-C-3 - Paramètres de type objet.......... 176
XII-C-3-a - Explications générales.......... 176
XII-C-3-b - Envoi de paramètres de classes différentes.......... 177
XIII - Utilisation des fichiers.......... 178
XIII-A - Introduction : Différents types de fichiers.......... 178
XIII-A-1 - Fichiers texte.......... 178
XIII-A-2 - Fichiers séquentiels.......... 178
XIII-A-3 - Fichiers binaires.......... 178
XIII-B - Manipulation des fichiers texte.......... 179
XIII-B-1 - Ouverture et fermeture d'un fichier texte.......... 179
XIII-B-2 - Lecture depuis un fichier texte.......... 181
XIII-B-3 - Ecriture dans un fichier texte.......... 182
XIII-B-4 - Utilisation des fichiers texte par les composants.......... 183
XIII-C - Manipulation des fichiers séquentiels.......... 184
XIII-C-1 - Présentation.......... 184
XIII-C-2 - Ouverture et fermeture d'un fichier séquentiel.......... 185
XIII-C-3 - Lecture et écriture depuis un fichier séquentiel.......... 186
XIII-D - Fichiers binaires.......... 189
XIII-D-1 - Présentation.......... 190
XIII-D-2 - Capture des erreurs d'entrée-sortie.......... 190
XIII-D-3 - Lecture et écriture dans un fichier binaire.......... 191
XIII-D-3-a - Paramètres variables.......... 192
XIII-D-3-b - Paramètres non typés.......... 193
XIII-D-3-c - Description des deux procédures de lecture et d'écriture.......... 193
XIII-D-4 - Lecture et écriture des différents types de données.......... 193
XIII-D-4-a - Compléments sur les types de données.......... 193
XIII-D-4-b - Préliminaires.......... 195
XIII-D-4-c - Types simples : entiers, réels et booléens.......... 195
XIII-D-4-d - Types énumérés.......... 200
XIII-D-4-e - Types chaîne de caractères.......... 202
XIII-D-4-f - Autres types.......... 205
XIII-D-4-g - Structures avancées dans des fichiers binaires.......... 205
XIV - Découverte des composants les plus utilisés - 2ème partie.......... 206
XV - Manipulation de types abstraits de données.......... 207
XV-A - Introduction.......... 207
XV-B - Piles.......... 208


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

XV-B-1 - Présentation....208
XV-B-2 - Une définition plus formelle....208
XV-B-3 - Implémentation d'une pile avec un tableau....209
XV-B-4 - Compléments sur les pointeurs : chaînage et gestion de la mémoire....214
XV-B-5 - Implémentation par une liste chaînée....216
XV-C - Files....220
XV-C-1 - Présentation et définition....220
XV-C-2 - Implémentation d'une file par une liste chaînée....221
XV-D - Listes....226
XV-D-1 - Présentation et définition....226
XV-D-2 - Notions de tri....228
XV-D-3 - Implémentation par un tableau....230
XV-D-4 - Implémentation par une liste chaînée....234
XV-D-5 - Implémentation permettant différents tris....242
XV-D-5-a - Présentation....242
XV-D-5-b - Paramètres fonctionnels....243
XV-D-5-c - Implémentation du tri à la demande....244
XV-E - Arbres....247
XV-F - Arbres binaires....250
XV-G - Mini-projet : calculatrice....254
XVI - Programmation à l'aide d'objets....255
XVI-A - Introduction....255
XVI-B - Concepts généraux....255
XVI-B-1 - De la programmation traditionnelle à la programmation objet....255
XVI-B-2 - La programmation (orientée ?) objet....255
XVI-B-3 - Classes....256
XVI-B-4 - Objets....260
XVI-B-5 - Fonctionnement par envoi de messages....260
XVI-B-6 - Constructeur et Destructeur....261
XVI-C - Bases de la programmation objet sous Delphi....261
XVI-C-1 - Préliminaire : les différentes versions de Delphi....261
XVI-C-2 - Définition de classes....262
XVI-C-3 - Déclaration et utilisation d'objets....264
XVI-C-4 - Utilisation d'un constructeur et d'un destructeur, notions sur l'héritage....271
XVI-C-5 - Visibilité des membres d'une classe....274
XVI-C-6 - Propriétés....277
XVI-C-6-a - Propriétés simples....277
XVI-C-6-b - Propriétés tableaux....282
XVI-C-6-c - Propriété tableau par défaut....285
XVI-C-7 - Mini-projet n°5 : Tableaux associatifs....287
XVI-D - Notions avancées de programmation objet....287
XVI-D-1 - Retour sur l'héritage....287
XVI-D-2 - Polymorphisme....288
XVI-D-3 - Surcharge et redéfinition des méthodes d'une classe....290
XVI-D-4 - Méthodes abstraites....292
XVI-D-5 - Méthodes de classe....296
XVI-E - Conclusion....297
XVII - Liaison DDE....298
XVII-A - Présentation....298
XVII-A-1 - Introduction....298
XVII-A-2 - Composants....299
XVII-A-3 - Utilisation....300
XVII-B - Partie Serveur....300
XVII-C - Partie Client....300
XVII-D - Cas particulier : Serveur Non-Delphi....302
XVII-D-1 - Serveur Non-Delphi....302
XVII-D-2 - Client Non-Delphi....303
XVII-E - Conclusion....304


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

XVIII - DLL.......305
XVIII-A - Introduction.......305
XVIII-B - Ecriture des DLLs & Utilisation.......305
XVIII-B-1 - DLL de fonction.......305
XVIII-B-2 - Théorie & Subtilité.......308
XVIII-B-3 - DLL de Classe.......309
XVIII-B-4 - DLL de Composant.......312
XVIII-B-5 - DLL & Chaîne de caractère.......314
XVIII-C - Chargement statique/dynamique.......316
XVIII-D - DLL et Autres Langages.......318
XVIII-E - Conclusion.......319
XIX - Gestion des exceptions.......320
XIX-A - Introduction.......320
XIX-B - Try..Finally.......321
XIX-C - Try..Except.......322
XIX-C-1 - Grammaire.......322
XIX-C-2 - Listes non exhaustive de classe d'exception.......323
XIX-D - Exception personnalisée.......324
XIX-E - Raise.......325
XIX-F - Conclusion.......326
XX - TFileStream.......327
XX-A - Introduction.......327
XX-B - Lecture.......327
XX-C - Ecriture.......329
XX-D - Conclusion.......330


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# I - Introduction

Voici des sujets qui ne seront (hélas) PAS abordés (car non maîtrisés par votre serviteur, à savoir moi) :

- Bases de données
- Programmation pour Internet
- Programmation « multi-thread »

Si vous voulez de la documentation là-dessus, il faudra aller voir ailleurs, désolé.
Ce guide est conçu comme un vrai cours : il est découpé en chapitres comportant chacun du cours, des exercices et les corrigés détaillés. L'ensemble est prévu pour être suivi dans l'ordre des chapitres, mais rien ne vous oblige à respecter cet ordre si vous maîtrisez déjà tout ou partie des notions vues dans un chapitre particulier.
Des propositions de mini-projets (non résolus entièrement) sont également présentes dans le guide. Le but des mini-projets est de créer un logiciel permettant de résoudre un problème. Le principe est ici très différent des exercices résolus : vous réalisez un mini-projet à partir d'un cahier des charges. L'objectif est atteint lorsque le logiciel créé parvient à traiter le problème décrit dans le cahier des charges, et ceci quelle que soit cette manière. Pour l'instant, si vous voulez être corrigé ou demander des conseils, **contactez-moi**.
La version de Delphi qui a été utilisée pour réaliser les exemples, les captures d'écran et les corrigés est la version 5 anglaise. Si vous avez la version française, c'est mieux pour vous, sinon, une autre version de Delphi peut convenir mais certaines commandes auront des noms légèrement différents ou seront tout simplement inaccessibles (voir les **pré-requis** pour plus d'informations).
Voici quelques styles de présentation destinés à vous donner des repères dans le guide.

***Encadré :***
*Dans ce genre d'encadré seront présentées des informations plus techniques sous la mention 'Approfondissement'. Des révisions rapides seront également proposées sur différents sujets utilisés pendant le guide.*

```
Ce genre de paragraphe contiendra du texte écrit en langage Pascal. Vous pourrez effectuer un
copier-coller pour prendre le code et le placer sous Delphi sans avoir à le taper vous-même
(ce qui ne doit pas vous dispenser de le faire une fois de temps en temps).
```

Les parties entre crochets *[]* sont optionnelles, les textes écrits en *italique* sont des raccourcis pour désigner d'autres structures. Il faudra inclure ces structures et non le texte en italique.
L'un des principes de base de ce guide est de ne pas prendre l'utilisateur de haut. Si vous avez l'impression d'être pris de haut pendant votre lecture, n'hésitez pas à **me contacter** pour mettre les choses au point. De cette manière, chacun profitera des corrections effectuées dans le guide (je ne garantis nulle part que tout ce qui est écrit ici ne comporte pas d'erreur, les suggestions de corrections seront donc les bienvenues).
Enfin, car je commence à m'éterniser, ce guide est conçu pour avancer lentement mais sûrement. Certains d'entre vous trouverons au cours du guide que je ne le fais pas avancer assez vite. Que ceux d'entre vous qui pense qu'aller vite peut mener quelque part se rassurent : il existe beaucoup d'autres sites sur internet qui combleront leur attente et peut-être même plus. Je me bornerai pour ce guide à avancer à pas mesurés pour ne lâcher personne en cours de route, alors, bon courage et bonne lecture !


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## II - Pré-requis

Certains diront : "Vous avez dit dans l'introduction qu'on pouvait partir de rien !". Eh bien, au niveau connaissances sur la programmation, c'est vrai. Par contre, il vous faut avant de commencer ce guide (la liste paraît longue, mais ne vous affolez surtout pas) :

- Un ordinateur avec les logiciels suivants :
    - Microsoft Windows 95 ou supérieur (98, 98 SE, Millenium, XP...) *ou* Microsoft Windows NT 4 ou Windows 2000 (non testé).
    - Borland/Inprise Delphi version 2 ou supérieure (3, 4, 5, ...) pour suivre le guide.
      La version 5 est nécessaire pour pouvoir utiliser les exemples téléchargeables ici.
      La langue du logiciel est sans importance, mais le français est recommandé pour débuter.

  Il va sans dire que vous devez savoir utiliser au moins un peu ces logiciels.
- Des connaissances minimales en informatique générale. Vous devez savoir ce que désigne chacune des expressions ci-dessous :
    - Windows
    - Fenêtre, boite de dialogue
    - Logiciel, Application
    - Menus, barre d'outils, boutons, cases à cocher, info-bulle, liste, liste déroulante (« combo »), icône
- Vous devez savoir utiliser un ordinateur (clavier : touches Ctrl, Alt, Shift ; souris : clic, un double clic, un clic droit), Windows, l'Explorateur.
- Un sens de la logique suffisamment développé (la programmation fait largement appel à la logique).
  Quelques rappels seront cependant faits au cours du guide.
- Quelques connaissances en anglais (au moins quelques mots).
  Chaque mot anglais présent dans le guide sera traduit en français au moins une fois. Il vous incombe de comprendre ces mots les fois suivantes.
- De la patience, du courage, de la volonté, un rien de persévérance et du self-control quand vos premiers "bugs" (définition du premier dictionnaire venu : « erreur de rédaction d'un logiciel entraînant des dysfonctionnements ») apparaîtront (vous noterez que je n'ai pas dit « si » mais « quand » : des bugs finissent toujours par faire leur apparition).
- Du temps...

Bon courage !


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# III - Pascal ? Delphi ?

L'objectif de ce guide est de créer des logiciels et non plus de simplement les utiliser. Un moyen pour créer un logiciel est d'utiliser un langage informatique, traduit ensuite par l'ordinateur pour en faire un logiciel. C'est le moyen exploré dans ce guide : l'utilisation d'un langage de programmation informatique. Le langage étudié ici, c'est « Pascal ». C'est un langage qui permet d'écrire des logiciels. « Delphi », c'est le logiciel que vous allez employer pour écrire des textes dans ce langage (entre autres).

## III-A - Pascal

Pascal, c'est ainsi le nom du langage que vous allez découvrir dans ce guide. Comme tout langage, il a ses règles, sa syntaxe, ses structures, ses limites, ses exceptions (mais ce ne sont pas celles qu'on croit, nous le verrons plus loin). Pascal est utilisé pour communiquer avec l'ordinateur : c'est à vous de vous adapter en lui parlant dans un langage qu'il puisse comprendre.
Attention toutefois, en tant que programmeur, votre interlocuteur est l'ordinateur, et non pas le ou les utilisateurs des logiciels que vous écrivez. Il s'agit là d'une distinction subtile qu'il faut faire tout de suite : au moment où l'utilisateur fera ce qu'on attend de lui, à savoir utiliser un de vos logiciels, vous ne serez pas présent (ni pour lui expliquer ce qu'il faut faire, ni pour corriger les problèmes posés). Il faudra donc avoir dit précisément à l'ordinateur ce qu'il convient de faire.
En effet, l'ordinateur n'acceptera pas de fautes de langage, mais il vous obéira au doigt et à l'#il. Mais il y a une contrepartie : si vous lui dites de faire des âneries dans un langage des plus corrects (au niveau syntactique), il le fera, et peu lui importe les conséquences, il n'y aura que vous et vos utilisateurs que ça dérangera !

***Approfondissement (facultatif):***
*Pascal n'est pas unique : il existe d'autres langages de programmation informatique. Vous avez peut-être entendu d'autres noms de langages (informatiques), parmi lesquels l'assembleur, le Basic (et le Visual Basic), le C (et son grand frère le C++), le Java, le Lisp, Le Perl, etc. Ces langages, de principes, de domaines d'utilisations et de puissances variés, ne peuvent pas tous être compris par l'ordinateur, qui comprend un unique langage : le langage machine, directement exécuté par le microprocesseur, mais absolument imbuvable.*
*Pourquoi alors parler du langage machine ? Parce que c'est dans ce langage que Pascal devra être traduit avant d'être exécuté. Cette traduction sera effectuée par Delphi de façon très pratique et presque invisible pour vous et est nommée compilation. Le logiciel utilisé par Delphi et qui est chargé de cette traduction est nommé compilateur. Le résultat d'une compilation sera pour nous un logiciel, qui se présentera sous la forme d'un fichier avec l'extension .EXE*

Pascal est un langage dit de type « impératif ». N'ayez pas peur de ce terme barbare qui veut simplement dire que les logiciels que vous écrirez le seront au moyen de suites d'instructions (regroupées dans des structures que nous étudierons) qui seront exécutées une par une par l'ordinateur (dans un ordre qu'il sera possible de contrôler) : vous direz pas à pas à l'ordinateur ce qu'il convient de faire.
Voici un premier texte écrit en langage Pascal : examinez-le à loisir sans essayer de comprendre la syntaxe ni la signification exacte de chaque partie (il fait ce qu'il semble suggérer, à savoir l'addition de deux nombres) :

```
function Addition (Nombre1, Nombre2: Integer): Integer;
begin
  Result := Nombre1 + Nombre2;
end;
```

Pascal est un langage constitué de mots, organisés en structures (pas vraiment des phrases). Ces mots sont répartis d'une part en mots réservés, c'est-à-dire en mots dont le sens et l'usage sont déterminés une fois pour toutes. Dans le morceau de texte ci-dessus, les mots réservés sont en gras (procedure, begin, end). Ces mots sont à la base du langage.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Le reste des mots utilisés est séparé en deux groupes : les mots fournis par le langage, tels « Integer » ou « Result » et les autres, créés par le programmeur (à savoir vous) suivant des règles que nous apprendrons. Ne vous inquiétez pas : ces deux derniers groupes ne sont pas discernables à l'#il nu par vous pour l'instant, mais ils le deviendront au fil du guide.
Au niveau de la présentation, le langage Pascal ne fait pas de distinction entre une majuscule et une minuscule. Ainsi chaque mot du langage pourra être écrit avec ou sans majuscules. Vous remarquez également que des espaces et des retours à la ligne judicieusement placés permettent d'avoir quelque chose de plus agréable que ce qui suit (et pourtant, l'extrait ci-dessous serait accepté et ferait exactement la même chose que le premier extrait) :

```
procedure Addition(Nombre1,Nombre2:Integer):Integer;begin
Result:=Nombre1+Nombre2;end;
```

Le langage Pascal va nous servir à écrire des programmes. Un programme est un texte plus ou moins long, entièrement écrit en Pascal. On peut y inclure beaucoup de choses, et entre autres des instructions qui seront exécutées par l'ordinateur. Le premier extrait de code (ou le deuxième, mais le premier est nettement préférable) est un morceau de programme. Ces programmes seront utilisés comme matière première pour la création des logiciels.

## III-B - Delphi

Delphi est le nom d'un logiciel actuellement largement employé pour créer des logiciels. Delphi permet d'utiliser le langage Pascal. Il faut bien comprendre que Pascal et Delphi NE SONT PAS une seule et même chose : Pascal est un langage informatique, Delphi est un logiciel destiné à créer des logiciels avec ce langage. Delphi n'est qu'un enrobage, une enveloppe de confort autour de Pascal, c'est-à-dire qu'il simplifie de nombreuses tâches liées à la programmation en langage Pascal. Un autre fait qu'il faut noter est que Delphi est destiné à écrire des programmes fonctionnant exclusivement sous Windows. Nous ne nous intéresseront donc dans ce guide qu'à Windows (et seulement à Windows 95 ou ultérieur).

***Note aux programmeurs en assembleur et aux utilisateurs de Turbo Pascal pour DOS ou Windows :***
*Delphi va un peu plus loin que le Pascal utilisé dans Turbo Pascal ou Borland Pascal en inventant une nouvelle syntaxe, plus souple, ouverte et puissante : le langage employé dans Delphi est en fait le Pascal Objet. Si vous connaissez déjà bien le Pascal, le Pascal Objet ne vous apportera que peu de surprises, et souvent de très bonnes. Sinon, le Pascal Objet est tout ce que vous avez besoin de connaître car seule la nouvelle syntaxe est reconnue par Delphi.*
*Delphi permet en outre l'insertion de code écrit en assembleur 32 bits, mais il faut limiter cette possibilité à des cas très restreints.*

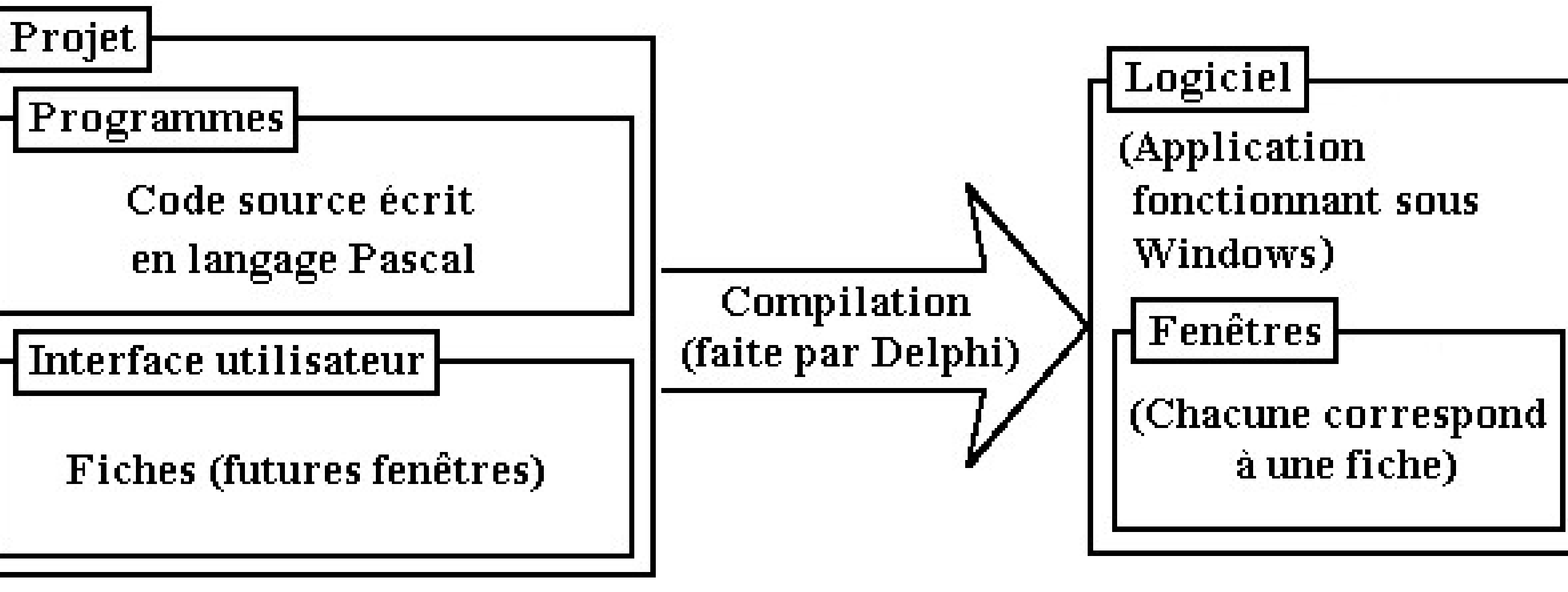
Comme beaucoup de logiciels, Delphi existe en plusieurs versions. Actuellement des versions numérotées de 1 à 5 existent. Ces versions successives du logiciel ont vu de nombreuses améliorations, tant cosmétiques, qu'au niveau du langage. La version la plus élevée est la plus intéressante, car permet toujours plus de choses. Certains aspects du langage vus dans les derniers chapitres sont réservés à Delphi 5 ou ultérieur (ce sera signalé).
Nous aurons, tout au long du guide, de nombreuses occasions d'utiliser Delphi, ce qui vous permettra de vous habituer petit à petit à ce fantastique logiciel.

## III-C - Un peu de vocabulaire

L'informatique, en tant que science, a son propre jargon, qu'il est nécessaire de connaître. Les quelques termes qui suivent sont des incontournables. Lorsque vous les rencontrerez, vous devrez savoir ce qu'ils signifient. Les définitions ci-dessous ne sont valables qu'à l'intérieur du guide (et sont souvent utilisées en dehors !) :
« **Programme** » : texte écrit dans un langage informatique, comportant dans notre cas des instructions structurées (organisées en structures). Il est destiné à être « converti » par Delphi en un logiciel utilisable sous Windows.
« **Développer en Delphi** » : écrire des programmes en utilisant le langage Pascal. Par abus, on confond le langage (Pascal) et le logiciel (Delphi) qui le gère, et on parle de développer, plutôt que de programmer.
« **Application** » : Logiciel fonctionnant sous Windows.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

« **Projet** » : c'est la base d'une application. Sous Delphi, pour créer une application, on constitue d'abord un projet, constitué de divers morceaux (notion importante vue plus loin).
« **Code** », « **Code Pascal** », « **Code source** » : morceau de programme, texte d'un programme écrit en Pascal.
« **Interface (utilisateur)** » : la partie d'un logiciel qui est visible par l'utilisateur, à l'opposé du code source, invisible à l'utilisateur.
« **Fiche** » : fenêtre à l'état non compilé. Les fiches sont les alter ego sous Delphi des fenêtres sous Windows.
Le schémas suivant tente de reprendre les termes mentionnés ci-dessus :





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# IV - Premiers pas, découverte de Delphi

## IV-A - Installation et lancement de Delphi

***Révision éclair : Répertoires et Fichiers***
*Votre ordinateur possède au moins un disque dur. Ce disque est souvent accessible par une lettre de lecteur (présente dans le poste de travail, dans l'explorateur) qui est souvent C (le disque dur est alors nommé C:).*
*Le lecteur C: est, comme tous les supports informatiques classiques (Disque durs, disquettes, CD-Roms, ZIP, DVD-Rom) organisé à la manière d'un gigantesque classeur. Il contient une racine, appelée aussi répertoire racine, qui obéit à la même règle que les répertoires (appelés aussi dossiers). Cette règle est : elle (la racine) ou il (le répertoire) peut contenir des répertoires et des fichiers.*
*Sous Windows 95 ou ultérieur, les fichiers et les répertoires sont nommés avec beaucoup moins de rigueur qu'avant (8 caractères, un point et une extension de 3 caractères définissant classiquement le type de fichier, les noms de répertoires étant généralement privés d'extension). Les noms peuvent maintenant comporter plus de 8 caractères, et les caractères autorisés augmentent en nombre, avec les espaces, les lettres accentués, les majuscules et minuscules. Une règle qui reste d'actualité est l'extension : elle suit toujours le nom du fichier, précédée d'un point, et comporte trois caractères (quoique cette limitation à trois caractères n'est plus obligatoire mais recommandée). Cette extension indique toujours le type de fichier.*
*Un fichier peut être donné directement ou en fonction du répertoire dans lequel il se trouve. Prenons un fichier « exemple 1.doc ». S'il est situé à la racine du disque C:, son nom complet sera : « C:\exemple 1.doc ». S'il est dans le répertoire « Mes Documents », lui-même à la racine du disque, le nom complet du fichier devient alors « C:\Mes Documents \exemple 1.doc «.*
*Vous connaissez peut-être l'invite MS-Dos, pour l'avoir connue avant Windows 95 ou seulement après. Dans les deux cas, tout nom de fichier comportant des caractères interdits avant Windows 95 (tels les espaces, les caractères accentués) devra être donné entre guillemets : " ".*
*(Il n'est pas question ici de vous apprendre les commandes DOS, pour cela, consultez un manuel ou l'aide de Windows)*

Si vous utilisez Delphi à la fac ou tout autre lieu où Delphi est directement accessible, vous pouvez passer ce paragraphe puisque Delphi est probablement directement accessible sur les ordinateurs. Vous ne pourrez et ne devez pas en général installer de logiciels sur ces ordinateurs pour éviter les risques d'infection par d'éventuels virus. Sinon, selon la version de Delphi que vous possédez, le fichier à lancer change en général assez peu, généralement, c'est setup.exe ou install.exe . L'installation vous propose un certain nombre d'options, comme l'emplacement de Delphi sur votre disque, ou les composantes de Delphi à installer. Un bon choix pour le répertoire est quelque chose de court n'utilisant pas de noms longs (par exemple : un répertoire BORLAND ou INPRISE à la racine de votre disque dur, puis un répertoire DELPHI où vous installez Delphi), parce que vous aurez un jour besoin (quoique ce besoin tende à disparaître avec les versions récentes de Delphi) de vous balader dans le répertoire en mode DOS, et que taper des noms longs sous DOS n'est pas un sport très agréable à pratiquer (imaginez vous simplement taper la commande :

```
cd "program files\borland\delphi 6\source\rtl\win"
```

Avec les choix que je vous ai donné, vous taperez plutôt :

```
cd borland\delphi\source\rtl\win
```

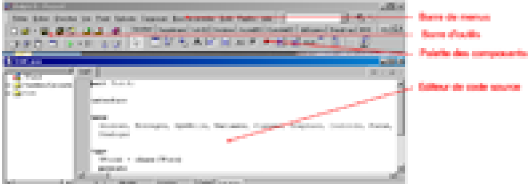
Ce qui est quand même plus court, et vous comprendrez rapidement qu'un minimum de paresse (mais n'allez pas me faire dire plus) en informatique ne fera de mal à personne et surtout pas à vous.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

D'autre part, parmi les options d'installation, assurez-vous bien d'inclure toute l'aide possible (l'aide de Delphi et celle des API Windows, même si cela ne vous dit encore rien), l'éditeur d'images (si cette option est proposée), et le source de la VCL (disponibles dans toutes les version sauf les versions standard. Ces éléments seront décrits et utilisés plus tard. Ils sont d'une importance non négligeable et une source irremplaçable pour les plus curieux d'entre vous).
L'installation requiert pas mal de place suivant les versions (jusqu'à 200-300 Mo) mais est justifiée par la pléthore d'outils qui sont mis à votre disposition, et dont vous n'apprécierez la puissance qu'à l'usage.
L'installation terminée, des raccourcis ont été créés pour vous dans le menu Démarrer de Windows. Si Delphi a été installé par quelqu'un d'autre que vous, parcourez le menu Démarrer à la recherche des icônes de Delphi, et lancez le logiciel (raccourci nommé « Borland Delphi x » ou « Borland Developer Studio x » où x désigne le numéro de version.

## IV-B - Premier contact avec Delphi

Une fois lancé, le logiciel que vous allez utiliser tout au long de votre initiation apparaît : Delphi. La capture d'écran disponible ci-dessous (cliquez sur la petite image pour afficher la grande) montre Delphi 5 (version anglaise) tel qu'il est présenté après un démarrage normal (votre version peut seulement ressembler à la capture d'écran, avec des différences mineures, notamment dans les menus, les icônes de la barre d'outils et la disposition des fenêtres) :
(Note : si vous n'avez pas du tout cette présentation, c'est que quelqu'un s'est plu à tout déplacer. Il est également possible qu'un programme précédemment créé s'ouvre à nouveau, dans ces cas, cliquez sur Fichier, puis Nouveau Projet ou Nouvelle Application)

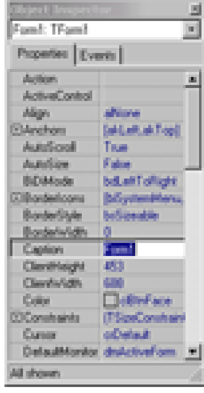


*(Cette capture montre Delphi 5)*

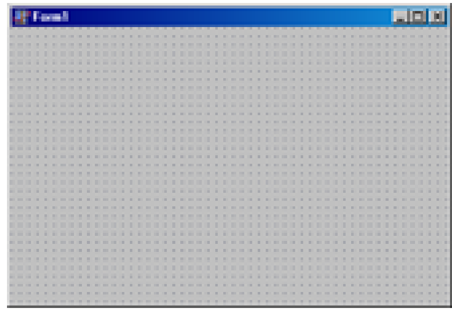
Le texte rouge indique les parties que vous devez pouvoir repérer dés à présent, sans avoir à connaître encore leur utilité :

- Barre de menus
- Barre d'outils
- Palette des composants
- Editeur de code source (s'il n'est pas visible, ce n'est pas grave pour l'instant, il le sera bientôt)

D'autres fenêtres flottantes (que vous pouvez déplacer) peuvent être présentes sur votre écran. Entre autres l'inspecteur d'objet (à gauche) et une fiche (fenêtre) vide (à droite) comme ci-dessous :




Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



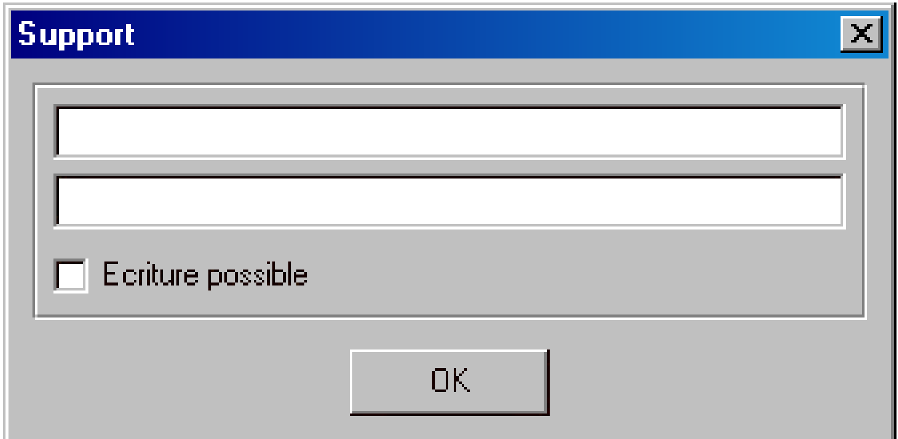
Rappelez-vous que vous allez créer des logiciels pour Windows. Ils pourront donc contenir les effets visuels présents dans les autres applications fonctionnant sous Windows, tels les fenêtres, les cases à cocher, les boutons, les menus, les barres d'outils, les infos-bulles : tout ce qui fait d'une application une application Windows.
Delphi permet de créer une application (un logiciel, donc) à la fois, mais permet de créer simultanément les deux aspects interdépendants d'une application :

1 le coté visible (l'interface, pour les utilisateurs du logiciel) (repensez aux fiches du schémas dans le § III.C.
2 le coté invisible (là ou se situe l'intelligence du logiciel) constitué en grande partie de programmes (cf. § III.C)

Pour créer un logiciel en utilisant Delphi (on dira plutôt *sous* Delphi), il faut créer ces deux parties en même temps (c'est-à-dire que les deux parties visibles et invisibles, loin d'être indépendantes, sont vraiment liées dans le sens où la partie visible n'est qu'une façade, un masque sous lequel la partie invisible fait tout le travail. C'est le schéma de l'iceberg : l'utilisateur ne devra voir que la partie au dessus de l'eau, il n'aura pas à se soucier et devra ignorer ce qu'il y a sous l'eau, domaine réservé au programmeur (on dit également développeur) que vous êtes désormais. La partie immergée, bien qu'invisible, prend plus de place, contrôle ce qui est visible au dessus de l'eau et permet à l'iceberg de flotter sans couler !

## IV-C - Utilisateur, Programmeur et les deux à la fois

En tant que programmeur, vous êtes obligatoirement et d'abord un utilisateur : c'est très certainement vous qui allez tester vos logiciels (avant de les lâcher dans la nature, si telle est votre intention, et ce devrait toujours être une de vos intentions), en les utilisant.
C'est là que le challenge commence pour le programmeur : alterner les deux casquettes d'utilisateur et de programmeur est plus délicat qu'il n'y paraît, parce qu'*on s'habitue peu à peu à l'interface qu'on crée de ses mains*, au point de prendre des réflexes conditionnés et de croire évidentes des choses qui ne le sont pas. On impose alors des cheminements incohérents ou inconsistants à l'utilisateur. Pour illustrer ce piège, prenons un exemple simple :




A la création de cette fenêtre, le programmeur tient pour évident que la zone de saisie en haut de la fiche est destinée à recevoir le nom du support, et que celle du dessous recevra le même nom au pluriel. Résultat : aucun texte explicatif ne vient le dire sur la fenêtre, et un utilisateur en manque d'imagination ne saura pas quoi écrire, donc n'écrira probablement rien de peur de se tromper, et se verra dans ce cas adresser un message d'erreur du style « Vous

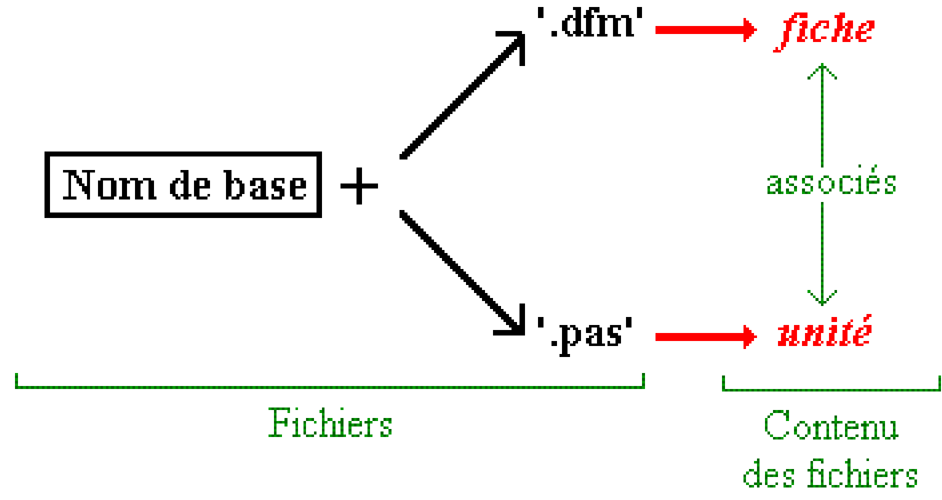

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

devez indiquer un nom pour le support ! ». Imaginez-vous dans cette situation : frustration ou dégoût ? La solution au problème est ici de réduire la taille des deux zones de saisie afin d'écrire « Nom (singulier) » et « Nom (pluriel) ». (n'essayez pas encore d'effectuer de manipulations dans Delphi, cela sera vu en détail dans la suite du guide). Au contraire, la case à cocher du dessous est bien signalisée, l'utilisateur sait ce qu'il active comme option en la cochant.
Delphi met à votre disposition tout un arsenal (et je vous assure que le mot n'est pas trop faible) d'outils pour réaliser une interface (partie d'un logiciel réservée aux utilisateurs) fouillée, précise et agréable. Mais ceci a un prix : vous devez savoir vous servir de Delphi pour apprécier la puissance qui est mise à votre disposition.
Dans la partie qui vient, quelques principes de Delphi vont être abordés. Cette partie que vous allez peut-être trouver fastidieuse est un passage obligé pour bien comprendre l'interface de Delphi qui sera examinée ensuite.

## IV-D - Notions de base de la programmation sous Delphi

### IV-D-1 - La notion de projet

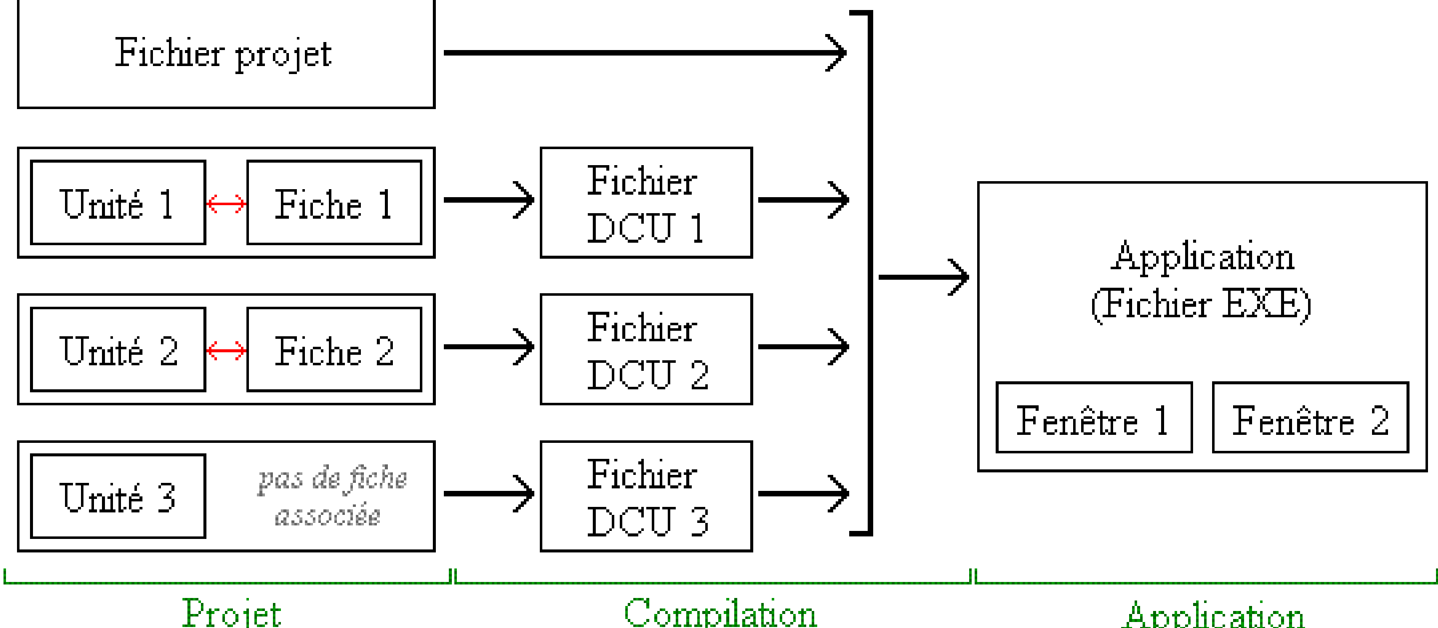
Delphi permet de créer une seule application (un futur logiciel) à la fois, ouverte en tant que projet. Un projet est l'état non compilé d'une application (d'un logiciel). Chaque projet compilé devient une application. Concrètement, un projet se compose d'un certain nombre de fichiers et d'options (également stockées dans des fichiers). Une sage habitude est de consacrer complètement un répertoire (dossier) à chaque application qu'on souhaite programmer (chaque projet). Ce répertoire contiendra tous les fichiers constituant le projet (le nombre de fichiers augmentera au fur et à mesure que le projet s'étoffera).
Une application Windows est constitué exclusivement de fenêtres. Tout ce qu'un logiciel vous montre est contenu dans une fenêtre (même si elle est parfois bien déguisée). Les logiciels simples peuvent ne contenir qu'une fenêtre. Ceux qui sont plus compliqués peuvent en contenir des dizaines (rarement plus).
Un projet non compilé (c'est-à-dire avant sa transformation en logiciel), contient ces fenêtres (à l'état non compilées également) : les fiches. Comprenez bien que chaque fenêtre que vous voudrez dans votre logiciel existe d'abord sous forme de fiche. Cette fiche, lors de la compilation du projet en application, sera transformée en fenêtre.
A chaque fiche est adjointe une (et une seule) unité, c'est-à-dire un texte écrit en langage Pascal, qui contiendra tout ce qui se rapporte à cette fiche : ce qu'elle contient (boutons, menus, cases à cocher, #), ce qui doit se passer dans certaines situations (lorsqu'on clique sur un bouton par exemple), et encore bien d'autres choses.
Au niveau informatique, chaque fiche est stockée dans un fichier (ce fichier ne contient que cette fiche et rien d'autre) comportant l'extension 'DFM' (par exemple « Options.dfm »). Chaque unité est également stockée dans un fichier ne contenant que cette unité et portant l'extension 'PAS' (par exemple « Principale.pas »). La fiche et son unité associée portent le même nom (mais pas la même extension) (Delphi ne demande ce nom qu'une seule fois et l'utilise automatiquement pour l'unité et la fiche, ainsi vous n'avez aucun risque de vous tromper). Par exemple, si une fiche est stockée dans le fichier nommé « FichePrinc.dfm », son unité associée sera stockée dans le fichier nommé « FichePrinc.pas ».
Comme un schéma vaut souvent mieux que de longues explications, voici :





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il est également possible d'utiliser des unités qui n'ont pas de fiche associée. Ceci est utile par exemple pour rassembler des morceaux de programme qui n'ont aucun rapport avec une quelconque fiche, telles des algorithmes de calcul mathématique. Le fichier qui contient cette unité porte l'extension 'PAS', comme s'il avait une fiche associée, mais il n'en a pas : il n'y a pas de fichier nommé identiquement, mais avec l'extension 'DFM'.
Un projet sous Delphi est constitué d'un fichier-projet (portant l'extension 'DPR'), d'unités et de fiches. Chaque fiche a son unité correspondante, mais certaines unités n'ont pas, comme on l'a vu ci-dessus, de fiche associée (il y a toujours au moins autant d'unités que de fiches dans un projet, et souvent plus). Pensez en outre à donner des noms descriptifs, ou au moins significatifs lors des enregistrements des unités ou des fiches (Ainsi, « unit1.pas » est un bien mauvais choix !).
Lors de la compilation du projet (transformation en application), d'autres fichiers seront créés : des fichiers DCU (fichiers portant l'extension 'DCU', essayez de vous habituer à ces abus de langage). Ces fichiers sont la forme compilée des fichiers de même nom qui portent les extensions 'PAS' et 'DFM' : un 'PAS' (avec son 'DFM' associé s'il y en a un) est transformé en 'DCU' lors de la compilation. Un fichier 'EXE' est enfin créé si la compilation atteint son terme (si tous les fichiers 'DCU' ont pu être créés). Ce fichier est nommé avec le même nom de base que celui du fichier projet (fichier DPR).
Le schéma ci-dessous représente la transformation d'un projet en application, avec les éléments que vous devez désormais connaître. Ce projet comporte deux fiches (avec les unités associées) ainsi qu'une unité seule :




Lors des enregistrements successifs d'un projet, d'autres fichiers avec des extensions ~DP, ~DF, ~PA, ~DC, ~DPR, ~DFM, ~PAS, ~DCU sont créés : ce sont des copies de sauvegarde (des versions plus anciennes) des fichiers portant le même nom, sans le signe tilde (~)
(exemple : « optionsaffich.~PA » est une copie de sauvegarde de « optionsaffich.pas »)
Enfin, d'autres fichiers ayant le même nom que le fichier projet (celui qui porte l'extension DPR) mais avec l'extension .RES, .OPT, .CFG, .DOF, .DSK, .DSM sont créés dans certaines circonstances. Vous n'avez pas à vous souciez de ces fichiers.
Le tableau suivant donne une liste aussi complète que possible des fichiers pouvant être rencontrés dans le répertoire d'un projet Delphi :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| Extension du fichier | Description et Commentaires |
| --- | --- |
| DPR | (**D**elphi **PR**oject)<br>Contient l'unité principale du projet |
| PAS | (**PAS**cal)<br>Contient une unité écrite en Pascal. Peut avoir un .DFM correspondant |
| DFM | (**D**elphi **F**or**M** : fiche Delphi)<br>Contient une fiche (une fenêtre). Le .PAS correspondant contient toutes les informations relatives au fonctionnement de cette fiche, tandis que le .DFM contient la structure de la fiche (ce qu'elle contient, sa taille, sa position, #). Sous Delphi 5, les .DFM deviennent des fichiers texte qu'il est possible de visualiser et de modifier. La même manipulation est plus délicate mais possible sous Delphi 2 à 4. |
| DCU | (**D**elphi **C**ompiled **U**nit : Unité compilée Delphi)<br>Forme compilée et combinée d'un .PAS et d'un .DFM optionnel |
| ~??? | Tous les fichiers dont l'extension commence par ~ sont des fichiers de sauvegarde, pouvant être effacés pour faire place propre. |
| EXE | Fichier exécutable de l'application. Ce fichier est le résultat final de la compilation et fonctionne sous Windows exclusivement. Pour distribuer le logiciel, copier ce fichier est souvent suffisant. |
| RES | (**RES**source)<br>Fichier contenant les ressources de l'application, tel son icône. Ce fichier peut être édité avec l'éditeur d'images de Delphi. Ces notions seront abordées plus loin dans ce guide. |
| DOF<br>DSK<br>CFG | Fichiers d'options : suivant les versions de Delphi, ces fichiers contiennent les options du projet, les options d'affichage de Delphi pour ce projet, ... |

Lorsque vous aurez atteint des projets assez importants en taille (donc en nombre de fichiers), il pourra être avantageux de régler Delphi pour qu'il place les fichiers DCU d'une part, et le fichier EXE d'autre part, dans d'autres emplacements. Ceci permet de bien ranger les fichiers d'un projet. Cette manipulation sera décrite plus tard dans le guide.

## IV-D-2 - La notion de composant

Les plus attentifs auront certainement déjà fait le rapport avec une certaine « Palette des composants » déjà mentionnée précédemment, et qui le sera plus en détail dans un avenir très proche.
Nous avons déjà vu que Delphi permet de créer des programmes (on assimile souvent, par abus, le terme programme et le terme logiciel. Ici, les « programmes » mentionnés désignent des logiciels et non des textes en Pascal comme vous l'avez vu plus tôt) sous Windows, avec la possibilité d'utiliser ce que propose Windows à tout logiciel, à savoir tout un tas d'éléments prédéfinis permettant l'interaction avec l'utilisateur : Les fiches (i.e. les fenêtres : rappelez vous que 'Windows', en anglais, cela signifie 'Fenêtres') sont les premiers de ces éléments, mais il y en a une foule d'autres, parmi lesquels les boutons, les cases à cocher, les zones d'édition, les menus, les barres d'outils, les listes,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

les arbres (pensez à la partie gauche de l'Explorateur Windows) et encore des tas d'autres (Chacun de ces éléments se retrouve dans bon nombre de logiciels, et pour cause : Windows les fournit en standard aux programmeurs).
Chacun de ces éléments, à l'exception des fiches qui ont un statut particulier, est représenté par le terme composant, et est à ce titre accessible dans la « palette des composants » de Delphi. Ces composants seront, comme s'il s'agissait de dessiner, placés, dimensionnés, réglés un par un sur les fiches et permettront de constituer une interface utilisateur (le terme "dessin" est bien celui qui convient sous Delphi : Vous pourrez vraiment placer n'importe quel composant sur une fiche, tel un bouton, le déplacer, le dimensionner à loisir. C'est, vous pourrez le constater par vous-même, très agréable au début, on a un peu l'impression de jouer au magicien)

## IV-D-3 - Propriétés et événements

Ces notions sont assez délicates et nous allons juste expliquer ici le strict minimum qu'il vous faut savoir pour comprendre le fonctionnement de l'inspecteur d'objets (fenêtre « Inspecteur d'objets » de Delphi) dans la partie suivante.
En examinant l'inspecteur d'objets, vous avez certainement découvert deux onglets mystérieux : « Propriétés » et « Evénements ». Ces deux termes s'appliquent à tout « composant » accessible dans la barre de composants de Delphi, ainsi qu'aux fiches. Chaque fiche, chaque composant possède une liste de propriétés et une liste d'évènements.
Les propriétés sont des paramètres réglables pour un composant. Par exemple : les dimensions, les couleurs, les polices, le titre d'une fenêtre, le texte d'un bouton...
Les évènements sont tout autre : Lorsque vous utilisez un logiciel, vous provoquez sans cesse des évènements, sans même le savoir. Ainsi, clics et mouvements de souris, touches frappées au clavier font partie des évènements les plus simples. D'autres sont provoqués lorsqu'une fenêtre devient visible, invisible, lorsqu'une case à cocher est cochée, lorsque dans une liste un élément est sélectionné...
La presque totalité des composants déclenchent des évènements pendant l'exécution du logiciel. Ces évènements seront pour vous autant d'informations sur les agissements de l'utilisateur, et vous pourrez répondre à tel ou tel événement ou l'ignorer totalement (ce qui est le cas par défaut).

# IV-E - Premier aperçu de l'interface de Delphi

L'objet de cette partie est autant de vous familiariser avec l'interface de Delphi que de donner des informations fondamentales mais complètes sur la gestion des projets sur Delphi. Le mieux est d'avoir Delphi ouvert en même temps que ce document et d'effectuer au fur et à mesure les manipulations afin de briser la glace entre vous et Delphi.
L'utilisateur de Windows que vous êtes sûrement aura peut-être remarqué que Delphi n'est pas une fenêtre unique qui mobiliserait tout l'écran, avec des sous-fenêtres à l'intérieur, mais est constitué d'un « bandeau » regroupant menus, barres d'outils et palette de composants. Les autres fenêtres du logiciel peuvent être positionnées n'importe où sur l'écran. En voici quelques-unes parmi les plus importantes :

- L'inspecteur d'objets, dans une fenêtre flottante, est un outil indispensable et vous apprendrez à apprécier sa puissance cachée.
- L'éditeur de code, qu'il contienne seulement du texte ou qu'il soit agrémenté d'autres petites parties (qui ne seront décrites en détail que lorsque vous aurez les connaissances nécessaires pour les utiliser), est l'endroit ou vous écrirez votre Pascal.

Mais place aux détails.

## IV-E-1 - La barre de menus

Dans toute application Windows suffisamment imposante, une barre de menus permet d'accéder à la plupart des commandes disponibles. Delphi ne fait pas exception à cette règle sage, avec une barre de menus assez bien conçue (à partir de la version 4 en tout cas, mais ceci est une question de goût).
Le menu Fichier permet la gestion des fichiers constituant le projet en cours, le menu Edition permet les fonctions classiques d'édition du texte, le menu Recherche permet d'effectuer des recherches dans de longs programmes, le menu Voir permet d'avoir accès aux différentes fenêtres de Delphi, d'afficher des éléments constituant une application, le menu Projet permet d'accéder aux commandes spécifiques au projet (l'application en gestation) en cours, le menu Exécuter permet la compilation et le lancement de l'application ainsi créée, le menu Outils donne


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

accès à divers outils de Delphi, donc un seul est vraiment intéressant : l'éditeur d'images. Le menu Aide, enfin, permet d'accéder à l'aide du logiciel, plus ou moins bien faite suivant les versions (l'aide de la version 2 est assez mal fichue, celles des autres versions est nettement mieux faite), et l'accès à l'aide sur les API Microsoft (tout cela sera expliqué plus tard).
Il ne s'agit pas ici de connaître chacun des éléments de menus de Delphi, c'est inutile à ce stade car certaines commandes ne servent qu'en de rares occasions. Il est cependant essentiel de vous familiariser avec les commandes les plus utiles pour la gestion des projets : Nouveau#, Ouvrir, Enregistrer, Enregistrer sous#, Fermer. Il est malheureusement difficile ici de décrire ces éléments qui changent légèrement avec les versions de Delphi. Voici la description de quelques éléments pour Delphi 6 (ces éléments se retrouvent tous sous un nom très voisin dans les versions moins récentes) :

## IV-E-1-a - Menu Fichier

| | |
|---|---|
| Nouveau# | Permet de créer un nouvel élément. Cet élément peut être beaucoup de choses, parmi lesquelles un projet, une unité (un .PAS sans .DFM correspondant), une fiche (un .PAS et un .DFM associés) et beaucoup d'autres choses qui dépassent le cadre de ce guide (certains choix, comme 'Composant', 'Cadre', 'DLL' seront abordées plus loin). |
| Ouvrir | Permet d'ouvrir un fichier : on ouvre un projet en ouvrant son fichier .DPR unique. On ouvre une fiche en ouvrant le fichier .DFM ou le fichier .PAS correspondant. On ouvre une unité (sans fiche) en ouvrant le .PAS qui la contient. Enfin, tout fichier texte peut être ouvert dans Delphi, selon des besoins divers et variés. |
| Enregistrer | Permet d'enregistrer l'élément en cours d'édition. Cet élément est soit une unité (cf. description de l'éditeur de code), soit une fiche. Si l'élément n'a encore pas été enregistré, un nom vous sera demandé. Il peut être temps de créer un répertoire pour le projet, et de réfléchir au nom du futur fichier .EXE du logiciel si on enregistre le .DPR (car il est rappelé que ces deux fichiers ont le même nom de base). |
| Enregistrer sous... | Comme enregistrer, mais permet en plus de définir un nouveau nom. |
| Tout Enregistrer | Permet d'enregistrer d'un coup tous les fichiers d'un projet ouvert. Des noms de fichiers vous seront demandés si c'est nécessaire. |
| Fermer | Ferme l'élément en cours d'édition. Propose l'enregistrement si c'est nécessaire. |
| Tout Fermer | Ferme tous les éléments ouverts dans Delphi. Dans Delphi 2 à 4, si j'ai bonne mémoire, c'est plutôt 'Fermet Projet'. |
| Quitter | Quitte Delphi |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## IV-E-1-b - Autres menus

Chacun des autres menus possède des commandes intéressantes, mais ce serait assez fastidieux de les lister toutes, autant pour vous que pour moi, sachez toutefois que chacune de ces commandes sera nommée et expliquée en temps voulu dans le guide.

## IV-E-2 - La barre d'outils

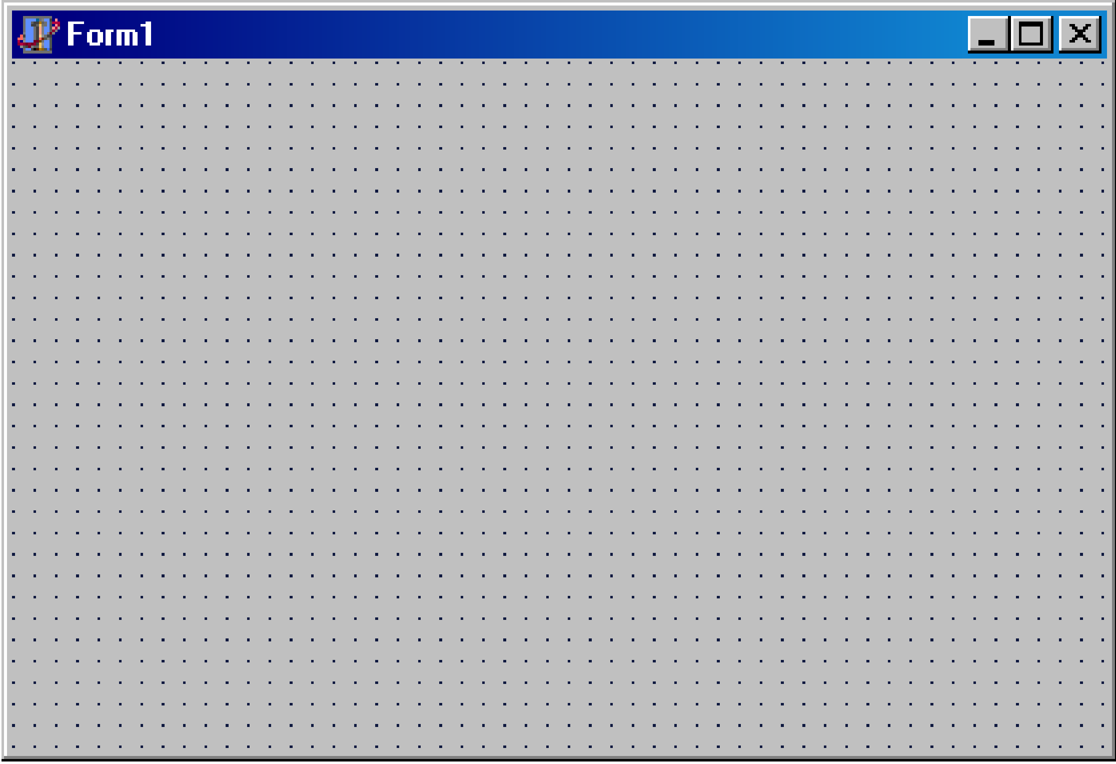
Comme dans toute application Windows, la barre d'outils permet l'accès rapide à certaines commandes des menus. Il est impossible ici de décrire chaque bouton étant donné qu'ils changent à chaque version de Delphi. Pour avoir une description de chaque bouton, lisez la bulle d'aide qui ne manquera pas de s'afficher lors du passage de la souris.
La barre d'outils pourra, dès la version 2, être personnalisée pour être agrémentée à vos goûts et habitudes. Pour cela, effectuez un clic droit sur l'une des barres et une commande « personnaliser » devrait être disponible. Les manipulations pour personnaliser les barres sont similaires à celles qu'on rencontre dans des logiciels tels Microsoft Word.

## IV-E-3 - La palette des composants

Cette palette, généralement située à droite en dessous des menus, donne accès à l'ensemble des composants (voir définition plus haut si besoin) utilisables avec Delphi. Ces composants, trop nombreux pour être tous présents sur une simple barre d'outils, sont présents dans un classeur à onglets. Chacun de ces onglets donne accès à un certain nombre de composants. C'est ici que vous trouverez ce que Windows vous offre pour constituer votre interface : boutons, listes, cases à cocher, arbre et listes, grilles, ...
Il n'est pas encore nécessaire pour vous de connaître une liste de composants utiles, cela viendra plus tard. Pour l'instant, prenez simplement le temps de choisir un onglet, et de promener la souris (sans cliquer) sur chacun des boutons qui apparaissent. Les petites images présentes sur ces boutons vous feront parfois penser à des éléments que vous connaissez déjà, tel celui présenté ci-dessous (présent dans l'onglet 'Standard') : Composant MainMenu qui représente une barre de menus ('MainMenu' s'affiche dans une bulle d'aide pour ce bouton)
Il n'est pas encore question ici pour vous de commencer à créer une interface, alors évitez de cliquer ou de double-cliquer sur ces boutons. Dans la suite du guide, vous trouverez une liste des composants à connaître, ainsi qu'une description et leurs utilisations possibles.
Voici maintenant l'instant tant attendu de la manipulation. La suite d'instructions ci-dessous vous permettra de placer un bouton sur une fiche :

1. Créez un nouveau projet (en utilisant les menus de Delphi) : menu Fichier, commande 'Nouveau Projet' ou 'Nouvelle Application'
2. Ceci a pour effet de créer un projet minimum constitué du fichier projet (non visible), d'une unité et de la fiche correspondante à cette unité. Vous devez à présent avoir sous les yeux une fiche vide :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



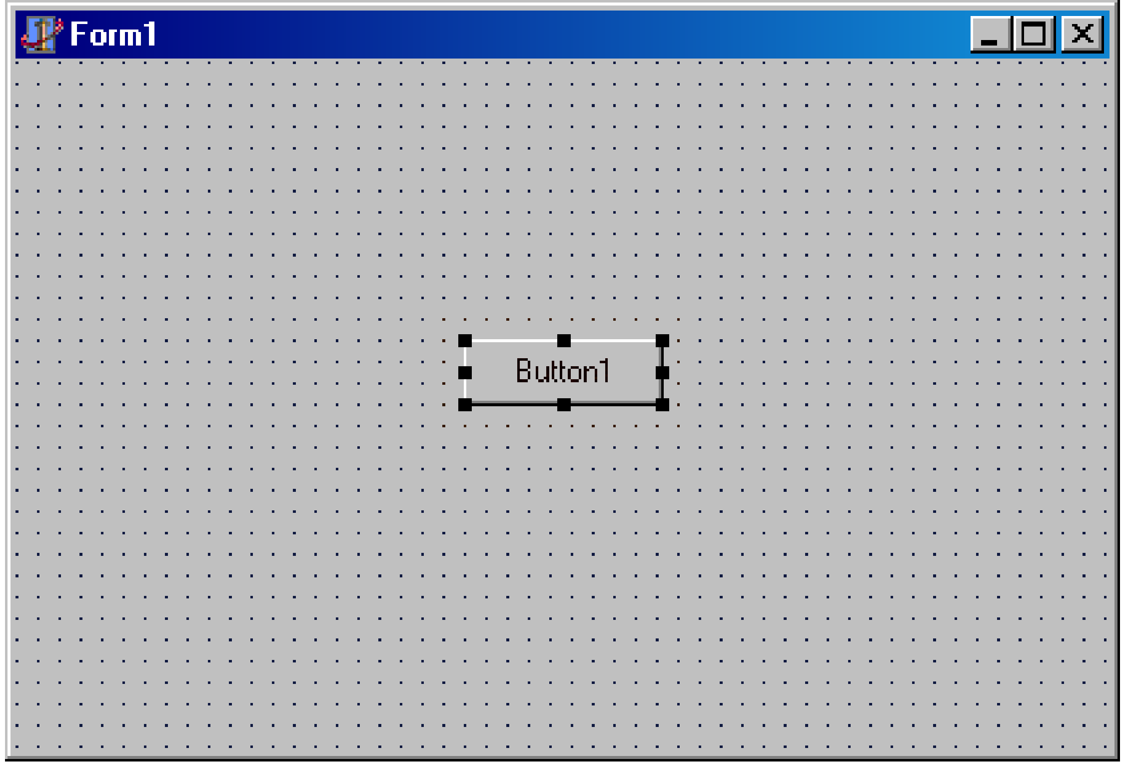

Si vous avez seulement du texte coloré du genre 'unit unit1; #', appuyez sur F12 pour voir la fiche. Remarque : les points répartis en une grille sur la fiche servent à placer les composants. Ces points n'existent plus lorsque la fiche est compilée et devient une fenêtre.

3 Une fois la fiche visible à l'écran, allez dans la palette des composants, trouvez le bouton

Button (son info-bulle doit être 'Button') dans l'onglet nommé 'Standard', et cliquez une seule fois dessus, ceci a pour effet de le faire paraître enfoncé.

4 Cliquez maintenant une seule fois a peu près au centre de la fiche. Ceci a pour effet de placer un bouton ('button' en anglais) sur la fiche avec comme texte probable : 'Button1'.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



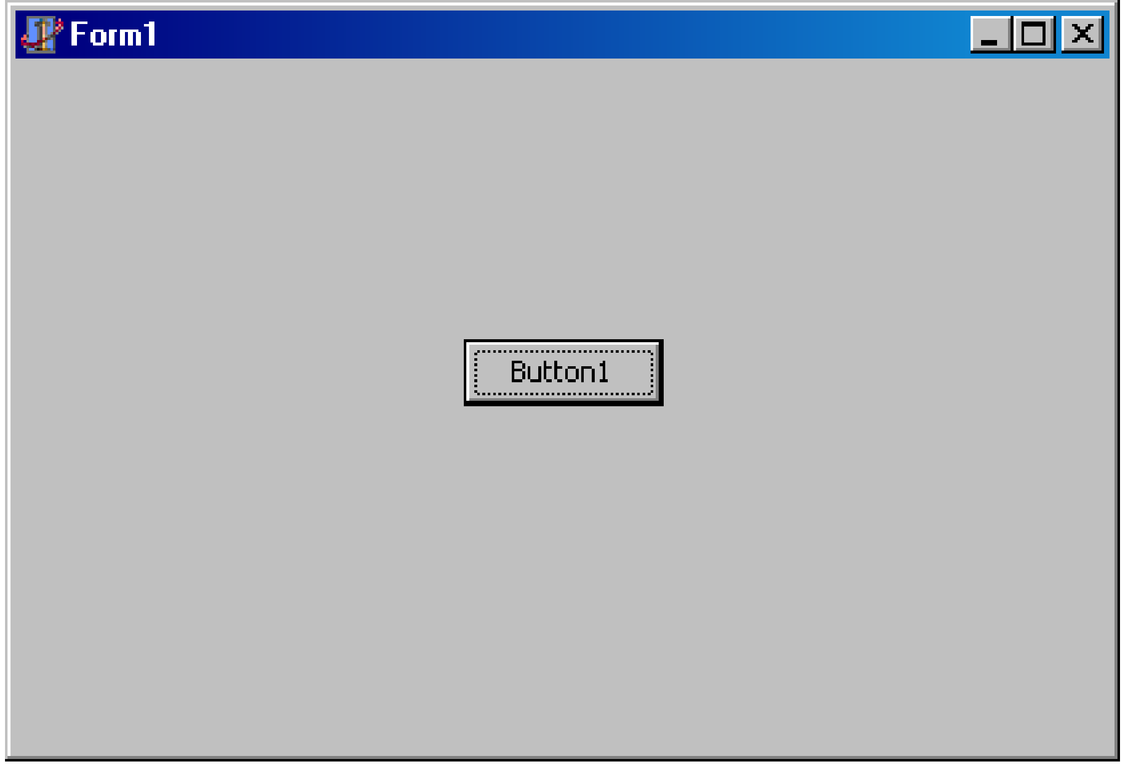

5 Pour déplacer le bouton, cliquez dessus (en dehors des petits carrés noirs), et déplacez la souris en maintenant le bouton gauche de la souris : le bouton suit la souris et se repositionne quand vous relâchez le bouton gauche (c'est la même manipulation que celle qui permet, par exemple, de faire un glisser-déposer (drag-drop) dans l'explorateur de Windows). La même manipulation est possible avec la fiche en cliquant sur sa barre de titre.

6 Pour changer la taille du bouton, positionnez le pointeur de la souris sur l'un des carrés noirs qui entourent le bouton (si ces poignées ne sont pas visibles, cliquez une fois sur le bouton, et plus généralement, retenez qu'on peut sélectionner un composant en cliquant dessus) : maintenez le bouton gauche de la souris enfoncé et déplacez ainsi le petit carré en question. Ceci permet de changer les dimensions d'un composant. La même manipulation est possible avec la fiche comme avec n'importe quelle fenêtre Windows redimensionnable.

7 Enregistrez maintenant le projet dans un répertoire vide de votre choix en cliquant sur le menu Fichier, puis 'Tout Enregistrer' ou 'Enregistrer projet'. Des noms de fichiers vont vous être demandés, avec de bien mauvaises suggestions. Lorsqu'on vous demandera de nommer le projet (nom suggéré : 'project1.dpr'), tapez à la place 'PremierEssai' (Delphi rajoutera '.dpr' tout seul). Lorsque l'on vous demandera un nom pour la seule unité de votre projet (nom suggéré : 'unit1.pas'), tapez à la place 'principale' (de même, Delphi ajoutera '.pas' tout seul). (Remarque : Puisque vous êtes sous Windows 95 ou ultérieur, vous pouvez taper des noms de plus de 8 caractères, ce qui est fort appréciable. Vous pouvez également utiliser des majuscules, mais vous devez vous limiter à des noms commençant par une lettre, sans espaces et n'utilisant que des lettres, des chiffres et le caractère _ (blanc souligné), mais pas de - (tiret), d'accents ou de caractères exotiques qui seraient rejetés par Delphi ou provoqueraient des erreurs (La raison de ces limitation sera donnée plus tard).

8 Vous voulez lancer votre premier projet (en faire un logiciel et l'exécuter) ? Rien de plus simple, appuyez sur F9. Ceci a parfois pour effet de montrer une petite fenêtre informant de la progression de la compilation. Ensuite, quelque chose de ce genre devrait apparaître :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Bien sûr, pour un premier essai, ce n'est pas une interface très élaborée, mais c'est un logiciel pour Windows que vous avez devant les yeux. Essayez de cliquer sur le bouton. Il ne se passe rien parce que vous n'avez rien prévu qu'il se passe. Il aurait fallu répondre à l'événement « clic sur le bouton » et donner des instructions, mais c'est encore un peu tôt.

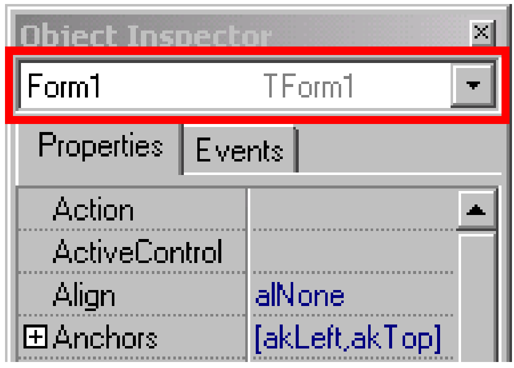
9 Pour quitter (à regret ou pas) votre premier logiciel, deux choix, soit vous appuyez sur Alt + F4, soit vous cliquez sur la petite croix en haut à droite de sa fenêtre. Vous revenez alors sous Delphi (l'une des premières choses que nous ajouterons sera un menu Fichier avec un choix Quitter).

## IV-E-4 - L'inspecteur d'objets

Si vous ne voyez pas la fenêtre « Inspecteur d'objets » actuellement, appuyez sur F11, ou dans le menu Voir, sélectionnez 'Inspecteur d'objets'. L'inspecteur d'objets permet, en employant des termes qui doivent maintenant vous être familiers, de modifier les propriétés et d'accéder aux événements des composants et des fiches. C'est un outil constamment utilisé lors de la conception de l'interface de vos logiciels. Il se compose d'une liste déroulante (combo), listant les composants présent sur une fiche, ainsi que cette fiche. Les propriétés et événements d'un élément sélectionné sont classées dans les deux onglets 'Propriétés' et 'Evénements'.
Pour éditer un composant ou la fiche dans l'inspecteur d'objets, il suffit de cliquer dessus dans la fiche correspondante (ceci le sélectionne). Essayez donc maintenant en cliquant (une seule fois) sur le bouton que vous venez de créer. L'inspecteur d'objet se met à jour si besoin en affichant les propriétés ou les événements de ce bouton (suivant l'onglet de l'inspecteur d'objets sélectionné). Cliquez maintenant sur la fiche (à un endroit où vous voyez le quadrillage) : le contenu de l'inspecteur d'objets change encore une fois pour afficher les propriétés et événements relatifs à la fiche.
Voici une manipulation qui vous aidera à comprendre son fonctionnement :

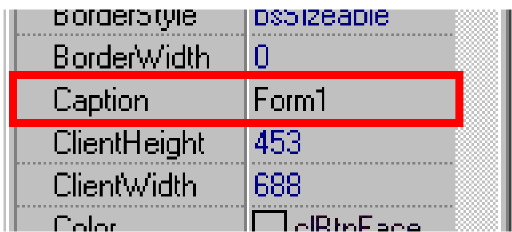
1 A partir du projet dans l'état où il doit être actuellement (une fiche avec un bouton dessus au centre), vérifiez que c'est bien la fiche qui est sélectionnée (en cliquant une fois sur un espace de la fiche ne contenant pas de composant) (si la fiche n'est pas visible, affichez l'inspecteur d'objets et appuyez sur F11).
2 L'inspecteur d'objets doit montrer le texte suivant dans la zone de sélection déroulante :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

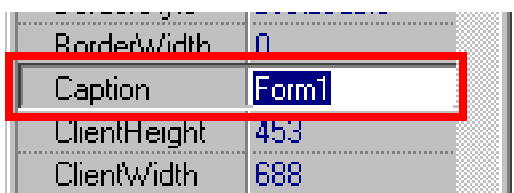



Ici, c'est l'onglet 'Propriétés' (" properties " en anglais) qui est sélectionné. En dessous, deux colonnes : la première donne le nom des propriétés disponibles, celle de droite donne leur valeur actuelle. Essayez de cliquer sur l'onglet 'Evénements' (« events » en anglais) pour voir la liste des événements disponibles. Normalement, à ce stade du développement, aucun événement ne doit avoir de valeur. Retournez sur l'onglet 'Propriétés'.

3 En utilisant si besoin la barre de défilement de droite, trouvez la propriété dont le nom est 'Caption' ('Légende' en français).




4 Sélectionnez cette propriété en cliquant soit sur son nom, soit sur sa valeur (qui doit être 'Form1')




5 La propriété 'Caption' d'une fiche correspond au texte qui est écrit dans sa barre de titre (le bandeau supérieur coloré). Vous pouvez modifier librement ce texte en vous servant de la zone d'édition contenant le texte 'Form1'. En même temps que vous tapez dans cette zone de saisie, regardez la barre de titre de votre fiche, et remarquez comment elle se met à jour selon le texte que vous tapez. Une fois que vous avez terminé l'édition du texte, cliquez simplement ailleurs ou appuyez sur Entrée pour accepter le nouveau texte. Appuyez plutôt sur Echap pour revenir au texte tel qu'il était avant votre dernière intervention.

6 Sélectionnez maintenant le bouton (en cliquant une fois dessus). Vous constatez qu'ici encore, une propriété 'Caption' est disponible. Changez-là et observez : 'Caption' désigne le texte affiché sur le bouton. Mais ici, entrer un texte trop long le fait dépasser des bords du bouton, il va falloir régler la largeur du bouton.

7 La largeur ('Width' en anglais) est modifiable via la propriété Width. Cette propriété n'accepte que des nombres entiers positifs. Essayez par exemple d'entrer 120 comme largeur : le bouton doit se redessiner et être plus large qu'avant (sa largeur d'origine est normalement 75). Fixez comme texte du bouton : 'Action !' (en modifiant la propriété Caption comme pour la fiche).

8 Passons maintenant à quelque chose de plus intéressant : les dimensions et l'emplacement de la fiche (vous savez déjà le faire directement avec la souris, mais il est possible de le faire avec l'inspecteur d'objets). Les propriétés concernées sont : 'Left', 'Top', 'Width', 'Height' (Respectivement Gauche, Haut, Largeur, Hauteur en français). Essayez de modifier ces 4 propriétés et regardez l'effet que cela a sur la fiche (évitez les valeurs extravagantes et remarquez comme Delphi refuse les valeurs négatives).

9 Fixez enfin comme valeurs respectives de ces 4 propriétés : 200, 130, 400, 260. (Remarque : presque tous les composants que vous placerez sur une fiche disposeront de ces 4 propriétés permettant de décider précisément des dimensions et de l'emplacement du composant).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## IV-E-5 - L'éditeur de code

L'éditeur de code est l'endroit où vous taperez l'intégralité des instructions en langage Pascal. Cet éditeur présente une série d'onglets en haut, qui donnent en titre le nom de l'unité éditée actuellement. En appuyant sur F12 lorsque l'onglet d'une unité est sélectionné, on affiche la fiche qui lui correspond (seulement si elle existe, et inversement, depuis une fiche, F12 permet de voir l'unité qui lui correspond). Ces onglets permettent également de changer d'unité éditée. L'éditeur en lui-même est un éditeur de texte assez classique avec les fonctions d'insertion, d'effacement et de copier-coller, à ceci près qu'il mettra en évidence certains mots, certaines parties de texte, suivant leur fonction. Ceci sera particulièrement pratique.
A ce propos, la relation entre une fiche et son unité est non modifiable. Il n'y a qu'au moment où vous créez une unité ou une fiche que vous avez le choix d'adjoindre ou non une fiche à l'unité. Attention cependant à ne pas tomber dans l'excès : certaines unités n'auront absolument pas besoin de fiche correspondante (des unités comportant exclusivement des calculs mathématiques par exemple, comme cela arrive fréquemment).
Au point où nous en sommes, le texte contenu dans la seule unité du projet doit vous paraître un peu obscur, et je vous comprends (je suis passé par là comme vous). C'est normal, c'est du Pascal (ou plus précisément du Pascal Objet). Remarquez simplement, pour le moment, que certains mots sont colorés, d'autres en gras ou en italique. Cette mise en évidence ne doit évidemment rien au hasard et permet de mettre en évidence certains mots, certaines parties particulières de langage Pascal. La seule ligne que vous pouvez aisément comprendre actuellement, c'est la première :

```
unit Principale;
```

Cette ligne décrit le fichier comme étant une unité nommée Principale. C'est, rappelez-vous, le nom de fichier que vous aviez choisi pour l'unité lors de l'enregistrement du projet. Sachez que le point-virgule termine l'instruction, car cette ligne est bel et bien une instruction écrite en Pascal. Delphi se chargera tout seul de cette instruction comme de quelques autres que nous verrons plus tard.

## IV-E-6 - Conclusion

Voilà, vous en savez maintenant assez sur Delphi pour le moment, vous pouvez maintenant enregistrer le projet. Nous allons passer à une partie plus théorique pendant laquelle vous utiliserez les connaissances déjà acquises et le projet dans son état actuel pour effectuer les premières manipulations et les exercices et tester les résultats. Ainsi, vous ferez vos premiers pas en Delphi en même temps que vos premiers pas en Pascal.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# V - Préliminaires

Cette partie est consacrées à l'étude du langage Pascal Objet (ou Pascal). Cette première partie introduira le langage, sa syntaxe et ses structures. Dans un premier temps, le cours sera purement théorique. Dans un second temps, dés que vous aurez les connaissances requises, les manipulations et exercices arriveront.

## V-A - Données simples dans un programme Pascal

***Révision éclair : la mémoire de l'ordinateur***

*Votre ordinateur possède, en plus du disque dur, une mémoire physique appelée aussi « mémoire vive », ou encore RAM (Random Access Memory, Mémoire à accès aléatoire). La taille de cette mémoire est généralement comprise entre 16 Mo (Mégaoctets) et 128 Mo. 1 Mégaoctet contient 1024 (2 exposant 10) Kilooctets et chaque kilooctet contient 1024 octets. Chaque octet contient 8 bits. Un bit est une unité de stockage élémentaire qui peut avoir seulement 2 valeurs : 0 ou 1. Ces 8 bits forment un nombre binaire qui, transformé en nombre décimal classique, est compris entre 0 et 255 (2x2x2x2x2x2x2x2-1).*
*Cette mémoire est occupée par tout ce qui fonctionne sur votre ordinateur, depuis son allumage et jusqu'à son extinction. Windows contrôle cette mémoire, s'en réserve une part et met le reste à votre disposition. Dans cette mémoire, les autres logiciels pourront être copiés depuis le disque dur, puis exécutés. Les logiciels pourront également y stocker leurs données.*
*Mais la taille de cette mémoire n'étant pas infinie, il faudra réfléchir lorsque vous aurez beaucoup de données à y stocker. Windows, lorsque la mémoire est pleine, sait en libérer, mais ceci a un prix : une extrême lenteur (ce n'est pas pour ça qu'il faudra rechigner à en utiliser, la mémoire est faite pour servir !)*

Tout programme Pascal que vous créerez manipulera des données. Ces données sont stockées, comme le programme, dans la mémoire vive (RAM et non disque dur, mais ce n'est pas vraiment important) de l'ordinateur.
Un type de donnée, ou type, est un nom donné à une *catégorie* très précise de donnée. Ce type permet à Delphi de savoir de quelle manière vont être stockées les données dans la mémoire, quelle quantité de mémoire sera utilisée, quelles valeurs seront autorisées et quelles opérations seront possibles sur ou à partir de ces données. Chaque donnée dans un programme devra avoir un type déterminé. Il existe quelques règles pour échapper à cette restriction, mais elles servent dans la majorité des cas à dissimuler des faiblesses dans la construction d'un programme.
Les parties qui suivent décrivent les types de données courants déjà connus par Delphi, que vous pourrez utiliser dans vos programmes. Pour chaque type de donnée, un mot Pascal vous sera indiqué, c'est ce mot que vous utiliserez dans vos programmes (la manière de vous servir de ces mots vous sera expliquée plus tard). Ensuite viendront des types plus complexes et les types personnalisés.
Vous en êtes probablement à vous dire : « C'est bien joli tout ça, et les exemples ? ». Une réponse à votre interrogation : « Ce n'est pas encore possible ». Les exemples viendront en effet lorsque vous serez à même d'écrire vous-même vos premiers programmes écrits en Pascal (ce qui ne devrait pas trop tarder). Tout ce bavardage sur les types de données, bien que fastidieux pour vous, est indispensable.

### V-A-1 - Nombres

Au grand regret de ceux d'entre vous que les mathématiques rebutent profondément, les nombres en programmation (et en Pascal entre autres) sont partout. C'est le type de données le plus simple et le plus utilisé, et de nombreuses variantes existent qui autorisent différents intervalles et permettent ou pas les nombres décimaux.

#### V-A-1-a - Nombres entiers

Les types de nombres entiers utilisables sous Delphi sont indiqués dans le tableau suivant. Les deux premier sont à utiliser le plus souvent possible car ils offrent les meilleures performances en terme de temps processeur (en gros, la meilleure vitesse, en plus fin, le processeur va plus vite à les traiter car elles prennent 4 octets, taille de prédilection pour les processeurs actuels). Les plus souvent utilisés sont mis en évidence. N'essayez surtout pas de mémoriser


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

tout le tableau, ce serait inutile et fastidieux : Il est juste présenté ici à titre d'information et de référence pour plus tard. Seuls les trois mots Integer, Byte et Word sont à retenir pour l'instant.

| Mot Pascal | Octets occupés en mémoire | Valeur minimale | Valeur maximale | Remarques |
|---|---|---|---|---|
| ***Integer*** | 4 | -2 147 483 648 | 2 147 483 647 | C'est le type entier parfait pour la plupart des usages. |
| Cardinal | 4 | 0 | 4 294 967 295 | N'accepte pas les nombres négatifs, par contre, il va plus loin dans les nombres positifs |
| Shortint | 1 | -128 | 127 | |
| Smallint | 2 | -32 768 | 32 768 | |
| Longint | 4 | -2 147 483 648 | 2 147 483 647 | Utilisez plutôt Integer |
| Int64 (Depuis Delphi 5 seulement) | 8 | $-2 \times 10^{63}$ | $2 \times 10^{63} - 1$ | Permet de passer outre les limitations de Integer pour de très grands nombres. |
| ***Byte*** | 1 | 0 | 255 | Idéal pour de petites valeurs, encombrement très réduit en mémoire. |
| ***Word*** | 2 | 0 | 65 535 | Préférable à Smallint. |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Les nombres entiers, en Pascal, s'écrivent tels quels, sans séparateur de milliers. Un nombre négatif doit être précédé d'un signe -. Les nombres positifs peuvent être précédés d'un signe +, non obligatoire et encombrant.
Ainsi, '+12' et '12' sont valides et valent le même nombre. '-12' et '- 12' sont valides et valent le même nombre.
Une petite remarque enfin pour les amateurs de nombres hexadécimaux (un nombre en base 16, contrairement à la classique base 10) : pour utiliser un nombre hexadécimal, il suffit de le précéder par le symbole $. Ainsi, $10F sera considéré comme 271.

## V-A-1-b - Nombres à virgules

Notez bien ici qu'on ne parle pas de nombres réels : si vous êtes un peu versé dans les mathématiques (comme votre dévoué serviteur), sachez que Delphi, et l'ordinateur en général, permet d'employer une bien maigre part des nombres réels.
Vous aurez parfois à utiliser des nombres à virgule dans vos programmes, ne serait-ce que pour exprimer des tailles en mètres, des prix en francs ou en euros. Pour cela, Delphi met à votre disposition quelques types de nombres à virgules, avec plus ou moins de décimales (c'est-à-dire que le nombre de chiffres après la virgule est limité). On parle de nombre de chiffres significatifs, c'est-à-dire que les nombres de chiffres avant et après la virgule, ajoutés, est limité.
Le tableau suivant, comme dans le cas des nombres entiers, résume tout cela.

| Mot Pascal | Octets en mémoire | Valeurs minimales etmaximales autorisées | chiffres significatifs | Remarques |
|---|---|---|---|---|
| ***Single*** | 4 | 1.5 x $10^{-45}$ 3.4 x $10^{38}$ | 7-8 | Idéal pour des nombres avec peu de décimales |
| Double | 8 | 5.0 x $10^{-324}$ 1.7 x $10^{308}$ | 15-16 | Intermédiaire : plus de précision mais plus de place en mémoire. |
| Extended | 10 | 3.6 x $10^{-4951}$ 1.1 x $10^{4932}$ | 19-20 | Précision maximale : idéal pour de grandes valeurs |
| ***Currency*** | 8 | -922337203685477.5808 922337203685477.5807 | | A utiliser pour les sommes d'argent : seulement 4 chiffres après la virgule, mais on peut |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | | | | |
|---|---|---|---|---|
| | | | | atteindre des sommes assez énormes. |

Les nombres à virgule s'écrivent comme les nombres entiers, à ceci près que le séparateur décimal à utiliser n'est pas la virgule mais le point (oui, ce n'est pas très malin, mais l'informatique suit les règles américaines, et ces chers américains préfèrent le point à la virgule).
Ainsi, '- 1.1' est correct, '-1,1' ne l'est pas, '1,1' est incorrect, '1.1' est correct (chaque exemple pris évidemment sans les quotes : ' ').

## V-A-1-c - Opérations sur les nombres

Les nombres, qu'ils soient entiers ou à virgule peuvent être utilisés dans des opérations. Les opérations usuelles : l'addition, la soustraction, la multiplication et la division, sont effectuées par les signes respectifs :
+ - * (étoile) / (barre oblique)
Les opérations, dans le langage Pascal, s'écrivent comme à la main. Les espaces entre les nombres et les signes d'opérations sont sans importance.
Ainsi, pour additionner 1,1 et 2,86, il faut tout simplement écrire :

```
1.1 + 2.86
```

Pour soustraire 1,2 à 2,43, il faut écrire :

```
2.43 - 1.2
```

Pour diviser 3,23 par 2,19, il faut écrire :

```
3.23 / 2.19
```

L'utilisation des parenthèses est possible et répond aux mêmes règles qu'en mathématiques. Les priorités sont les mêmes. Ainsi,

```
(2.43 - 1.2) * 2
```

et

```
2.43 - 1.2 * 2
```

ne valent pas la même chose (2.46 et 0.03).
Il est à noter ici que la division par 0 est évidemment, comme toujours, interdite et provoquera une erreur d'exécution du programme.
Il est important de comprendre que le résultat d'une opération est du type du nombre le plus complexe dans l'opération. Ainsi, si on additionne un nombre entier à un nombre à virgule, le résultat est un nombre à virgule, ou plus précisément est une donnée de type nombre à virgule, même si sa partie décimale vaut 0.
Le type de donnée du résultat est le type le plus *large* utilisé dans l'opération (celui qui autorise le plus de choses). Ainsi, la multiplication d'un nombre de type 'single' par un nombre de type 'extended' donne un résultat de type 'extended'. Par contre, lorsqu'une opération est faite entre deux nombres de mêmes types, le type du résultat est ce type.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La division est une exception à cette dernière règle : le résultat est toujours à virgule, quels que soient les nombres mis en #uvre (pensez à 9 / 2 qui vaut 4.5 et qui ne peut donc pas être stocké sous la forme d'un nombre entier).
Il est également possible d'utiliser les deux autres opérations mod et div à la place de /. L'opération div effectue une division entière, à savoir qu'elle renvoie le quotient entier de la division (dans l'exemple de 9 / 2, l'opération 9 div 2 renvoie 4). L'opération mod est le complément de div et renvoie le reste de la division entière (Ainsi : 9 mod 2 vaut 1).

Exercice 1 : (voir la **solution**)

1. Ecrire l'opération qui ajoute 2 à 4,1 et divise ce nombre par la soustraction de 4 à -2
2. De quel type de données le résultat sera-t-il ?

Autre piège classique de l'informatique : si nous avons deux valeurs de type 'byte', disons 200 et 100, que croyez-vous que vaut 200+100 dans ce cas ? La réponse qui saute aux lèvres est 300, et c'est une réponse fausse, puisque le type du résultat est 'byte', qui ne supporte que les nombres entre 0 et 255 ! Que vaudra alors 200+100 ? La réponse exacte est 44 (vive l'informatique !), car l'ordinateur a enlevé autant de fois 256 (c'est un modulo l'amplitude du type (le nombre de valeurs possibles), ici 255 - 0 + 1) qu'il fallait pour retomber dans un intervalle correct. Il faut se méfier de ce genre de comportement car alors si on a deux 'integer' de valeurs 2000000000 et 1000000000, leur somme vaut non pas 3000000000 mais -1294967296 ! Etonnant pour une somme de deux nombres positifs, non ?

*Attention : ce qui est expliqué ici est un piège dont il faut se méfier, et non pas un moyen d'effectuer des calculs, parce qu'on ne peut se fier aux résultats de calculs menés hors des sentiers battus.*

Exercice 2 : (voir la **solution**)
On a deux valeurs de type 'word' : 65535 et 2, combien vaut dans ce cas précis 65535 + 2 ?

*Remarque : Si vous n'avez pas tout (ou rien) compris, ce n'est pas bien grave. Adressez-vous au professeur de maths le plus proche et il vous expliquera tout ça, ou alors vous pouvez me contacter pour me demander des explications plus détaillées ou vous adresser à la* ***mailing-list****.*

## V-A-2 - Caractères et chaînes de caractères

Pour utiliser des caractères tels que ceux que vous êtes en train de lire, il vous faudra utiliser d'autres types de données, à savoir les types caractères et chaînes de caractères.
Une précision avant de commencer : les caractères et chaînes de caractères font la différence, au contraire du reste d'un programme, entre majuscules et minuscules.
Le type caractère, associé au mot Pascal 'char' (on dira maintenant type char, ou type single, ou type integer pour désigner les types), permet de stocker un unique caractère, que ce soit une lettre, un chiffre, des caractères moins fréquents tels que : # ! ? , ; . : @. Etc.
C'est parfois très pratique, comme nous le verrons plus tard, mais se limiter à un seul caractère est tout de même gênant. C'est pour cela que le type 'string' (« chaîne » en français) existe. Il permet, dans les versions 2 et 3 de Delphi, de stocker 255 caractères, et n'a plus de limite à partir de la version 2 (bien que l'ancien type limité à 255 caractères soit alors accessible par le type 'shortstring').
Il est également possible de personnaliser la longueur (tout en respectant quand même la limite maximale de 255 dans ce cas) en utilisant le type 'string[n]' ou n prend une valeur comprise entre 1 et 255. Ce type permet d'utiliser des chaînes d'au plus n caractères.
Les chaînes et les caractères se notent entre simples quotes (' ') à ne pas confondre avec les guillemets (" ") :
'truc' est une chaîne, 'truc et 1000 autres trucs' en est une autre. Pour inclure une simple quote dans une chaîne, il suffit d'en mettre deux. Ainsi, 'l''oiseau' représente en fait « l'oiseau ». La chaîne vide se note ''.
Il est possible de concaténer deux ou plus de chaînes en une seule (c'est-à-dire de les coller bout à bout). Pour cela, on utilise l'opérateur +.
Exemples :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

1 'inte' + 'lligent' et 'intelligent' désignent la même chaîne de caractères.
2 'i'+'ntelli'+'gent' et 'intellig'+'ent' également.
3 par contre, 'i'+' ntelli '+'gent' , qui comporte des espaces, est différente de 'intelligent'.
4 'exemple'+ ' ' +'idiot' et 'exemple idiot' représentent la même chaîne.
5 'l'+''''+'oiseau' et 'l''oiseau' également.
6 'l'+''''+'Oiseau' et 'l''oiseau', par contre, sont différentes, car la première contient une majuscule et pas la deuxième.

Deux remarques pour terminer cette partie :

1 'abc' + 1 n'a pas de sens (il faudrait écrire 'abc' + '1' pour obtenir 'abc1'), car on ne peut pas additionner une chaîne et un nombre. On dit alors que ces deux types sont incompatibles. Il existe des moyens pour passer outre ce genre d'incompatibilité, même si l'intérêt ne saute pas aux yeux, mais il est encore un peu tôt pour en parler.
2 Ensuite, une chaîne de caractère ne peut pas être coupée par un retour à la ligne. C'est-à-dire que lorsque vous taperez une chaîne dans Delphi, vous n'aurez pas le droit de l'écrire directement sur plusieurs lignes. Si vous souhaitez le faire, découpez-la en morceaux (écrits chacun sur une seule ligne) concaténés ensuite par des opérateurs +. Ceci sera illustré dans des exemples présents dans la suite du guide.

## V-A-3 - Booléens

**Révision éclair : les booléens**
On appelle booléen, ou valeur booléenne, une valeur pouvant être 0 ou 1. On associe souvent ces deux valeurs aux deux expressions logiques « faux » (« false » en anglais) et « vrai » (« true » en anglais).
Les opérateurs booléens permettent de combiner logiquement plusieurs valeurs booléennes. Les opérateurs à connaître sont (avec leur table de vérité) :

- **not (négation)**
  (not A) est la négation logique de A, à savoir vrai si A est faux, et faux si A est vrai.

| A | not A |
|---|---|
| 0 | 1 |
| 1 | 0 |

- **or (ou inclusif)**
  (A or B) est vrai si A est vrai, si B est vrai, ou si A et B sont vrais à la fois.

| A | B | A or B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

- **and (et)**
  (A and B) est vrai seulement si A et B sont vrais à la fois.

| A | B | A and B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- **xor (ou exclusif)**
  (A xor B) est vrai si exactement l'une des deux valeurs de A et B est vrai et que l'autre est fausse (i.e. A faux et B vrai, ou bien A vrai et B faux).

| A | B | A xor B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Les tables de vérité de ces opérateurs sont également à connaître et sont données ici à titre indicatif.
Les valeurs booléennes (0 ou 1) ont une place privilégiée en informatique, et sont naturellement intégrées à Delphi.
Le type 'boolean' permettra d'avoir une donnée de type booléen. Une telle donnée peut avoir seulement deux valeurs : false (faux, valeur : 0) ou true (vrai, valeur : 1).
Le mot Pascal **not** permet la négation d'une donnée booléenne. Ainsi, « not true » vaut « false » et « not false » vaut « true ».
Il est possible d'utiliser les opérateurs booléens classiques, qui sont des mots Pascal : **or** (ou inclusif logique), **xor** (ou exclusif logique) et **and** (et logique) avec les données de type 'boolean'. Ces opérateurs répondent aux priorités en vigueur en Logique, à savoir :

1. **not** est le plus prioritaire ;
2. **and** vient ensuite ;
3. **or** et **xor** viennent enfin au même niveau.

Exemples :

1. true **and** true vaut 'true'
2. false **or** true vaut 'true'
3. **not** false **and** true vaut 'true'
4. **not** false **xor** **not** true vaut 'true'
5. **not** **not** false vaut 'false'

Les expressions booléennes permettront souvent de décider si une partie de programme est exécutée ou non (la valeur true entraînera l'exécution d'un morceau de programme, la valeur false permettra de passer par dessus, par exemple, ou d'exécuter un autre morceau de programme).
Il est maintenant temps de voir quelques autres opérateurs liés aux booléens et qui vous servirons souvent. Les symboles que nous allons voir sont des *opérateurs de comparaison* (+ ou / par exemple sont des *opérateurs* simples). Comme leur nom l'indique clairement, ils permettent de comparer deux expressions (souvent des nombres ou des booléens). Ils s'utilisent de la façon suivante :

```
« Expression 1 » opérateur « Expression 2 »
```

(pensez à « 1 + 2 » pour vous convaincre que cette écriture n'est pas si terrible qu'elle en a l'air)
Par *expression*, on entend n'importe quoi qui soit une donnée du programme ou quelque chose de calculable.
Le résultat de ces opérateurs, contrairement aux opérateurs +, -, ... est un booléen, et vaut donc **true** ou **false**. Il est donc primordial de comprendre que ce résultat peut être utilisé comme n'importe quel donnée booléenne (cf. exemples)
Voici la liste des opérateurs et leur signification :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| Opérateur | Signification | Fonctionnement |
|---|---|---|
| = | Est égal à | Renvoie true si les deux expressions sont égales |
| < | Est strictement inférieur à | Renvoie true si « Expression 1 » est strictement inférieure à « Expression 2 » |
| > | Est strictement supérieur à | Renvoie true si « Expression 1 » est strictement supérieure à « Expression 2 » |
| <= | Est inférieur ou égal à | Renvoie true si « Expression 1 » est inférieure ou égale à « Expression 2 » |
| >= | Est supérieur ou égal à | Renvoie true si « Expression 1 » est supérieure ou égale à « Expression 2 » |
| <> | Est différent de | Renvoie true si « Expression 1 » est différente de « Expression 2 » |

Exemples :

1. 0 = 1 vaut **false**
2. 1 < 1 vaut **false**
3. 1 >= 1 vaut **true**
4. 1 + 1 > 3 vaut **false** (les opérations sont prioritaires sur les opérateurs de comparaison)
5. (1 + 1 > 2) or (2 < 3) vaut **true**
6. **not** (10 - 22 > 0) et (10 - 22 <= 0) sont équivalents et valent **true**.
7. **not** (2 * 3 - 1 >= 5) = **true** vaut... **false**. (explication de ce dernier cas un peu tordu : 2 * 3 - 1 >= 5 est vrai, donc vaut **true**, la négation vaut donc **false**, et **false** = **true** est alors faux, donc vaut **false**).

Ces opérateurs permettront de faire des tests sur des données d'un programme, et de prendre des décisions en fonction du résultat des tests.

## V-A-4 - Types énumérés

Les types énumérés sont énormément utilisés par Delphi et l'inspecteur d'objets, mais existaient bien avant Delphi. Ils sont utiles lorsqu'une donnée ne doit pouvoir prendre qu'un certain nombre de valeurs, chacune ayant une signification particulière. Ils permettent de définir directement les valeurs possibles pour une donnée (de ce type). Ces valeurs sont données sous forme de noms, qui seront considérés ensuite comme des mots du langage Pascal. Exemple concret : vous voulez définir l'alignement horizontal d'un texte. 3 alignements sont possibles : gauche, centré, droite. Vous pouvez alors utiliser le type de donnée dit *type énuméré* nommé « TAlignment ». Une donnée de ce type peut avoir trois valeurs, qui sont comme des mots du langage Pascal, à savoir : taLeftJustify, taCenter et taRightJustify.
Vous n'avez absolument pas à savoir ce que sont en réalité ces trois expressions (nommés plutôt *identificateurs*). Pour donner un alignement centré à un texte, il faudra utiliser taCenter (et nous verrons où et comment faire cela dans très peu de temps). Il vous faudra cependant savoir que chacun de ces *identificateurs* possède une valeur *ordinale* positive ou nulle (nous reparlerons de cela au chapitre 5).
Il sera également possible pour vous de créer des types énumérés en donnant le nom du type (comme 'TAlignment') et en donnant les valeurs possibles pour les données de ce nouveau type. Vous verrez que cela est bien utile dans certaines situations mais ce n'est pas encore pour tout de suite.

## V-A-5 - Récapitulatif

Cette partie consacrée aux types de données les plus simples (pas forcément encore pour vous, désolé si c'est le cas) est terminée. Vous l'avez certainement trouvé ennuyeuse, et vous avez raison : elle l'est. Lorsque vous parcourrez


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

la suite de ce guide, il sera certainement intéressant pour vous de revenir à cette partie, qui vous semblera à ce moment bien moins abstraite. L'essentiel pour l'instant est qu'aucun des mots du langage Pascal qui suivent ne vous soit inconnu à partir de ce point :

```
Integer byte word
single currency
div mod
char string shortstring string[n]
boolean true false
and or xor not
```

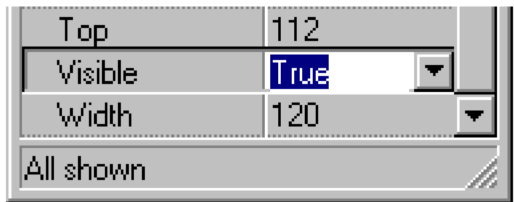
De même, la liste des symboles ci-dessous doit être connue de vous :
+ - * /
= < > <= >= <>
Si ce n'est pas le cas, essayez de revenir en arrière pour voir ce qui vous a échappé, sinon, vous pouvez avancer en toute confiance. La suite du chapitre est consacrée à des manipulations servant de mise en pratique pour vos connaissances toutes fraiches. Vous verrez alors que les booléens, les données de types énumérées, les chaînes et les nombres sont partout.

## V-B - Utilisations des types simples dans l'inspecteur d'objets

Place maintenant à quelques manipulations : retournez sous Delphi, ouvrez si besoin le projet PremierEssai et faites apparaître les propriétés de la seule fiche du projet dans l'inspecteur d'objets (si vous ne savez pas comment faire, relisez le paragraphe : **IV-E-4 L'inspecteur d'objets** où la manipulation est décrite en détail).
Comme nous l'avons déjà expliqué, les propriétés de la fiche sont des paramètres modifiable de celle-ci. Chacune de ces propriété est d'un type bien déterminé. Vous connaissez maintenant certains de ces types. Voici une liste de propriétés de la fiche qui sont de types connus par vous.

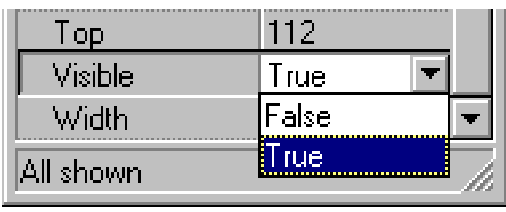
- La propriété 'width' (largeur de la fiche) déjà utilisée auparavant. Cette propriété est un nombre entier et son type est 'integer' (pourtant, vous constaterez que les valeurs négatives sont interdites. C'est dû à des mécanismes que vous n'avez pas à connaître pour l'instant, qui seront vus beaucoup plus tard dans le guide).
- La propriété 'caption' (texte de la barre de titre) est de type 'string'. Vous pouvez y taper n'importe quel texte, il y a prise en compte des majuscules et minuscules.
- La propriété 'visible' (visibilité) est de type booléen et détermine si un composant est visible ou non pendant l'exécution de l'application. Cette propriété est plus souvent employée avec autre chose que les fiches. Effectuez la manipulation suivante :

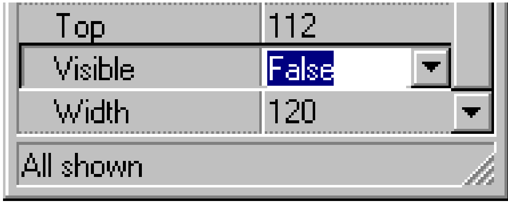
  1. Affichez les propriétés du bouton ('Action !').
     Sa propriété 'visible' devrait normalement être à 'true'.

     Lorsque vous lancez l'application, le bouton est visible.
  2. Changez maintenant cette propriété en 'false' en utilisant la liste déroulante :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

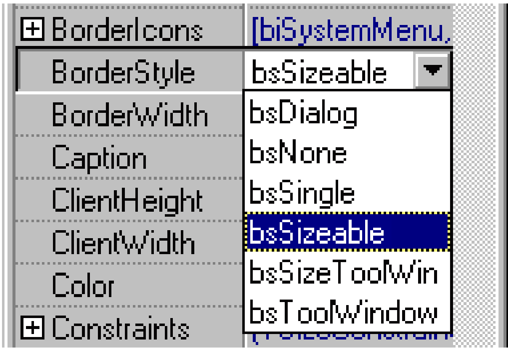



et lancez l'application : le bouton n'est pas visible.

3 Redonnez la valeur 'true' à la propriété 'visible' du bouton.

- La propriété 'BorderStyle' permet de définir le style de bordure de la fenêtre pendant l'exécution de l'application. Elle est de type énuméré et ses différentes valeurs sont accessibles dans l'inspecteur d'objets par une liste déroulante :




(Il est possible que vous n'ayez pas tout à fait la même liste d'expressions, mais certaines doivent y être, comme 'bsSizeable', 'bsDialog', 'bsNone' et 'bsSingle')
Le type de bordure de la fenêtre dépend de l'élément que vous sélectionnez ici (qui sera la valeur de la propriété 'BorderStyle').
Si vous souhaitez expérimenter les différents styles de bordures, rien de plus simple : sélectionnez le style désiré, et lancez l'application pour voir le résultat (notez qu'avec certains choix, la fiche ne peut plus être redimensionnée).

Ceci termine les manipulations dirigées. L'exercice ci-dessous vous donnera l'occasion de découvrir d'autres propriétés de la fiche et du bouton.

Exercice 3 : (voir la **solution** )
Les propriétés suivantes sont communes à la fiche et au bouton (chacun de ces deux composants en a un exemplaire). Essayez de trouver de quel type elles sont en utilisant l'inspecteur d'objets comme dans les manipulations ci-dessus (rappelez-vous que bien que les changements apportés à certaines propriétés sont directement visibles, d'autres requièrent de lancer l'application) :

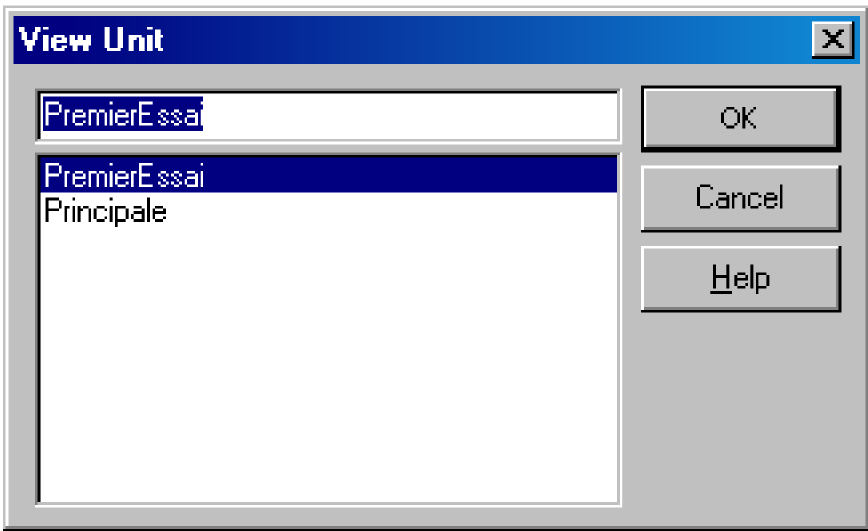
1 'Cursor' (pointeur utilisé pour la souris quand elle passe sur le composant)
2 'Height' (hauteur de la fiche)
3 'Hint' (texte de la bulle d'aide associée au composant)
4 'ShowHint' (décide si la bulle d'aide est montrée ou pas)


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## V-C - Structure d'un programme Pascal

Nous entrons enfin dans le vif du sujet. Cette partie est consacrée à la compréhension pour vous de la structure (de l'organisation) et non du contenu (du sens) de tout programme écrit en langage Pascal. Tout au long de cette partie, des exercices vous seront proposés afin de manipuler un peu le langage. Vous aurez également des occasions de mettre en pratique vos connaissances toutes fraîches sur les types de données.
Vous savez déjà, ou alors il est grand temps de savoir, qu'un logiciel qu'on crée avec Delphi passe par l'état de projet. Ce projet est constitué du fichier projet (DPR), d'unités (PAS) et de fiches (DFM). Le fichier projet étant en fait une unité particulière, nous nous attacherons d'abord à la structure de ce fichier, qui diffère de celle des unités. Nous nous intéresserons ensuite à la structure d'une unité (toutes les unités ont la même structure générale (en fait, unité désigne plutôt un fichier contenant une unité, c'est un abus de langage que nous nous permettrons désormais). La structure des fichiers DFM (fiches) ne sera pas vue car ce n'est pas du texte compréhensible qui y est inscrit (mais nous y reviendrons).
Le mieux pour suivre les explications données est de lancer Delphi et d'ouvrir le projet PremierEssai déjà commencé. Les seuls éléments du projet sont alors une unité (« Principale ») et une fiche (Form1). Pour voir le fichier projet (PremierEssai), deux possibilités :

1. Si elle est disponible, Utilisez la commande 'Source du projet' du menu 'Voir'.
2. Sinon, une autre méthode qui marche toujours et également avec les unités est de cliquer sur le menu 'Voir', puis 'Unités...' (La fenêtre qui s'affiche alors liste les unités du projet ainsi que le fichier-projet (Vous pourrez l'utiliser pour voir la ou les autres unités). Dans la liste qui s'affiche :




Double-cliquez sur PremierEssai. L'éditeur de code affiche alors du texte commençant par « program ... » (Note : vous voyez, en passant, que Delphi considère le fichier projet comme une unité puisqu'il est listé parmi les unités).

### V-C-1 - Structure d'un fichier projet

Le fichier projet est un des domaines réservés à Delphi, mais dont vous devez connaître l'existence et la structure. L'ensemble du texte du fichier est généré par Delphi. Dans l'état actuel, le texte du fichier projet devrait être (avec peut-être d'autres styles de caractères, mais un texte identique) :

```
program PremierEssai;

uses
  Forms,
  Principale in 'Principale.pas' {Form1};
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Servez-vous de ce texte comme illustration de ce qui suit.
Avant tout, ce qui est écrit entre accolades (texte vert en italique) est un *commentaire*. Ce qui est à l'intérieur de ces accolades est normalement ignoré par Delphi, à certaines exceptions bien précises près.
Encore une fois, le contenu du fichier projet est géré entièrement par Delphi, et une intervention de votre part, bien que possible et autorisée, risque de provoquer des erreurs. Il est assez rare d'avoir à modifier directement ce fichier, et plus avantageux de laisser Delphi se débrouiller tout seul.
Un fichier projet est un fichier (donc un morceau de programme, donc par abus d'écriture un programme) écrit en langage Pascal. Un fichier projet est constitué de blocs de texte. Chaque bloc possède sa propre syntaxe. Les blocs sont terminés par un point-virgule (;) sauf le dernier qui est terminé par un point (.). Un fichier projet commence par le mot réservé **program** et se termine par le point final du dernier bloc.
Le premier bloc d'un fichier projet consiste en la déclaration de son nom. Ceci est fait par l'utilisation du mot Pascal **program**, suivi d'un espace (l'espace est un séparateur en Pascal, comme en français. C'est le séparateur minimum. On peut y ajouter au besoin d'autres espaces, des sauts de lignes, des tabulations, bien utiles pour bien présenter le texte) puis du nom du programme, et enfin d'un point-virgule séparant le bloc du suivant. Le nom doit être identique au nom du fichier .DPR, à l'exception de « .dpr » qui doit être omis.
Exemple : Dans notre cas, le fichier projet se nomme PremierEssai.dpr. La première ligne du fichier projet comporte donc comme nom de programme « PremierEssai ».
Le second bloc d'un fichier projet est un bloc spécial qu'on retrouvera dans les unités, sous une forme simplifiée. Ce bloc sert de connexion entre les différentes unités et le fichier-projet.
Le bloc commence par le mot Pascal **uses** (en français : « utilise ») et donne ensuite une liste d'éléments, séparés par des virgules. Le bloc se termine par un désormais classique point-virgule. Chaque élément de cette liste comporte au moins le nom d'une unité (l'alter-ego du nom de programme spécifié après **program** mais pour une unité), qui est le nom du fichier dans lequel elle est stockée, privé de « .pas ». Cette unité peut faire partie du projet ou pas. Si l'élément s'arrête à la mention du nom, c'est que l'unité ne fait pas partie du projet.
Exemple : l'unité 'Forms' est mentionnée dans le fichier projet, mais ne fait pas partie du projet. C'est une des nombreuses unités fournies par Delphi et utilisables à volonté.
Les éléments qui font référence aux unités du projet comportent en plus le mot Pascal **in** suivi d'une chaîne de caractères (entre simple quotes, vous vous en rappelez ?) contenant le nom d'un fichier (le fichier qui contient l'unité). Si l'élément se termine ici, c'est une unité sans fiche. Peut ensuite venir entre accolades (donc en commentaire, mais ceux-ci ne sont pas vraiment ignorés par Delphi du fait de leur présence dans le bloc **uses** du fichier-projet) un autre élément qui fait référence à la fiche et qui n'est évidemment présent que si celle-ci existe (dans le cas d'une unité non associée à une fiche, cet élément n'apparaît donc pas) : c'est le nom de la fiche (C'est la propriété 'name' de la fiche).
Exemple : l'unité stockée dans Principale.pas a pour nom « Principale », elle est associée à une fiche dont le nom est Form1 (affichez les propriétés de la fiche dans l'inspecteur d'objet et regardez la propriété 'name' pour vous en convaincre).
Vient ensuite une ligne indispensable comportant un commentaire spécial que nous n'expliquerons pas ici : « {$R *.RES} » (lisez l'approfondissement ci-dessous si vous voulez quand même en savoir plus). Ce n'est pas un bloc puisque les commentaires sont ignorés par Delphi. Ce commentaire est seulement un commentaire spécial que vous pouvez ignorer (vous devez commencer à vous demander si les commentaires sont vraiment des commentaires. C'est effectivement le cas dans 99% des cas, sauf entre autres pour ceux commencant par le symbole '$'. Rassurez-vous, lorsqu'on se contente d'écrire du texte simple, il n'y a aucun problème).

***Approfondissement :***
*Dans le répertoire dans lequel vous avez enregistré le fichier projet (et normalement le projet entier), se trouve normalement un fichier nommé 'PremierEssai.res'. Ce fichier est un fichier de ressources. Anciennement, ces fichiers de ressources contenaient des ressources compilées telles des fiches, des images, des icônes, ...*
*Delphi utilise ce fichier pour stocker en général un unique élément : l'icône de l'application. Si vous souhaitez vous en assurer, vous pouvez lancer l'éditeur d'images de Delphi (menu Outils/ Editeur d'images) et ouvrez le fichier. Ce sera également une méthode pour changer cette icône.*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

*La ligne {$R *.RES} est une directive de compilation, qui est destinée non pas à l'ordinateur ni à Delphi mais au compilateur (le programme qui compile le projet et le transforme en application). Cette commande est exécutée par le compilateur et ne sera pas incluse dans l'application. La commande demande d'inclure tous les fichiers .RES du répertoire du projet dans la compilation. C'est ainsi que l'icône de l'application est incluse dans le fichier .EXE final. C'est aussi comme cela que nous pourrons ajouter d'autres éléments extérieurs, mais cela est pour beaucoup plus tard.*
*Souvenez vous de ces explications lorsque, dans les unités, vous verrez la ligne {$R *.DFM}. Cette ligne a le même rôle, pour les fichiers .DFM, donc les fiches (en fait, les fichiers contenant les fiches). C'est ainsi que les fiches sont incluses dans la compilation.*

Le reste du fichier, avant « end. », est la partie (le bloc) la plus importante du fichier projet : c'est celle qui permet l'exécution de l'application. Cette partie commence par le mot réservé (mot Pascal) **begin** (en français : « début ») et se termine par le mot réservé **end**. Ce bloc est constitué lui-même comme le fichier projet, à savoir de blocs. Chaque bloc est séparé des autres par un point-virgule (omis avant le premier et après le dernier bloc). Chacun de ces petits blocs est une *instruction* du programme.
Exemple : dans notre cas, il y a trois instructions qui sont :

1. Application.Initialize
2. Application.CreateForm(TForm1, Form1)
3. Application.Run

La signification de ces instructions est hors de notre propos. Concentrez-vous simplement sur ce terme : *instruction*. Il signifie que chacun des 3 morceaux de texte ci-dessus est un ordre en Pascal compréhensible par l'ordinateur (après compilation, mais c'est jouer sur les mots) et qui sera donc exécuté par ce dernier. En fait, ces instructions sont parmi les toutes premières qui seront exécutées par l'ordinateur au démarrage de l'application.
Voilà, désormais, vous connaissez la structure d'un fichier-projet. En gros, vous pouvez retenir qu'un fichier-projet est organisé comme suit :

- bloc de déclaration de nom :
  **program** *nom_du_programme*;
- bloc d'utilisation d'unités :
  **uses** *liste_d_unites*;
- bloc d'instructions :
  **begin**
  *instructions*
  **end.**

## V-C-2 - Structure d'une unité

Etant donné que vous aurez peu à modifier le fichier projet, vous passerez le plus clair de votre temps à écrire du Pascal dans des unités. Pour voir la seule unité actuelle du projet PremierEssai, effectuez la même manipulation que pour le fichier-projet, mais en sélectionnant cette fois 'Principale'. Le texte écrit dans la fenêtre d'édition devrait être :

```
unit Principale;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

Le squelette de base de l'unité (et de toute unité, mis à part le nom qui évidemment, changera) est :

```
unit Principale;

interface

implementation

end.
```

La ligne « **unit** Principale; » est le premier bloc de l'unité. Il permet, comme dans un fichier-projet, de donner le nom de l'unité. Ce nom est, comme pour le fichier-projet, le nom du fichier privé de l'extension '.pas'.

*Important : Ce nom ne doit pas être modifié par vous directement, utilisez plutôt la commande Fichier/Enregistrer sous... des menus pour changer le nom du fichier .PAS contenant une unité, ce qui aura pour conséquence de mettre à jour le nom d'unité.*

Les mots réservés **interface**, **implementation**, et **end** délimitent deux parties : le premier est contenu entre **interface** et **implementation** et se nomme tout naturellement l'*interface* de l'unité. La deuxième partie est entre **implementation** et **end** et se nomme l'*implémentation* de l'unité. Un point final termine l'unité comme dans un fichier-projet.

*Remarque : Bien que le fichier-projet soit considéré par Delphi comme une unité (mais ce n'en est pas une, pour vous dire enfin toute la vérité), il ne possède pas ces deux parties interface et implémentation qui sont réservées aux unités.*

L'interface de l'unité 'Principale' est donc :

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Et l'implémentation est :

```
{$R *.DFM}
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce qui est assez court mais qui aura une très nette tendance à s'allonger.
Vous vous souvenez du mot réservé **uses** ? Il permet, je le rappelle, d'accéder à une autre unité. En réalité, seule l'*interface* d'une unité est alors accessible, mais jamais l'*implémentation* (et il n'y a absolument aucun moyen d'accèder à l'*implémentation* d'une autre unité, et ça ne servirait d'ailleurs à rien, nous le verrons).
On pourrait faire ici l'analogie avec une unité centrale d'ordinateur : l'interface est alors l'extérieur du boîtier, et l'implémentation est l'intérieur de ce même boîtier. Lorsque vous utilisez une unité centrale, vous utilisez les quelques boutons et afficheurs (diodes) qui se trouvent en face avant. Vous utilisez également les prises qui sont à l'arrière du boîtier pour brancher divers câbles. Vous utilisez en quelque sorte une interface, sans vraiment vous soucier de l'intérieur du boîtier. Vous pouvez même ignorer totalement ce qu'il y a à l'intérieur et jamais cela ne vous posera de problème : vous êtes l'utilisateur de l'ordinateur et non son créateur.
C'est le même principe avec une unité : seule l'interface vous intéresse, vue de l'extérieur, l'implémentation n'a plus d'importance vu de l'extérieur de l'unité. Mais n'allez pas croire que l'implémentation n'est pas importante ! En effet, comment croyez-vous que votre ordinateur fonctionnerait si votre unité centrale était complètement vide ? La réponse est simple, il ne fonctionnerait pas. Vous auriez beau appuyez sur le bouton On/Off, il ne se passerait rien car rien n'aurait été prévu dans l'implémentation (l'intérieur du boitier) pour réagir à vos ordres (pas de fils branchés sur l'interrupteur et reliés à un quelconque circuit électrique).
L'implémentation regroupe tout ce qui n'a pas à être vu de l'extérieur. Vous y mettrez la majorité du texte Pascal, ne laissant dans l'interface que le strict nécessaire depuis l'extérieur.
Chacune de ces deux parties a un rôle particulier. On peut dire, sans encore rentrer dans les détails, que l'interface est la partie *déclarative* de l'unité tandis que l'implémentation est plutôt la partie *exécutive*. Pour s'exprimer un peu plus clairement, l'interface contiendra tout ce qui devra être connu de l'extérieur : on y *déclarera* ce qu'on veut mettre à la disposition de l'extérieur. L'implémentation, quand à elle, contiendra majoritairement des *instructions* (comme celles présentes à la fin du fichier-projet) organisées en groupes.
L'interface et l'implémentation possède chacune un bloc unique et optionnel désormais connu de vous : **uses**. Il a le même rôle que pour le fichier-projet, à savoir faire appel à d'autres unités. La syntaxe de ce bloc est très simple par rapport à celle employée dans le fichier-projet : le mot-clé **uses** est suivi par un espace, puis par une liste de noms d'unités, séparés par des virgules. Un point-virgule termine le bloc. On dira, lorsque le nom d'une unité est présent dans le bloc **uses**, que cette unité est utilisée.
Le bloc **uses** est valable au moins pour toute la partie dans laquelle il est inclus, c'est-à-dire que les unités listées dans le **uses** de l'interface (resp. de l'implémentation) sont accessibles partout dans l'interface et dans l'implémentation (resp. l'implémentation seulement). Le tableau ci-dessous résume cela :

| bloc uses de la partie | parties dans lesquelles ce bloc agit |
|---|---|
| Interface | Interface, Implémentation |
| Implémentation | Implémentation |

Quelques règles sont à connaître dans l'utilisation des unités :

1. Une unité peut être utilisée par un nombre quelconque d'autres unités.
2. Une unité ne peut être utilisée qu'une fois dans une autre unité, c'est-à-dire qu'elle sera utilisée dans l'interface ou dans l'implémentation, mais pas dans les deux.
3. Les utilisations d'unités depuis l'interface des unités ne peuvent pas être circulaires.Pour expliquer ce concept assez tordu mais imparable, supposons que nous avons trois unités nommées unit1, unit2 et unit3 (ces noms sont destinés à simplifier les explications, ils ne sont pas à utiliser car non descriptifs du contenu de l'unité). Si l'unité unit1 utilise l'unité unit2 dans son interface, si l'unité unit2 utilise l'unité unit3 dans son interface, alors l'unité unit3 n'aura pas le droit d'utiliser l'unité unit1 (et pas l'unité unit2 non plus) dans son interface, car ceci formerait un cycle.
4. Il est toujours possible d'utiliser une unité dans l'implémentation d'une unité (en respectant toutefois la règle 2). Ceci permet de tempérer la règle précédente : dans le dernier exemple, les unités unit1 et unit2 pourront quand même être utilisées par l'unité unit3, non pas dans son interface mais quand même dans son implémentation.

Même si ces règles peuvent paraître compliquées, elle le seront moins à l'emploi. Le coté contraignant ne devrait pas dans l'absolu vous déranger puisque vos programmes devront être écrits en tenant compte de ces règles.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## V-D - Constantes et Variables

Ce paragraphe est consacré à l'étude des fameuses « données » dont on n'a cessé de parler presque énigmatiquement dans les parties précédentes. En Pascal, ces données peuvent être des constantes ou des variables.
Une constante est un nom que vous associez une fois pour toute à une valeur. Cette valeur ne pourra pas être modifiée pendant l'exécution de l'application. Ce nom devient alors un mot Pascal reconnu par Delphi.
Une variable est un nom que vous donnez à une donnée d'un type bien défini. Ainsi vous pouvez définir des variables de chaque type déjà vu dans ce guide, par exemple 'byte', 'single', 'char', 'string[10]', etc. La valeur d'une variable peut, comme son nom l'indique clairement, changer au cours de l'exécution de l'application.
Les paragraphes qui suivent décrivent l'usage de ces deux notions.

### V-D-1 - Préliminaire : les identificateurs

Les noms (techniquement : les chaînes de caractères utilisés pour les nommer) de constantes, de variables, d'unités, de programmes, de types sont ce qu'on appelle des *identificateurs*. Un identificateur répond aux exigences suivantes, imposées par le langage Pascal :

- C'est une suite de caractères, pris parmi les lettres majuscules ou minuscules, les chiffres, le blanc souligné ( _ ) (en dessous du 8 sur votre clavier). Par contre, les espaces, le tiret ( - ), les accents, et tout le reste sont interdits.
- Un nombre de caractères maximum raisonnable est 50 (le nombre maximal a augmenté avec les différentes versions de Delphi).
- Le premier caractère ne peut pas être un chiffre.
- Les majuscules et minuscules sont vues comme les mêmes lettres. Il est cependant recommandé d'utiliser les minuscules autant que possible et les majuscules seulement pour les premières lettres des mots. Ainsi par exemple 'PremierEssai' est un identificateur valable et descriptif.

Exercice 4 : (voir la **solution**)
Parmi les propositions ci-dessous, lesquelles sont des identificateurs valables, et pour ceux qui n'en sont pas, pourquoi ?

1 test1
2 1_test
3 test 1
4 test-1
5 test_1
6 TEST_2

Pour les exemples qui vont suivre, vous pourrez utiliser Delphi et taper le texte indiqué juste avant le mot **implementation** dans l'unité 'Principale' :

```
...
  end;

var
  Form1: TForm1;

{ <-- Ici }
implementation

{$R *.DFM}
...
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Vous ne pourrez pas avoir de résultat visible, mais vous pouvez vérifier que vous avez bien tapé les textes en compilant le projet (Menu Exécuter / Compiler, ou raccourci clavier : Ctrl + F9). Les erreurs éventuelles vous seront alors indiquées par un ou plusieurs messages d'erreur.

## V-D-2 - Constantes

Les constantes sont une possibilité très intéressante de la programmation en Pascal. Elles permettent, sans aucune programmation, de faire correspondre un *identificateur* et une *valeur*. Cette valeur peut être de beaucoup de types (un seul à la fois, quand même) mais il faudra parfois indiquer ce type.
Les constantes se déclarent dans un bloc adéquat. Ce bloc peut être positionné, pour l'instant, dans l'interface ou dans l'implémentation d'une unité en dehors des autres blocs. La structure d'un tel bloc est des plus simple : on débute le bloc par le mot réservé **const** (le bloc n'a pas de marque de fin, c'est le début d'un autre bloc ou la fin de l'interface ou de l'implémentation qui terminent le bloc).

```
const
```

Vient ensuite une liste aussi longue qu'on le souhaite de déclarations de constantes. Une déclaration de constante a la forme :
*Identificateur = Valeur;*
Où *Identificateur* est un identificateur (non utilisé ailleurs) et *Valeur* est une valeur de type nombre, chaîne ou booléen (d'autres types seront utilisables par la suite au prix d'une syntaxe adaptée).
Exemple :

```
const
  LargeurParDefaut = 640;
  NomLogiciel = 'Premier Essai';
  MontreEcranDemarrage = True;
  Touche01 = 'z';
```

Les constantes de types plus complexes devront plutôt utiliser la syntaxe suivante :
*Identificateur : Type = Valeur;*
Où Type est le type de la constante, et Valeur une valeur acceptable pour ce type.
Exemple :

```
const
  AlignementParDefaut: TAlignment = taCenter;
```

Une constante peut également être déterminée à partir de tout ce dont on connaît la valeur à l'endroit où on déclare cette constante. En pratique, on peut utiliser les constantes déclarées plus haut dans l'unité, et les constantes des unités utilisées (listées dans le bloc uses).
Exemple :

```
const
  Valeur1 = 187; {187}
  Valeur2 = Valeur1 + 3; {190}
  Valeur3 = Valeur2 div 2; {95}
  Condition1 = (Valeur3 > 90); {True}
  ChBonj = 'Bonjour';
  ChBonjM = ChBonj + ' ' + 'Monsieur';
```

*Rappel : les textes entre accolades sont des commentaires ignorés lors de la compilation.*

*Note : Plusieurs blocs de déclaration de constantes peuvent se suivre sans problème.*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La zone d'effet de la constante (la zone où la constante est accessible) commence juste après la déclaration et se termine à la fin de l'unité. De plus, lorsqu'une constante est déclarée dans l'interface d'une unité, elle est également accessible à toute autre unité utilisant la première. Les constantes peuvent également être déclarées dans le fichier-projet entre le bloc **uses** et le bloc d'instructions.

### V-D-3 - Variables

Une variable est un *identificateur* associé à un *type* donné. La valeur de cette variable pourra changer au cours de l'exécution de l'application. Les valeurs autorisées pour cette variable, ainsi que les opérations possibles avec et sur cette variable dépendent exclusivement du type de la variable et de l'endroit où elle a été déclarée.
Car comme les constantes, une variable se déclare dans un bloc spécifique situable pour l'instant dans l'interface ou dans l'implémentation d'une unité, en dehors de tout autre bloc. Il est également autorisé de déclarer des variables dans le fichier-projet entre le bloc **uses** et le bloc d'instructions (retenez que les endroits possibles pour les déclaration de constantes et de variables sont les mêmes).
Un bloc de déclarations de variables est de la forme suivante :

```
var  Déclaration de variables; Déclaration de variables; ... Déclaration de variables;
```

Que la description ci-dessus ne vous leurre pas, il peut très bien n'y avoir qu'une déclaration de variables par bloc. Chacun de ces blocs « Déclaration de variables » sert à déclarer une ou plusieurs variables d'un même type, et est de la forme :

```
Identificateur_1, identificateur_2, ..., identificateur_n : type;
```

Chaque déclaration de variables peut ne déclarer qu'une variable, ou plusieurs, mais du même type. *Identificateur_n* désigne toujours un identificateur non utilisé ailleurs. Lorsqu'on veut déclarer des variables de types différents, on utilise plusieurs blocs de déclaration.
Les exemples ci-dessous donnent l'éventail des possibilités qui vous sont offertes :

```
var
  A: integer;

var
  N1, N2: single;

var
  Test: Boolean;
  Indx: Integer;
  S1, S2: string;
```

L'intérêt des variables est énorme : il nous sera possible de stocker un résultat de calcul dans une variable, de réagir suivant la valeur d'une autre, d'en additionner deux et de stocker le résultat dans une autre, ... les possibilités sont illimitées.
En dehors des variables et des constantes que vous déclarez, rappelez-vous bien que vous aurez accès aux constantes et variables des unités que vous utiliserez (bloc **uses**). Bon nombre de constantes et de variables sont ainsi proposées et il suffit d'utiliser l'unité les contenant pour y avoir accès. Nous illustrerons cela dans un prochain exemple.

## V-E - Conclusion

Cette première et longue partie sur le langage Pascal est terminée. Vous y avez appris les rudiments du langage : à savoir la structure des unités et du fichier-projet, les liaisons possibles entres ces éléments. Vous avez également appris ce que seront concrètement les données utilisées dans vos programmes.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Le prochain chapitre est consacré à l'étude de deux notions importantes du langage Pascal : les procédures et les fonctions. Ces deux notions seront ensuite utilisées activement dans le guide car les procédudes et les fonctions seront partout dans les programmes écrits en Pascal.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# VI - Procédures et Fonctions

Jusqu'ici, nous avons étudié la structure des programmes Pascal ainsi que les déclarations de constantes et variables. Ces notions sont fondamentales mais nous n'avons pas encore vu une miette (jusqu'à maintenant) des fameuses *instructions* en Pascal dont nous parlions au début du guide.
Dans le langage Pascal, les instructions doivent être regroupés en blocs nommés *procédures* et *fonctions*, similaires aux blocs déjà connus de vous comme le bloc **uses** ou le bloc **var**. Un aspect de la philpsophie du langage Pascal est en effet que toute tâche complexe peut être découpée en tâches élémentaires. La tâche complexe, c'est l'application, tandis que les tâches élémentaires, ce seront les procédures et les fonctions. Chaque procédure ou fonction effectuera un travail particulier et bien ciblé (par exemple, répondre au clic sur un bouton).
Les procédures et les fonctions seront de ce fait les endroits privilégiés où nous écrirons les instructions en Pascal. Une instruction permettra l'exécution (on dira aussi l'appel) d'une procédure ou d'une fonction (des instructions contenues dedans). Il existe d'autres endroits que les procédures ou les fonctions où les instructions peuvent être écrites, mais chacun de ces endroits à une spécificité.
Le seul connu de vous actuellement est entre le mot **begin** et le mot **end** du fichier-projet. Cet endroit est réservé aux instructions qui doivent être exécutées au tout début de l'application, avant tout le reste.
Vous voyez maintenant l'intérêt d'étudier les procédures et les fonctions : il n'y a qu'après cela que nous pourrons réellement commencer à programmer en Pascal. Il serait en effet impossible d'utiliser seulement le fichier-projet.

*Note à ceux qui connaissent : La syntaxe des procédures et des fonctions va être présentée ici dans sa forme de base. Les diverses variantes (conventions d'appel, paramètres variables ou constants, surcharges, paramètres ouverts) seront présentées en temps opportun mais bien plus tard dans le guide. L'objectif de ce guide n'est pas de recopier l'aide en ligne de Delphi (qui ne vaut rien au plan pédagogique) qui donne d'un coup toutes les options, mais plutôt de proposer une méthode progressive qui laisse le temps de s'habituer à une version de base pour y greffer petit à petit d'autres possibilités.*

Nous allons étudier dans un premier temps les procédures, puis les fonctions, mais sachez dés à présent que les deux notions, du point de vue syntaxique, sont assez proches, et on se permettra de parler de fonction avant de les avoir vues car les fonctions sont en quelque sorte des procédures évoluées. La fin du chapitre sera consacrée à une longue manipulation avec Delphi pour vous permettre de vous exercer tout de suite. Entre temps, hélas, il faudra freiner votre envie de tester vos essais, car les connaissances nécessaires ne sont pas encore tout-à-fait disponibles, mais c'est tout proche.

## VI-A - Procédures

Les procédures sont en fait constituées par un ou deux blocs de texte pascal, dont le premier est optionnel et permet de *déclarer* la procédure. Nous reviendrons plus loin sur ce premier bloc qui n'a pas toujours d'utilité. Le second bloc, lui, est obligatoire et constitue le *corps* de la procédure (on dit également *son implémentation*). Les deux endroits possibles pour écrire ce bloc sont :

- Dans le fichier-projet, entre le bloc uses et le bloc d'instructions :

```
...
{$R *.RES}

{ <-- Ici }

begin
  Application.Initialize;
...
```

- Dans l'implémentation d'une unité.

Ce bloc doit être écrit à l'extérieur des blocs de déclaration comme **uses**, **var** et **const**. Par contre, il sera possible d'écrire une procédure à l'intérieur d'une autre procédure. Ceci ne sera pas souvent utile, mais je tâcherai de vous trouver un exemple pertinent dans la suite du guide.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Passons maintenant à l'aspect pratique : la syntaxe du bloc.

```
procedure identificateur [(paramètres)];
[déclarations locales]
begin
  [instructions]
end;
```

(Cette présentation sous forme de spécifications est affreuse, j'en conviens volontiers, mais c'est à mon sens le moins mauvais moyen de commencer)
Le bloc débute par le mot Pascal réservé **procedure** qui définit clairement le bloc comme étant une procédure. Le second mot est un identificateur qui donne le *nom* de la procédure. Ce nom sera utilisé dans l'instruction qui commandera l'exécution de la procédure.
Vient ensuite, éventuellement, une liste de *paramètres* entre parenthèses. Les paramètres sont des données intervenant dans la procédure, mais dont vous ne pouvez connaître la valeur : les paramètres seront renseignés à chaque exécution de la procédure. Considérons par exemple que vous avez une procédure qui dessine un cercle : les paramètres seront alors la position du centre et le rayon. Vous ne connaissez pas la valeur de ces paramètres mais vous allez quand même les employer dans la procédure.
La liste de paramètres, si elle est présente, est constituée d'au moins un paramètre, mais peut en contenir plus. Chaque paramètre est séparé des autres par un point-virgule. La liste ne comporte pas de point-virgule à son début ni à sa fin. Voici la syntaxe d'une liste de paramètres :
*paramètre 1[; paramètre 2] ... [; paramètre n]*
Chaque paramètre est de la forme suivante :
*identificateur : type*
Si plusieurs paramètres sont du même type, il est possible de les regrouper comme suit :
*identificateur1, identificateur2 : type*
Mais c'en est assez de toutes ces spécifications : voici des exemples (juste pour la première ligne).

```
procedure Explosion1;

procedure Explosion2 (Force: Integer);

procedure Explosion3 (Force, Vitesse: Integer);

procedure Explosion4 (Force, Vitesse: Integer; Son: Boolean);

procedure Explosion5 (Force, Vitesse: Integer; Son: Boolean;
  Forme: TFormeExplosion);
```

Ces 5 exemples, outre leur coté un peu sordide, montrent les différentes possibilités qui existent pour les paramètres : aucun, un seul, plusieurs, avec des regroupements, de différents types. Le 5ième exemple montre qu'on peut couper cette première ligne en plusieurs plus courtes à des fins de présentation, sans toutefois tomber dans l'excés.
Passons maintenant à la deuxième ligne : les *déclarations locales*. Ce sont des déclarations de constantes (blocs **const**) de variables (blocs **var**) ou d'autres procédures. L'ensemble de ces déclarations locales ne seront accessibles, contrairement aux déclarations normales, qu'à l'intérieur de la procédure. On parlera alors de variables, de constantes et de procédures locales.
Exemple :

```
procedure Cercle (X, Y: Integer; Rayon: Word);
var
  Angle: Single;
begin
end;
```

Dans l'exemple ci-dessus, une seule déclaration locale a été faite : celle d'une variable 'Angle' de type 'single'. Cette variable, qui servira pendant le tracé du cercle, ne doit pas être déclarée à l'extérieur de la procédure dans un bloc


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

**var** car cette variable n'a d'utilité qu'à l'intérieur de la procédure. Voici cependant un extrait de code qui déclarerait 'Angle' à l'extérieur de la procédure :

```
var
  Angle: Single;

procedure Cercle (X, Y: Integer; Rayon: Word);
begin
end;
```

Lorsqu'une variable (resp. une constante) est déclarée hors de toute procédure (ou fonction), elle est appelée variable *(resp. constante) globale*. Lorsque, par contre, elle est déclarée à l'intérieur d'une procédure ou d'une fonction, on l'appelle variable *(resp. constante) locale*.
Il faut éviter au maximum les variables globales (mais pas les constantes) : elles sont non seulement contraires à la philosophie du langage Pascal mais monopolisent également davantage de ressources que les variables locales car restent en mémoire en permanence tandis que les variables locales n'existent en mémoire que pendant que la procédure est exécutée. Pensez pour vous convaincre que l'effort en vaut la peine à ces programmes que vous lancez sur votre ordinateur et qui en ralentissent énormément le fonctionnement.
Passons maintenant au reste de la procédure : les mots Pascal réservés **begin** et **end** (sans le point-virgule final) délimitent le début et la fin des instructions contenues dans la procédure. L'ensemble des trois (begin, instructions et end) constitue ce qu'on appelle un *bloc d'instructions* (souvenez-vous de ce terme, il sera utilisé plus tard pour désigner ces trois éléments).
Le terme *instructions* désigne une suite d'instructions, terminées chacune par un point-virgule. Les puristes vous diront que la dernière instruction ne doit pas avoir de point-virgule final (et ils ont raison) mais ce dernier point-virgule avant le mot réservé **end** est toutefois autorisé et est en pratique très répandu, jusqu'à être présent dans l'aide en ligne de Delphi. Vous pourrez donc vous permettre d'écrire ce dernier point-virgule sans aucun remords.
Quant aux instructions proprement dites, elles seront vues progressivement dans la suite du guide, car elles peuvent prendre bon nombre de formes et requièrent des connaissances spécifiques.
Les procédures ne sont pas acceptées dans l'interface des unités, pourtant, seule l'interface est accessible depuis l'extérieur. Comment accèder alors à ces procédures (et fonctions) depuis l'extérieur ? De même, une procédure ne peut être utilisée que par la partie de l'unité qui la suit, comment dans ce cas faire si deux procédures doivent s'appeler mutuellement ?
la réponse à ces deux questions est dans le premier bloc optionnel dont nous parlions au début de la partie. Ce bloc, c'est la *déclaration* de la procédure. Cette déclaration est constituée par la première ligne du deuxième bloc (jusqu'au premier point-virgule inclus), avec quelques exceptions que nous verrons quand l'occasion se présentera. La déclaration d'une procédure (resp. d'une fonction) précède toujours cette procedure (resp. cette fonction) puisque la déclaration doit se trouver dans l'interface de l'unité. La procédure (resp. la fonction) est utilisable tout de suite après sa déclaration. Du fait de la présence de la déclaration dans l'interface, la procédure (resp. la fonction) devient accessible depuis l'extérieur de l'unité.
Voici un exemple complet (sans instructions dans la procédure) :

```
unit test;

interface

procedure Cercle (X, Y: Integer; Rayon: Word);

implementation

procedure Cercle (X, Y: Integer; Rayon: Word);
var
  Angle: Single;
begin
end;

end.
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## VI-B - Fonctions

Comme on l'a dit au début du chapitre, les fonctions sont assez proches des procédures. En fait, une fonction est une procédure avec une possibilité de plus : celle de renvoyer un résultat final. La syntaxe change alors un peu, mais seule la première ligne change et devient :

```
function identificateur [(paramètres)]: type_resultat;
```

'type_resultat' indique, dans la définition ci-dessus, le type du résultat de la fonction, c'est-à-dire le type d'une variable utilisable dans chaque fonction, bien que non déclarée : result. result est en effet utilisable au même titre qu'une variable, mis à part cette particularité unique de n'être pas déclaré. Ceci amène une restricton : l'identificateur 'result' est réservé à Delphi, vous n'y avez pas droit. La valeur de result après que la dernière instruction de la fonction ait été exécutée devient le résultat de la fonction. Vous ne voyez peut-être pas encore bien comment cela peut marcher : c'est un peu normal, lisez la suite et vous comprendrez tout.

## VI-C - Premières instructions en Pascal

### VI-C-1 - Affectations

Pour fixer le résultat d'une fonction, il va nous falloir donner une valeur à une variable (et cette variable sera 'result'). Cela se fait au moyen (je vous le donne en mille...) d'une instruction. Cette instruction a un nom : c'est une *affectation*. Sa syntaxe est la suivante :
*variable := valeur*

*Attention :* *le point-virgule final a ici été omis car les instructions sont séparées par ces point-virgules : ces derniers sont considérés comme ne faisant pas partie des instructions.*

Voici un exemple qui donne un résultat (fixe) à une fonction :

```
function TauxEuro: Single;
begin
  Result := 6.55957;
end;
```

Parmi les choses à remarquer dans le listing ci-dessus, le mot clé **function** qui débute la fonction. Le type de résultat a été fixé à 'single' car le taux de l'euro n'est pas un nombre entier. La variable 'Result', accessible puisque l'on est dans une fonction, est par conséquent de type 'single'.
La seule instruction de la fonction est une affectation : elle fixe la valeur de 'Result' au taux de l'euro, qui est un nombre à virgule.
Cet exemple n'est hélas pas très heureux (mais je suis bien trop fatigué pour vous en chercher un autre), car on aurait pu se servir d'une constante, ce qui serait revenu au même :

```
const
  TauxEuro = 6.55957;
```

Mais le terme 'valeur' dans la structure de l'affectation peut être beaucoup de choses, parmi lesquelles :

- Une constante
- Une variable
- Un paramètre (dans une fonction ou une procédure)
- Le résultat d'une fonction


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Une valeur déterminée à partir des éléments ci-dessus. Lorsqu'une instruction contient une affectation, cette affectation est exécutée en dernier, ce qui permet d'effectuer des calculs, par exemple, et d'affecter le résultat à une variable.

Voici un exemple illustrant cela :

```
unit test;

interface

function AireDisque(Rayon: Single): Single;

implementation

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

end.
```

Ce petit exemple illustre diverses choses :

- La fonction est déclarée dans l'interface, pour pouvoir être utilisée de l'extérieur.
- La fonction a un unique paramètre : 'rayon', qui est utilisé dans le calcul de l'aire.
- La fonction fait également appel à une fonction du même genre que celle qui est écrite ci-dessus : pi. C'est bien une fonction qui est appelée mais nous en reparlerons dans le paragraphe suivant.
- Ce qui est important, c'est que le résultat ('result') est calculé directement à partie d'un calcul : la multiplication d'un paramètre (dont on ne connait pas la valeur mais ceci ne nous gène pas puisque c'est le justement le but de la fonction que de calculer l'aire d'un disque en fonction de son rayon) et du résultat d'une fonction (car PI renvoie évidemment 3.14159...). L'opération est effectuée d'abord et le résultat de cette opération est mis dans 'result'.

*Remarque : La fonction PI renvoie un nombre de type 'extended' pour donner un nombre maximal de décimales de PI. C'est le type le plus large du calcul, donc le résultat des multiplication est de type 'extended'. Pourtant, 'result' est de type 'single', qui est beaucoup moins large que 'extended'. Une conversion implicite a eu lieu pendant l'affectation : ce qui ne pouvait pas être stocké dans 'result' a tout simplement été perdu, mais ce n'est pas grave : qu'aurions-nous fait d'un aire avec 20 décimales ?*

## VI-C-2 - Appels de procédures et de fonctions

Nous venons d'apprendre à écrire des procédures et des fonctions, et même à donner un résultat aux fonctions. Mais ceci ne sert à rien si nous ne savons pas faire appel à ces procédures et fonctions (les exécuter). L'appel d'une procédure ou d'une fonction est de la forme :
*nom_de_procedure [(valeurs_des_parametres)]*
'nom_de_procedure' est le nom de la procédure ou de la fonction que l'on veut exécuter. Si cette procédure ou cette fonction a des paramètres, il faudra fournir des valeurs de type correct et dans l'ordre de déclaration des paramètres. Les valeurs des paramètres sont données entre parenthèses, et les valeurs sont séparées entre elles par des virgules (et non pas par des points-virgules). Ces valeurs peuvent être directement données (un nombre, une chaîne de caractères), être les valeurs de constantes, de variables, de paramètres, ou de calculs dont le type de résultat convient.
Dans le cas d'une procédure, l'appel (l'exécution) d'une procédure constitue une instruction complète. Par contre, dans le cas d'une fonction, l'appel peut être inséré dans une instruction : le bloc d'appel avec la syntaxe décrite ci-dessus se comportera comme une constante dont le type est celui du résultat de la fonction. Les exemples ci-dessous illustrerons cela.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Exemple :

```
function VolumeCylindre1(Hauteur: Single): Single;
begin
  Result := AireDisque(1) * Hauteur;
end;
```

Cette nouvelle fonction calcule le volume d'un cylindre de rayon 1. Le seul paramètre est la hauteur du cylindre. Plutôt que de calculer l'aire du disque de base dans la fonction, on utilise celle qui a été écrite précédemment, et qui a besoin d'un paramètre de type single. On lui donne cet unique paramètre, entre parenthèses : c'est la valeur 1. Vous voyez en passant que 'result' est encore calculé à partie d'une opération, qui fait elle-même appel à une autre fonction : 'AireDisque'.
Comme vous pouvez le constater, cette fonction a un intérêt assez limité, puisque le rayon du cylindre est fixé. On va donc en écrire une plus générale à laquelle on va ajouter un paramètre nommé 'RayonBase' de type single. Il restera à faire intervenir ce paramètre dans le calcul :

```
function VolumeCylindre(RayonBase, Hauteur: Single): Single;
begin
  Result := AireDisque(RayonBase) * Hauteur;
end;
```

L'exemple ci-dessus est déjà plus pertinent : la valeur du paramètre 'RayonBase' de la fonction 'VolumeCylindre' est transmise en tant que valeur pour le paramètre 'Rayon' de la fonction 'AireDisque'. La fonction AireDisque est exécutée et renvoie l'aire du disque, que l'on multiplie par la hauteur pour obtenir le volume.

Exercice 1 : (voir la **solution**)

1. Ecrivez une fonction VolumeCyl qui calcule directement le volume d'un cylindre, sans faire appel à la fonction 'AireDisque'.
2. Ecrivez une fonction PerimetreCercle qui calcule le périmètre d'un cercle (périmètre = 2 x PI x Rayon).
3. Ecrivez enfin une fonction SurfaceCyl qui calcule la surface totale d'un cylindre régulier droit. Pour cela, vous devrez additionner l'aire des deux disques de base avec l'aire de la partie circulaire, que vous pourrez calculer en utilisant la fonction PerimetreCercle écrite auparavant.

Pour fonctionner, la deuxième fonction doit pouvoir avoir accès à la première. Il y a plusieurs moyens que vous connaissez déjà :

- Si elles sont dans deux unités différentes, alors l'unité contenant 'VolumeCylindre' doit utiliser (bloc uses) l'unité contenant 'AireDisque', que ce soit dans l'interface ou dans l'implémentation. De plus, la fonction AireDisque devra être déclarée dans l'interface de l'unité dans laquelle elle est écrite.
- Si les deux fonctions sont dans la même unité :
  - Si la fonction 'AireDisque' est écrite au dessus de la fonction 'VolumeCylindre', alors il n'y a rien à faire de plus.
  - Si la fonction 'AireDisque' est écrite en dessous de la fonction 'VolumeCylindre', alors 'AireDisque doit être déclarée dans l'interface de l'unité. Il existe une possibilité pour éviter cela si vous ne voulez pas déclarer 'AireDisque' dans l'interface, c'est de le faire dans l'implémentation, au dessus de 'VolumeCylindre'. Mais, me direz-vous, j'ai dit au dessus que c'était interdit, et je maintiens. La déclaration, pour être acceptée dans l'implémentation, doit ajouter à la fin de la ligne de déclaration, après le point-virgule final, le texte suivant :

```
            forward;
```

    ('forward' signifie littéralement 'plus loin')


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La déclaration de la fonction devient alors :

```
function AireDisque(Rayon: Single): Single; forward;
```

C'est ce qu'on appellera une déclaration forward. L'intérêt principal des déclarations forward est de ne pas avoir à déclarer une fonction (ou une procédure) dans l'interface si on ne veut vraiment pas la laisser accessible de l'extérieur de l'unité.
Pour ceux que cela intéresse, la raison pour laquelle on ajoute **forward** est assez simple : si on ne l'ajoute pas, tout ce qui suivra dans l'implémentation sera considéré, à tort évidemment, comme étant des déclarations locales de la fonction, ce qui ne manquera de provoquer d'innombrables erreurs lors de la compilation.
Voici maintenant des exemples complets faisant apparaître chacune des différentes possibilités :
Exemple 1 : Fonctions dans le bon ordre

```
unit test;

interface

implementation

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

function VolumeCylindre(RayonBase, Hauteur: Single): Single;
begin
  Result := AireDisque(RayonBase) * Hauteur;
end;

end.
```

Exemple 2 : Fonctions dans l'ordre inverse, avec une déclaration dans l'interface

```
unit test;

interface

function AireDisque(Rayon: Single): Single;

implementation

function VolumeCylindre(RayonBase, Hauteur: Single): Single;
begin
  Result := AireDisque(RayonBase) * Hauteur;
end;

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

end.
```

Exemple 3 : Fonctions dans l'ordre inverse, avec une déclaration forward dans l'implémentation

```
unit test;

interface

implementation

function AireDisque(Rayon: Single): Single; forward;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
function VolumeCylindre(RayonBase, Hauteur: Single): Single;
begin
  Result := AireDisque(RayonBase) * Hauteur;
end;

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

end.
```

Exemple 4 : Fonctions dans deux unités différentes

```
unit test;

interface

uses
  calcul;

implementation

{ Le bloc uses :

uses
  calcul;

aurait aussi pu être placé à cet endroit, en enlevant ce descriptif et les marques de commentaire }

function VolumeCylindre(RayonBase, Hauteur: Single): Single;
begin
  Result := AireDisque(RayonBase) * Hauteur;
end;

end.
```

```
unit calcul;

interface

function AireDisque(Rayon: Single): Single;

implementation

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

end.
```

## VI-D - Manipulations

Il est essentiel de manipuler un peu ces notions pour voir comment elles marchent. Nous allons avoir besoin de quelques petits compléments avant les manipulations. Allez donc sous Delphi, et ouvrez le projet PremierEssai dans son état actuel. Si vous ne voyez pas la seule fiche du projet, affichez l'unité qui lui correspond, à savoir 'principale' puis appuyez sur F12 pour basculer de l'unité à la fiche.
Une fois la fiche visible, effectuez un double-clic sur le bouton 'Action !'. Cela a pour effet de vous faire retourner dans l'unité 'principale' où du texte a été ajouté. Le curseur se trouve maintenant entre le begin et le end d'une procédure :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

*Note : plus tard, si cette procédure venait à être effacée par Delphi (rassurez-vous, Delphi ne l'efface que lorsque vous n'avez rien ajouté au "squelette" ci-dessous), il suffira de renouveler cette manipulation pour la faire revenir.*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
{ <-- Curseur }
end;
```

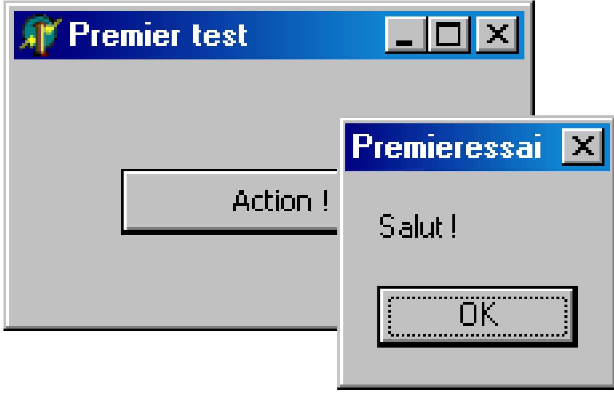
Vous allez devoir (je suis désolé, mais c'est impossible de faire autrement) ignorer les choses suivantes encore pendant un certain temps (mais je vous promets que cela ne durera pas trop longtemps) :

- comment cette procédure est arrivée là.
- le nom étrange de cette procédure : 'TForm1.Button1Click', qui n'est pas un identificateur valide (pour vous, le point est encore interdit, nous verrons plus tard dans quelles circonstances l'utiliser).
- le type de l'unique paramètre de cette procédure : 'TObject' (pour les curieux de tout, c'est une classe d'objets, mais ne m'en demandez pas plus pour l'instant, c'est pour plus tard !).

La seule chose qu'il est important pour vous de savoir maintenant est que cette procédure sera exécutée à chaque fois qu'un clic sera effectué sur le bouton pendant l'exécution de l'application.
Il va commencer à être urgent d'avoir des réponses visibles de la part de l'ordinateur, c'est pour cela que nous utiliserons la procédure 'ShowMessage' (lisez « Show Message », « Montrer Message » en français) fournie par Delphi et contenue dans une unité appelée 'Dialogs' que vous n'aurez pas besoin d'ajouter dans l'un des blocs 'uses de l'unité 'principale' car elle doit déjà y être : vérifiez et ajoutez-là si elle n'y est pas.
Cette procédure accepte un unique paramètre, de type chaîne de caractères ('string'). La chaîne en question, lors de l'exécution de la procédure, sera affichée dans une petite boite de dialogue avec un bouton OK. Ce sera suffisant dans l'immédiat pour nous permettre de contrôler ce que fait notre application pendant son exécution.
Insérez le code présenté ci-dessous (entrez juste l'instruction puisque le reste est déjà écrit) :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Salut !');
end;
```

Et lancez l'application. Lorsque vous cliquez sur le bouton, une petite fenêtre avec le texte 'Salut !' et un bouton OK apparaît :




Ceci ne constitue qu'un premier essai qui vous permet de vérifier par vous-même votre agilité sous Delphi. Lorsque vous en avez assez vu, quittez l'application (n'oubliez jamais de quitter l'application avant de la modifier sous Delphi, les résultats seraient assez imprévisibles). Nous allons maintenant passer à quelque chose de plus consistant.
Ajoutez maintenant une nouvelle unité au projet (une unité sans fiche) : pour cela, allez dans le menu 'Fichier', puis 'Nouveau...' et choisissez 'Unité' dans la liste qui vous est proposée. Une nouvelle unité est alors générée par Delphi, avec un texte minimal (le bloc unit, les deux parties interface et implémentation et le end final). Commencez par


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

enregistrer cette unité (Menu 'Fichier', puis 'Enregistrer'), en donnant 'calculs' comme nom de fichier ('.pas' sera ajouté automatiquement, et la première ligne de l'unité changera pour afficher le nouveau nom : 'calculs'.
Dans cette nouvelle unité, écrivez la fonction 'AireDisque' (texte ci-dessus) et déclarez-là pour qu'elle soit accessible de l'extérieur. Essayez de le faire sans regarder le listing ci-dessous. Une fois terminé, comparez votre travail avec ce qui suit :

```
unit calculs;

interface

function AireDisque(Rayon: Single): Single;

implementation

function AireDisque(Rayon: Single): Single;
begin
  Result := PI * Rayon * Rayon;
end;

end.
```

Il va maintenant nous falloir appeler cette fonction. Pour cela, il suffira d'écrire :

```
AireDisque(3.2);
```

Mais il serait plus avantageux de pouvoir stocker ce résultat, même si nous n'allons rien en faire dans l'immédiat. Pour stocker ce résultat, il nous faut de toute manière une variable, et son type doit être choisi pour que la variable accepte les valeurs renvoyées par AireDisque. 'single' paraît donc le plus adapté. Sans regarder le listing ci-dessous, déclarez une variable locale nommée 'Aire' de type 'single' dans la procédure nommée 'TForm1.Button1Click'. Vérifiez ensuite ci-dessous :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
begin
  ShowMessage('Salut !');
end;
```

Nous allons pouvoir utiliser cette variable pour stocker le résultat de la fonction AireDisque. Ceci se fera au moyen d'une affectation (Le résultat de la fonction sera affecté à la variable 'Aire' après son exécution). L'instruction sera donc :

```
Aire := AireDisque(3.2);
```

L'affectation étant la moins prioritaire, AireDisque sera exécutée, avec 3.2 comme valeur du paramètre 'Rayon', et le résultat sera stocké dans la variable 'Aire'.
Tapez cette instruction à la fin de la procédure 'TForm1.Button1Click'. Le code source de la procédure doit maintenant être :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
begin
  ShowMessage('Salut !');
  Aire := AireDisque(3.2);
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il reste un détail à règler : 'AireDisque' est une fonction de l'unité 'Calculs', il vous faudra donc utiliser cette unité dans l'unité 'Principale' (bloc **uses** de l'interface).
Vous pouvez maintenant essayer de lancer l'application, mais comme nous n'avons rien prévu pour afficher le résultat à l'écran comme nous l'avions fait pour la chaine 'Salut !', rien d'autre ne s'affichera quand vous cliquerez sur le bouton, et pourtant, la fonction AireDisque sera appelée et son résultat stocké dans 'Aire'. Il est assez tentant de vérifier cela, non ?
Pour cela, il va nous falloir afficher à l'écran un nombre à virgule. C'est impossible directement : il va falloir *convertir* ce nombre en chaîne de caractère (transformer par exemple 1.23 en '1.23' car seules les chaînes de caractères peuvent être affichées via la procédure 'ShowMessage'. La conversion se fera au moyen de l'utilisation de la fonction nommée 'FloatToStr' (lisez « Float To Str », « de Flottant à Chaîne » en français). Cette fonction accepte un unique paramètre de type 'extended' et son résultat est de type 'string' (chaîne de caractères). La valeur du paramètre, dans notre cas, sera donnée par la variable 'Aire'.
Il va nous falloir une variable de type 'string' pour stocker le résultat de cette fonction : déclarez donc (toujours dans la même procédure une variable 'ChaineAire' de type 'string'. Ajoutez en fin de procédure l'instruction qui affecte à 'ChaineAire' le résultat de la fonction 'FloatToStr' à laquelle on donne comme unique paramètre la variable 'Aire'. Regardez ensuite le listing ci-dessous pour vous corriger.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
  ChaineAire: String;
begin
  ShowMessage('Salut !');
  Aire := AireDisque(3.2);
  ChaineAire := FloatToStr(Aire);
end;
```

Lorsque vous cliquerez sur le bouton, un petit message s'affichera, puis la fonction AireDisque sera exécutée et son résultat sera stocké dans 'Aire'. Enfin, la valeur de cette variable est transmise comme valeur de paramètre à la fonction 'FloatToStr' qui renvoie une chaîne de caractères. Cette dernière sera stockée dans la variable 'ChaineAire'.
Il ne reste plus maintenant qu'à afficher cette chaîne. Nous allons nous servir à nouveau de 'ShowMessage' en transmettant comme valeur de l'unique paramètre la valeur de 'ChaineAire' qui est bien de type chaîne. Essayez de rajouter à la fin de la procédure l'appel de la procédure 'ShowMessage' avec pour paramètre 'ChaineAire' (c'est plus simple que ce qui précède). Corrigez-vous, désormais comme d'habitude, avec le listing suivant :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
  ChaineAire: String;
begin
  ShowMessage('Salut !');
  Aire := AireDisque(3.2);
  ChaineAire := FloatToStr(Aire);
  ShowMessage(ChaineAire);
end;
```

Je tiens un pari sur votre erreur probable : vous avez peut-être bien écrit 'ChaineAire' au lieu de ChaineAire. La première écriture, qui est fausse, désigne la chaîne contenant le texte « ChaineAire ». Le second désigne la variable ChaineAire. C'est déjà une variable de type chaîne, qui n'a donc pas besoin d'être à nouveau mise entre simples quotes.
Vous pouvez maintenant lancer l'application : deux messages vont s'afficher, l'un que vous connaissez déjà, et l'autre qui est nouveau, il doit ressembler à cela :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce qui répond bien à nos attentes.

*Note : Un problème subsiste : pour le trouver, utilisez une calculatrice (même celle de Windows) convient, en utilisant la valeur de PI : 3.141592653589. Vous avez deviné ? Eh oui, le résultat donné par notre petite application est... faux ! Pas de beaucoup, je vous l'accorde : les 4 premières décimales sont justes. Mais je profite de cette occasion pour vous apprendre une chose : le problème est insoluble en programmation comme avec une calculatrice : vous aurez toujours un résultat approché, jamais exact. On pourrait augmenter la précision du calcul en n'utilisant pas le type 'single' mais le type 'extended' plus précis, mais l'erreur reviendrait au bout de quelques décimales. Vous ne pouvez pas y faire grand chose, mis à part être vigilant et tenir compte des approximations lorsque vous écrirez des programmes faisant intervenir des nombres à virgule.*

Il faudrait maintenant faire quelque chose de plus convivial. Annoncer comme cela un nombre n'est pas très convenable. Il faudrait mieux quelque chose du genre : « L'aire d'un disque de rayon 3,2 cm vaut ??? cm² ». Pour cela, nous allons utiliser une notion vue avec les chaînes de caractères : la concaténation. On va en effet coller plusieurs chaînes ensemble, même si l'une d'entre elle est la valeur d'une variable, c'est possible.
Déclarez donc une autre variable nommée 'ChaineAire2' de type 'string' (pensez que plusieurs variables du même type peuvent se déclarer en un seul bloc). Cette nouvelle variable va devoir recevoir la concaténation de plusieurs chaînes, qui vont constituer ensemble un message complet. Voici :

```
ChaineAire2 := 'L''aire d''un disque de rayon 3,2 cm vaut ' + ChaineAire +
' cm².';
```

Les deux chaînes que l'on donne directement entre simples quotes seront concaténées avec la valeur de 'ChaineAire' pour donner un message compréhensible par l'utilisateur. Restera à bien afficher ce nouveau message et non plus l'ancien dans l'instruction finale. Voici le nouveau code source :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
  ChaineAire, ChaineAire2: String;
begin
  ShowMessage('Salut !');
  Aire := AireDisque(3.2);
  ChaineAire := FloatToStr(Aire);
  ChaineAire2 := 'L''aire d''un disque de rayon 3,2 cm vaut ' + ChaineAire +
' cm².';
  ShowMessage(ChaineAire2);
end;
```

Effectuez les modifications dans Delphi et lancez l'application, vous devriez maintenant obtenir ceci :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce qui, vous me l'accorderez, est nettement plus convivial, mais qui n'est encore pas terminé. supprimez la première instruction (celle qui affiche 'Salut !') car vous avez certainement aussi marre que moi de voir s'afficher cette petite fenêtre ! (songez à la première fois, comme vous étiez content de la voir, vous vous habituez déjà à savoir l'afficher)
Nous allons maintenant passer à une autre phase de la programmation : l'optimisation du code source. En effet, notre code a beau fonctionner, il ferait hurler la plupart des habitués à Pascal. Nous allons donc procèder à quelques améliorations de fond.
En tout premier lieu, la variable ChaineAire2 n'a pas de raison d'être. En effet, il est autorisée d'affecter à une variable un résultat qui utilise cette variable dans sa détermination. Je m'explique : vous avez une variable, vous pouvez en une seule fois la modifier et affecter le résultat modifié dans la variable puisque l'opération est effectuée d'abord, et l'affectation ensuite. Voici ce que cela donnera avec notre exemple :

```
ChaineAire := 'L''aire d''un disque de rayon 3,2 cm vaut ' + ChaineAire +
' cm².';
```

Que cela ne vous choque pas : la valeur de ChaineAire sera d'abord concatènée avec les deux autres chaînes, et ensuite seulement, la chaîne résultante sera stockée dans ChaineAire. Ceci va nous permettre de supprimer la variable ChaineAire2 tout en n'oubliant pas d'afficher non plus ChaineAire2 mais ChaineAire dans la dernière instruction. Voici le code source avec ces modifications :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
  ChaineAire: String;
begin
  Aire := AireDisque(3.2);
  ChaineAire := FloatToStr(Aire);
  ChaineAire := 'L''aire d''un disque de rayon 3,2 cm vaut ' + ChaineAire +
' cm².';
  ShowMessage(ChaineAire);
end;
```

Voilà qui est déjà mieux. Nous allons maintenant nous intéresser à la deuxième et à la troisième ligne. Chacune de ces deux lignes correspond à une instruction d'affectation. Le résultat de la fonction FloatToStr est d'abord stocké dans ChaineAire et cette chaîne est ensuite « habillée » pour en faire un message. Ces deux instructions peuvent être rassemblées en une seule :

```
ChaineAire := 'L''aire d''un disque de rayon 3,2 cm vaut ' + FloatToStr(Aire) +
' cm².';
```

En effet, utiliser le contenu de ChaineAire ou le résultat de FloatToStr revient au même puisque nous voulons qu'ils aient la même valeur (d'où la première affectation). Voici alors le nouveau code source de la procédure :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
  ChaineAire: String;
begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  Aire := AireDisque(3.2);
  ChaineAire := 'L''aire d''un disque de rayon 3,2 cm vaut ' + FloatToStr(Aire) +
' cm².';
  ShowMessage(ChaineAire);
end;
```

Nous n'allons pas encore nous arrèter là. Une amélioration est encore possible : la variable 'ChaineAire' est utilisée dans une suite d'instructions particulière : une affectation puis une seule fois en tant que paramètre. C'est un cas typique qu'il vous faudra apprendre à reconnaitre. Dans ce cas, il est possible d'utiliser la valeur affectée à la variable *directement en tant que valeur de paramètre*. Voici donc le nouveau code source ; remarquez bien comme ce qui était auparavant affecté à ChaineAire est maintenant donné à 'ShowMessage' comme valeur de paramètre. La variable ChaineAire a été supprimée puisqu'elle n'est plus utilisée.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Aire: Single;
begin
  Aire := AireDisque(3.2);
  ShowMessage('L''aire d''un disque de rayon 3,2 cm vaut ' + FloatToStr(Aire) +
' cm².');
end;
```

Notre procédure a déjà pas mal perdu de poids. Notez que la variable 'Aire' est dans le même cas que l'ex variable 'ChaineAire', et que l'on pourraît donc s'en passer. Par souci de lisibilité du code source de la procédure, et aussi parce que cette variable sera utilisée dans la suite du guide, nous la laisserons en place. Remarquez cependant que de 4 instructions et 3 variables, nous sommes descendus à 2 instructions et une seule variable : le code a été en grande partie optimisé.
La manipulation est maintenant terminée, vous pouvez télécharger le projet dans son état actuel **ici** ou, si vous avez téléchargé le guide pour le consulter hors connexion, vous pourvez ouvrir le sous-répertoire "Projets\Premier Essai 2\" du guide, et ouvrir le projet "PremierEssai.dpr" sous Delphi.

## VI-E - Conclusion

Cette partie vous a permis de sauter à pieds joints dans la programmation en Pascal. En effet, les procédures et les fonctions, sinon indispensables, sont souvent incontournables, au moins pour ce qui est de l'utilisation de ce que Delphi nous offre ('FloatToStr' et 'ShowMessage' par exemple).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# VII - Types de données avancés de Pascal Objet

Ce chapitre, plutôt théorique, est consacré à la suite de l'étude du langage Pascal Objet. Delphi ne sera utilisé ici que pour vous permettre de manipuler les notions nouvelles.
Les types de données que vous connaissez actuellement, à savoir nombres entiers, à virgule, caractères et chaînes de caractères, énumérés et booléens, suffisent pour de petits programmes, mais deviendront insuffisants dans de nombreuses situations. D'autres types de données plus élaborés devront alors être utilisés. Le début de ce chapitre aborde des types standards de Delphi non encore connus de vous, tandis que la fin du chapitre vous présentera l'une des deux manières pour construire son type personnalisé à partir d'autres types (l'autre manière, nettement plus complexe et puissante, sera vue, mais nettement plus tard, dans le guide).

## VII-A - Création de nouveaux types

A partir de ce point, il va vous falloir connaître un nouveau bloc Pascal : le bloc de déclaration de type. Ce bloc permet de définir des nouveaux types, comme les blocs de constantes et de variables définissaient des constantes et des variables.
Un bloc de déclaration de type peut se situer aux mêmes endroits que les blocs **const** et **var**. Le bloc commence par contre par le mot Pascal réservé **type** et possède la syntaxe suivante :

```
type
  Déclaration_de_type;
  { ... }
  Déclaration_de_type;

Chaque déclaration de type est de la forme :

Identificateur = Specification_du_type
```

où *Identificateur* est un identificateur non encore utilisé qui désignera le nouveau type. *Spécification du type* peut prendre diverses formes et au fur et à mesure des paragraphes ci-dessous, nous en verrons un certain nombre. Une habitude veut que tous les identificateurs de types personnalisés commencent par la lettre T majuscule suivi d'un nom descriptif (toutefois, les types pointeurs que nous étudierons au chapitre 9 font exception à cette convention en utilisant la lettre P).
Il est possible de donner pour *Spécification du type* un type standard, tel 'integer' ou 'byte', ou n'importe quel type connu.
Les types ainsi définis peuvent être utilisés ensuite dans les blocs **const** et **var** de la manière suivante :

```
type
  TChaineCourte = String[10];
const
  CBonj: TChaineCourte = 'Bonjour';
var
  InitialesNom: TChaineCourte;
```

Ces types peuvent également être utilisés partout où un type est requis, par exemple dans la liste des paramètres d'une procédure ou d'une fonction ou comme type de résultat d'une fonction. Voici un exemple :

```
function Maj(Ch: TChaineCourte): TChaineCourte;
begin
  result := UpperCase(Ch);
end;
```

(UpperCase est une fonction fournie par Delphi qui renvoie la chaine donnée en paramètre convertie en majuscules)

Exercice 1 : (voir la **solution**)
Ecrivez le bloc **type** complet déclarant les deux nouveaux types suivants :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- TChaine200 : type chaîne de 200 caractères.
- TEntier : type équivalent au type integer.

## VII-B - Type ordinaux

Les types ordinaux sont non pas un nouveau type mais une particularité pour un type. Certains types permettent des valeurs classées suivant un ordre. Ces types sont alors dits *ordinaux*. C'est le cas, parmi les types que vous connaissez, de tous les types entiers ('integer', 'byte', ...), des booléens, des caractères (mais pas des chaînes de caractères) et des énumérés. Il sera dorénavant mentionné lors de la présentation de nouveaux types s'ils sont ordinaux ou pas.
Les types ordinaux ont tous une sorte de correspondance avec les nombres entiers. Chaque donnée (constante ou variable) d'un type ordinal possède ce qu'on appelle une *valeur ordinale*, qui est donnée par la fonction 'Ord'. Cette fonction fait partie d'un ensemble de fonctions assez particulières qui acceptent des paramètres divers. En l'occurence, 'Ord' accepte n'importe quelle constante ou variable de type ordinal.
La fonction 'Ord' sera utilisée dans les paragraphes suivants, de même que nous reviendrons sur les types ordinaux.

## VII-C - Type intervalle

Le type intervalle permet de définir un nouveau type ordinal personnalisé autorisant un intervalle de valeurs ordinales, en donnant les valeurs extrèmales (son minimum et son maximum). Ceci permettra de nombreuses choses notamment pour un autre type que nous allons bientôt voir.
Ce type s'écrit comme suit :
*ValeurMinimale..ValeurMaximale*
Où *ValeurMinimale* et *ValeurMaximale* sont du même type ordinal. Il est à noter que la valeur ordinale de *ValeurMinimale* doit être inférieure ou égale à celle de *ValeurMaximale*, ou sinon le compilateur signalera une erreur.
La déclaration d'un type, ou d'une variable de ce type, se font de la manière suivante :

```
type
  nom_de_type = ValeurMinimale..ValeurMaximale;
var
  nom_de_variable: ValeurMinimale..ValeurMaximale;
```

Exemples :

```
type
  TLettreMaj = 'A'..'Z';
var
  NbPositif: 0..1000000;
  LettreMin: 'a'..'z';
  LecteurDisque: TLettreMaj;
  Chiffre: 0..9;
```

Déclarer une constante de type intervalle avec une valeur hors de cet intervalle est interdit. De même, pour des variables de type intervalle, assigner directement une valeur hors de l'intervalle est interdit par le compilateur :

```
NbPositif := -1;
```

provoquera une erreur, mais par contre :

```
Chiffre := 9;
Chiffre := Chiffre + 1;
```

sera accepté et pourtant "Chiffre" n'est plus entre les deux valeurs extremales. Ce sera à vous de faire attention.



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## VII-D - Compléments sur les types énumérés

Il est possible de déclarer de nouveaux types énumérés, personnalisés. La syntaxe en est la suivante :
*Nom_du_type = (IdValeur0, IdValeur1, ..., IdValeurn);*
où *Nom_du_type, IdValeur0, IdValeur1, ..., IdValeurn* sont des identificateurs non utilisés. *Nom_du_type* désignera alors le type et *IdValeur0, IdValeur1*, ..., *IdValeurn* seront les n+1 valeurs possibles d'une constante ou variable de ce type. Chacun de ces identificateur a une valeur ordinale (qu'on obtient, je rappelle, à l'aide de la fonction Ord) déterminée par la déclaration du type : le premier identificateur prend la valeur ordinale 0, le suivant 1, et ainsi de suite. *IdValeur0* a donc pour valeur 0, *IdValeur1* a pour valeur 1, *IdValeurn* a pour valeur n. Par convention, on essayera de nommer *IdValeurx* par deux ou trois lettres minuscules qui sont une abréviation du nom de type, puis par une expression significative.

***Note aux programmeurs en C, C++ ou autres :***
*Delphi n'offre pas les mêmes possibilités que le C ou le C++ en ce qui concerne les énumérations : il n'est en effet pas possible de donner des valeurs personnalisées aux identificateurs constituant la déclaration du type. Il sera cependant possible d'outrepasser cette limitation en déclarant une constante du type énuméré et en utilisant la syntaxe (qui effectue un transtypage) :*
*Nom_De_Type_Enumere(valeur_ordinale)*
*On n'a que rarement besoin d'un tel assemblage, contactez-moi donc si vous avez besoin de précisions.*

Sous Delphi, retournez au code source de la procédure TForm1.Button1Click. Remplacez son contenu par l'exemple ci-dessous :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TTypeSupport = (tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
var
  Supp1: TTypeSupport;
  I: Byte;
begin
  Supp1 := tsCDRom;
  I := Ord(Supp1);
  ShowMessage(IntToStr(I));
end;
```

Cet exemple déclare un nouveau type énuméré local à la procédure (mais il pourrait aussi bien être déclaré ailleurs) nommé 'TTypeSupport'. Une variable 'Supp1' est déclarée de ce type. Les valeurs possibles pour cette variable sont donc tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom et tsZIP. Ces identificateurs ont une valeur ordinale. Pour vous la faire voir, une valeur a été donnée à la variable (vous pouvez changer cette valeur en piochant dans les valeurs possibles mais si vous tentez de mettre autre chose qu'une de ces valeurs, le compilateur signalera une erreur). La valeur de la variable est alors transmise à la fonction Ord qui renvoie un nombre entier positif. On a choisi de stocker ce résultat dans une variable I de type 'byte' mais on aurait pu choisir 'integer' (il y a moins de 256 valeurs possibles pour le type 'TTypeSupport', donc le type 'byte' convient bien avec sa limite de 255). Après l'exécution de l'instruction, on affiche la valeur de I à l'aide de ShowMessage et de IntToStr (IntToStr fonctionne comme FloatToStr, mais uniquement avec les nombres entiers, il sera préférable de l'utiliser avec ces derniers, plutôt que d'utiliser partout FloatToStr). En lançant l'application, vous verrez donc s'afficher la valeur ordinale de 'tsCDRom', soit 2.
C'est l'occasion idéale pour vous parler de 2 fonctions et de 2 procédures assez particulières : high, low, inc, dec. Elles acceptent chacune un unique paramètre qui doit être ordinal.

- 'High' (c'est une fonction), tout comme 'Low' a une particularité : son paramètre peut être soit une variable de type ordinal, ou directement le nom d'un type ordinal. 'High' renvoie toujours une valeur (et non un type) du même type que le paramètre transmis (le type de la variable si le paramètre est une variable ou du type donné en paramètre si le paramètre est un type), qui est la valeur dont la valeur ordinale est la plus importante pour ce type. Par exemple, 'High(TTypeSupport)' vaudra 'tsZIP'.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- 'Low' (c'est une fonction) fonctionne de la même manière, mais renvoie la valeur dont la valeur ordinale est la plus faible. Ainsi, par exemple, 'Low(TTypeSupport)' vaudra 'tsDisq35'.
- 'Inc' (c'est une procédure) accepte n'importe quelle variable de type ordinal, et la modifie en augmentant sa valeur ordinale d'une unité. Ainsi, pour un nombre entier, 'Inc' a le même effet que si l'on additionnait 1. Par contre, sur les autres types, les effets sont plus intéressants car on ne peut pas leur ajouter 1 directement, il faut passer par 'Inc'.
- 'Dec' a l'effet contraire : il enlève 1 à la valeur ordinale de la variable qu'on lui transmet en paramètre.

Exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TTypeSupport = (tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
var
  Supp1: TTypeSupport;
begin
  Supp1 := Low(TTypeSupport);
  ShowMessage(IntToStr(Ord(Supp1)));
  Inc(Supp1);
  ShowMessage(IntToStr(Ord(Supp1)));
  Dec(Supp1);
  ShowMessage(IntToStr(Ord(Supp1)));
end;
```

Dans l'exemple ci-dessus, la valeur de la variable 'Supp1' est fixée à la valeur de valeur ordinale la plus faible du type 'TTypeSupport', qui est donc 'tsDisq35'. Un appel à 'ShowMessage' permet ensuite d'afficher la valeur ordinale de Supp1. Vous remarquez que nous avons grimpé d'un échelon en n'utilisant plus de variable I, mais directement la valeur qui lui était affectée.
La variable 'Supp1' est ensuite transmise à 'Inc', qui augmente sa valeur ordinale de 1. Supp1 vaut donc 'tsDisqueDur'. Pour nous en assurer, on affiche à nouveau la valeur ordinale de 'Supp1'. Le même procédé est enfin appliqué pour tester la procédure 'Dec'.
Si vous voulez voir la chose en application, entrez le code source ci-dessus dans Delphi (on se sert encore et toujours de la procédure 'TForm1.Button1Click', et lancez l'application.

Exercice 2 : (voir la **solution**)
Ecrivez un type énuméré pour les 7 jours de la semaine. Déclarez une variable de ce type, initialisez-la à une valeur de votre choix et affichez sa valeur ordinale (utilisez la procédure habituelle pour cela).
Si vous le pouvez, essayez de ne pas utiliser de variable intermédiaire comme dans l'exemple ci-dessus.

## VII-E - Type ensemble

Le type ensemble permet de créer des ensembles à cardinal fini (le cardinal d'un ensemble est le nombre d'éléments dans cet ensemble). Les ensembles en Pascal ne peuvent contenir que des constantes de type ordinal, dont la valeur ordinale est comprise entre 0 et 255. Toutes les constantes entières entre 0 et 255 conviennent donc, ainsi que les caractères et la plupart des types énumérés. On ne peut cependant pas mélanger les types à l'intérieur d'un ensemble, le type des éléments étant décidé dans la déclaration du type.
Le type ensemble s'écrit :
**set of** *type_ordinal*;
où *type_ordinal désigne au choix un nom de type ordinal ou spécifie directement un type ordinal. Voici des exemples :*

```
type
  TResultats = set of Byte;
  TNotes = set of 0..20;
  TSupports = set of TTypeSupport;
```

Dans l'exemple ci-dessus, 'TResultats' pourra contenir des entiers de type 'byte', 'TNotes' pourra contenir des entiers entre 0 et 20, et 'TSupports' pourra contenir des éléments de type 'TTypeSupport'.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il est alors possible de déclarer des variables de ces types. Pour utiliser ces variables, vous devez savoir comment on écrit un ensemble en Pascal : il se note entre crochets et ses éventuels sous-éléments sont notés entre virgules. J'ai bien dit éventuel car l'ensemble vide est autorisé. Par sous-élément, on entend soit un élément simple, soit un intervalle (rappelez-vous des types intervalles). les exemples ci-dessous sont des ensembles.

```
[] { ensemble vide }

[1, 2, 3] { ensemble d'entiers contenant 1, 2 et 3 }

[1, 1, 2, 3, 2] { ensemble d'entiers contenant 1, 2 et 3 }

[1..3] { ensemble d'entiers contenant 1, 2 et 3 }

['a'..'z', 'A'..'Z', '_', '0'..'9'] { ensemble de caractères contenant les lettres majuscules, minuscules, le bla

[tsDisq35..tsCDRom,
 tsZIP] { ensemble d'élements de type TTypeSupport contenant tsDisq35, tsDisqueDur, tsCDRom et tsZIP }
```

Il est en outre possible de manipuler ces ensembles comme les ensembles manipulés en mathématiques : union, intersection sont possible, de même que le retrait d'un élément (si jamais vous ne maîtrisez pas ces notions, adressez-vous au professeur de mathématiques le plus proche, ou à moi).

```
[1..3, 5] + [4]     { union de deux ensembles : le résultat est [1..5] }
[1..5] - [3, 5]     { retrait d'un sous-ensemble : le résultat est [1, 2, 4] }
[1..4, 6] * [3..8] { intersection de deux ensembles : le résultat est [3, 4, 6] }
```

Il est également possible de comparer deux ensembles. Pour cela, on utilise des opérateurs de comparaison, comme ceux que vous connaissez déjà. Le résultat de l'opérateur est alors un booléen (vous apprendrez assez rapidement quoi faire de ce genre de booléens, qui pour le moment, il est vrai, sont assez inutilisables, mis à part dans des affectations). ci-dessous, A et B désignent deux ensembles de même type, x est un élément acceptable dans ces ensembles.

```
A <= B { est inclus : renvoie vrai si A est inclus dans B }
A >= B { inclut : renvoie vrai si B est inclus dans A }
A = B  { égalité : renvoie vrai si les deux ensembles contiennent les mêmes éléments }
A <> B { différent : renvoie vrai si les deux ensembles sont différents }
x in A { élément de : renvoie vrai si l'élément x est dans A }
```

Il est également possible de déclarer des constantes de type ensemble. Voici un exemple :

```
const
  SuppAmovibles = [tsDisq35, tsCDRom, tsDVDRom, tsZIP];
```

Pour terminer ce point, voici un certainement très utile exemple général : Pour terminer ce point, voici un certainement très utile exemple général :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TTypeSupport = (tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
  TSupports = set of TTypeSupport;
var
  SuppDisponibles: TSupports;
const
  SuppAmovibles = [tsDisq35, tsCDRom, tsDVDRom, tsZIP];
begin
  { initialisation }
  SuppDisponibles := SuppAmovibles;
  { on enlève tsDVDRom et on ajoute tsDisqueDur }
  SuppDisponibles := SuppDisponibles - [tsDVDRom] + [tsDisqueDur];
```



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;
```

Cet exemple déclare les types 'TTypeSupport' et 'TSupports', une variable de type ensemble et une constante de type ensemble. La première instruction permet de donner à la variable une valeur de départ, on appelle cela une *initialisation*. Nous aurons l'occasion d'en reparler. La deuxième instruction permet de retirer un élément et d'en ajouter un autre. Notez que cela a été fait en une seule instruction, car les opérateurs sont exécutés de gauche à droite, sauf si des parenthèses sont présentes.

## VII-F - Tableaux

Les tableaux sont une possibilité puissante du langage Pascal. Imaginons par exemple que vous vouliez stocker 100 nombres entiers de type 'word'. Vous pourriez déclarer (très mauvaise méthode) 100 variables de type 'word'. Une solution préfèrable sera d'utiliser un tableau. Le principe d'un tableau est de regrouper un certain nombre d'élements du même type, en ne déclarant qu'une variable ou qu'une constante. Les éléments individuels sont ensuite accessibles via un ou plusieurs indices de position dans le tableau.
Les tableaux en Pascal peuvent avoir une ou plusieurs dimensions. Nous consacrerons un paragraphe à l'étude de chaque cas. Un troisième paragraphe présentera des notions avancées permettant de dépasser le cadre des tableaux standards vus dans les deux premiers paragraphes. Un dernier paragraphe présentera les tableaux de taille dynamique, utilisables à partir de Delphi 4.

### VII-F-1 - Tableaux à une seule dimension

Un tableau à une seule dimension se déclare de la façon suivante :
**array***[intervalle]* **of** *type*;
où **array** définit un tableau. les crochets entourent les données de dimension du tableau. Pour un tableau à une seule dimension, un seul paramètre est donné, de type intervalle (il est possible d'utiliser d'autres choses que les intervalles, ceci sera vu dans le **§ VII-F-3**). Vient ensuite le mot réservé **of** qui est suivi du type des éléments du tableau. Cet élément est un type. Tous les types vus jusqu'à présent sont acceptés, y compris d'autres tableaux. Il faudra cependant, pour certains types comme les ensembles, déclarer d'abord un type ensemble, puis utiliser ce nom de type pour déclarer le tableau. En principe, tous les types sont autorisés, du moment que la taille du tableau reste en dessous d'une limite dépendant de la version de Delphi.
Il est possible de déclarer des types, des variables et des constantes de type tableau (c.f. exemple ci-dessous). Pour accèder à un tableau, il suffit de donner le nom de la variable tableau de la constante tableau. Pour accèder à un élément, on donne le nom du tableau suivi de l'indice entre crochets. Voyez l'exemple ci-dessous pour mieux comprendre :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  { notez que tsAucun a été ajouté ci-dessous }
  TTypeSupport = (tsAucun, tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
  TSupports = set of TTypeSupport;
var
  Nombres: array[1..100] of word;
  TypeLecteur: array[1..26] of TTypeSupport;
const
  Reponses: array[0..2] of string = ('non', 'oui', 'peut-être');
begin
  Nombres[1] := 13;
  Nombres[2] := Nombres[1] + 34;
  TypeLecteur[1] := tsDisq35;
  TypeLecteur[3] := tsDisqueDur;
  TypeLecteur[4] := TypeLecteur[1];
end;
```

Vous remarquerez dans l'exemple ci-dessus que 'Nombres' est un tableau d'entiers positifs. On peut donc affecter à chaque élément (on dira également chaque *case*, ou *cellule*) un entier positif. La deuxième variable est un tableau d'éléments de type TTypeSupport. Un tel tableau pourrait être utilisé pour stocker les types de lecteur de A à Z sur


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

un ordinateur. Chaque case du tableau contient un élément de type TTypeSupport. On affecte des valeurs à 3 de ces cases.
La constante devrait retenir votre attention : pour donner la valeur d'une constante de type tableau, on donne une liste de tous les éléments, dans l'ordre, séparés par des virgules, le tout entre parenthèses. Ici, les éléments sont des chaînes de caractères et le tableau en contient 3 numérotées de 0 à 2.

Exercice 3 : (voir la **solution**)
Dans la procédure habituelle, déclarez un type tableau de dix chaînes de caractères, indicées de 1 à 10. Déclarez une variable de ce type et initialisez une de ses cases à la chaîne : « Bonjour ». Affichez ensuite le contenu de cette case à l'aide de ShowMessage (Pour cette fois, vous avez le droit d'utiliser une variable temporaire, essayez cependant de vous en passer).

## VII-F-2 - Tableaux à plusieurs dimensions

Vous aurez parfois besoin de tableaux à plusieurs dimensions (les tableaux à 2 dimensions sont similaires aux tableaux à double entrée que vous connaissez certainement déjà par exemple dans Microsoft Word, mais Pascal permet d'avoir plus de 2 dimensions). Je ne vous donnerai pas de preuve de leur utilité, vous en connaissez certainement déjà. Un tableau multidimensionnel se déclare ainsi :
**array**[*intervalle1, intervalle2*, ...] **of** *type*;
où *intervalle1* est la première dimension du tableau, *intervalle2* la deuxième, ... et *type est encore le type des éléments stockés dans le tableau.*
Les types, variables et constantes se déclarent en utilisant cette syntaxe. Pour donner des valeurs aux éléments d'une variable tableau, on utilise la syntaxe suivante :
*nom_du_tableau[indice1, indice2, ...] := valeur;*
où *nom_du_tableau* est le nom d'une variable de type tableau, *indice1* est l'indice dans la première dimension, *indice2* dans la deuxième, ... et *valeur* la valeur (dont le type est donné dans la déclaration du tableau) à stocker dans cette case.
Pour fixer la valeur d'une constante de type tableau multidimensionnel, c'est un peu plus compliqué. Considérons un tableau de dimension 3 : on devra faire comme si c'était en fait un tableau à 1 dimension dont chaque case contient un tableau à 1 dimension, qui lui-même est de dimension 1. Voyez l'exemple ci-dessous qui récapitule tout ce que nous venons de dire (Attention, la première instruction, à savoir l'affectation d'un tableau à une variable tableau, est possible à partir de Delphi 4 seulement, si vous n'avez que Delphi 2 ou 3, il vous faudra attendre que nous ayons vu les boucles 'for' pour passer outre ce genre de difficulté).

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TTab3Dim = array[1..3, 1..2, 0..1] of integer;
const
  TabExemple: TTab3Dim = { Examinez bien les parenthèses : 3 blocs qui chacun en }
    (((1, 2), (3, 4)),    { contiennent 2 qui contiennent eux-mêmes chacun deux }
     ((5, 6), (7, 8)),    { valeurs de type integer }
     ((9, 10), (11, 12)));
var
  Tab1: TTab3Dim;
begin
  Tab1 := TabExemple;
  Tab1[2, 1, 0] := 20; { ancienne valeur : 5 }
end;
```

Dans l'exemple ci-dessus, un type de tableau à 3 dimensions a été créé, ce type étant ensuite utilisé pour déclarer une constante et une variable. Attardez-vous sur la constante 'TabExemple' : sa valeur est donnée entre parenthèses. Chaque élément de la première dimension est considéré comme un tableau, et est donc encore noté entre parenthèses. Enfin, chaque élément de la deuxième dimension est considéré comme un tableau de 2 éléments, dont vous connaissez déjà la déclaration. Il est assez facile de se tromper dans les déclarations de constantes de type tableau, mais rassurez-vous, le compilateur saura vous signaler toute parenthèse mal placée lors de la compilation du projet.
La première instruction permet d'*initialiser la variable : on lui donne une valeur*. En effet, contrairement à certains langages que vous connaissez peut-être, la mémoire utilisée par les variables n'est pas nettoyée avant utilisation,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

c'est-à-dire que les variables, avant que vous leur donniez une première valeur, peuvent valoir tout et n'importe quoi (en général, c'est surtout n'importe quoi !). Une initialisation permet de donner une première valeur à une variable. Il ne faudra jamais utiliser la valeur d'une variable sans qu'elle ait été initialisée auparavant.
On affecte directement la valeur d'une constante de type TTab3Dim à une variable de même type, il n'y a donc pas de problème.
La deuxième instruction modifie la valeur d'une des cases du tableau, repèrez bien les indices et reportez-vous à la déclaration de la constante pour voir à quelle position ces indices font référence. Il vous faudra tenir compte des intervalles dans lesquels sont pris les indices (les indices de dimension 3 sont pris dans l'intervalle 0..1).

## VII-F-3 - Notions avancées sur les tableaux

Nous nous sommes limités jusqu'à présent pour les indices des tableaux à des intervalles. Ce n'est que l'une des possibilités offertes : la plus simple. Il est en fait possible d'utiliser tout type ordinal en tant qu'indice. Le cas le plus intéressant à examiner est celui des types énumérés : il sera possible de déclarer un tableau tel que celui-ci :
**array**[tsDisqueDur..tsDVDRom] **of** Char;
Mais il sera encore plus intéressant d'utiliser des noms de types énumérés en guise d'indices. Ainsi, la déclaration suivante permet d'utiliser un tableau dont chaque case est une valeur de type TTypeSupport :
**type** TTabSupport = **array**[TTypeSupport] **of string**;
Ce nouveau type pourra se révèler très utile, pour vous en rendre compte, regardez le code source suivant :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TTypeSupport = (tsAucun, tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
  TTabSupport = array[TTypeSupport] of string;
const
  NomDesSupports: TTabSupport =
    ('', 'Disquette 3,5"', 'Disque dur', 'CD-Rom', 'DVD-Rom', 'Disquette ZIP');
var
  Sup1: TTypeSupport;
  msg: string;
begin
  Sup1 := tsCDRom;
  msg := NomDesSupports[Sup1];
  ShowMessage('support : ' + msg);
end;
```

Le premier type est désormais habituel. Le deuxième déclare un tableau dont les cases sont indexées par des valeurs de type TTypeSupport et dont les cases contiennent des chaînes de caractères. Vient ensuite une déclaration de constante tableau. Celle-ci s'écrit en précisant d'abord le type de la constante précédé de deux points (:), puis en donnant la valeur de la constante après un signe =. La valeur s'écrit entre parenthèses : une liste de valeurs est donnée, et affectée aux cases dans l'ordre. Les valeurs sont séparées par des virgules. Chaque chaîne correspond à un élément de type TTypeSupport.
Les deux variables serviront dans les instructions.
La première de ces instructions initialise Sup1. La deuxième initialise msg, en utilisant la constante de type tableau. Comme les indices sont de type TTypeSupport, on transmet une valeur de type TTypeSupport en indice, en l'occurence c'est la valeur de 'Sup1' qui est utilisée. Chaque case du tableau étant une chaîne de caractères, NomDesSupports[Sup1] est une chaîne, qui est affectée à une variable de type chaîne. La dernière instruction est déjà connue de vous puisqu'elle se contente d'afficher un message contenant la valeur de 'msg'.
Lorsque vous exécutez l'application, un clic sur le bouton affiche alors la boite de dialogue ci-dessous :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Essayez de remplacer la valeur de départ de 'Sup1' par une autre valeur de type 'TTypeSupport' dans la première instruction. Ceci aura un effet sur la valeur de 'msg', et modifiera donc le message affiché.

Exercice 4 : (voir la **solution**)

1. Reprenez le code source de l' **exercice 2**. Déclarez une constante de type tableau avec comme indices le type TJourSemaine, et comme contenu des cases, des chaînes de caractères. Le contenu de chaque case sera une chaîne donnant le nom du jour représenté par la valeur de type TJourSemaine indicant cette case ('Lundi' pour jsLundi par exemple).
2. Dans le code source, au lieu d'afficher la valeur ordinale de la variable de type TJourSemaine, affichez à la place le nom du jour correspondant à la valeur de cette variable en utilisant la constante déclarée à la question 1 (indication : utilisez la valeur de la variable comme indice de la constante tableau et affichez la case ainsi obtenue).

## VII-F-4 - Tableaux de taille dynamique

*Attention : Les tableaux de taille dynamique sont utilisables à partir de Delphi 4. Si vous avez une version antérieure, rien ne vous empèche de lire ce paragraphe, mais les manipulations seront impossibles. Sachez également que les tableaux de taille dynamique ne sont pas exactement ce qu'ils semblent être, car ils sont liés à la notion de pointeurs vue plus loin dans le guide. Soyez prudents dans l'utilisation de ces tableaux tant que vous n'aurez pas lu la section consacrée aux pointeurs.*

Le langage Pascal Objet est en constante évolution dans les versions successives de Delphi. Depuis la version 5, il est possible de créer des tableaux de taille dynamique, c'est-à-dire dont la taille n'est pas fixée au départ une fois pour toutes. Ces nouveaux tableaux sont toujours indexés par des entiers, et à partir de 0. Les tableaux de taille dynamique multidimensionnels, bien qu'un peu plus délicats à manipuler, sont toutefois possibles.
Un tableau dynamique à une dimension est de la forme :
**array of** *type_de_base*
où *type_de_base*, comme pour les tableaux non dynamiques, est le type des éléments stockés dans le tableau. Vous remarquez que la partie qui donnait auparavant les dimensions, les bornes et les limites du tableau (la partie entre crochets) est justement absente pour les tableaux dynamiques.
Ces derniers tableaux sont plus complexes à manipuler que les tableaux standards : déclarer une variable de type tableau dynamique ne suffit pas, comme dans la plupart des autres cas de variables, pour que la variable soit utilisable. Du fait que leur taille n'est pas fixe, il faudra la gèrer en même temps que le tableau lui-même et donc fixer le nombre d'éléments avant la première utilisation du tableau au moyen d'une procédure : 'SetLength'. 'SetLength' accepte plusieurs paramètres : le premier est une variable de type tableau dynamique, le deuxième paramètre est le nombre d'éléments que le tableau doit pouvoir contenir. Par la suite, lorsque vous désirerez redimensionner un tableau dynamique, il suffira de rappeler "SetLength" avec une taille différente. Il est également possible de connaître la taille actuelle d'un tableau dynamique en utilisant la fonction 'Length'. 'Length' accepte un unique paramètre qui doit être une variable de type tableau dynamique.
L'exemple ci-dessous illustre ce que nous venons de voir : un tableau dynamique va être utilisé. Les explications seront données; comme d'habitude, après le code source :

```
procedure TForm1.Button1Click(Sender: TObject);
var
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  TabTest: array of string;
begin
  { initialise TabTest : TabTest contient 3 cases, indexées de 0 à 2 }
  SetLength(TabTest, 3);
  { affiche la taille du tableau }
  ShowMessage(IntToStr(Length(TabTest)));
  { initialise une case }
  TabTest[2] := 'coucou';
  { affichage du contenu de la case }
  ShowMessage(TabTest[2]);
  { redimensionnement du tableau }
  SetLength(TabTest, 2); { TabTest[2] n'est maintenant plus utilisable }
end;
```

Une variable locale de type tableau dynamique est déclarée. Les éléments stockés dans le tableau sont des chaînes de caractères. Le fait de déclarer une variable tableau dynamique ne suffit pas : avant son dimensionnement, elle ne contient pas de cellules. Ce nombre de cellules est fixé à 3 dans la première instruction par l'appel à 'SetLength'. Ceci permettra d'utiliser des cellules indexées de 0 à 2 (les index partent toujours de 0).
La deuxième instruction est plus délicate (il faut suivre les parenthèses, comme en mathématiques) : le tableau est donné en paramètre à la fonction 'Length' qui renvoie une valeur entière donnant le nombre de cases du tableau. Cette valeur entière est non pas stockée dans une variable mais directement donnée en paramètre à 'IntToStr' qui en fait une chaîne de caractères. Cette dernière est enfin directement transmise en tant que paramètre à 'ShowMessage' qui se chargera de l'afficher.
La troisième instruction initialise une case du tableau : la troisième (qui est indexée par 2). La 4ème affiche la chaîne 'coucou' en se servant de la valeur de la case du tableau. Enfin, la dernière instruction modifie la taille du tableau : il ne contient plus alors que 2 chaînes. La conséquence est que la chaîne dans l'ancienne troisième case est maintenant inaccessible : si on plaçait la quatrième instruction après la cinquième, une erreur serait signalée à la compilation ou à l'exécution.
Parlons maintenant un peu des tableaux dynamiques multidimensionnels : il n'est pas possible, dans la déclaration des tableaux dynamiques, de donner directement plusieurs dimensions : il faut utiliser une astuce : les éléments stockés dans le tableau à 1 dimension seront eux-mêmes des tableaux dynamiques à 1 dimension. Voici donc la déclaration d'un tableau dynamique à 2 dimensions contenant des entiers :
**array of array of** integer;
Il faut ici faire extrèmement attention à ne pas se tromper : ce n'est pas un vrai tableau multidimensionnel que nous avons ici : chaque case d'un tableau à une dimension contient un autre tableau. On est plus proche de ce qui suit :

```
array[1..10] of array[1..20] of integer;
```

que de cela :

```
array[1..10, 1..20] of integer;
```

Je ne dis pas ça pour vous embêter mais pour vous préparer à une possilité intéressante de ce genre de tableau : Voici un exemple qui utilise un tableau dynamique à 2 dimensions. Le tableau est d'abord utilisé en tant que tableau classique, puis on prend partie du fait que chaque cellule est un tableau dynamique, et donc que rien n'oblige tous ces petits tableaux à avoir la même taille.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TabTest2: array of array of Integer;
begin
  { init. de TabTest2 : remarquez les deux entiers données en paramètre }
 SetLength(TabTest2, 10, 20);
  { initialisation d'une case, une seule paire de crochets est utilisée }
  TabTest2[6, 18] := 108;
  { changement des dimensions, la cellule ci-dessus devient inaccessible }
  SetLength(TabTest2, 2, 4);
  { changement de la taille d'une cellule }
  SetLength(TabTest2[1], 10);
  { TabTest2[0, 8] n'existe pas, mais TabTest2[1, 8] existe }
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  TabTest2[1, 8] := 29;
end;
```

Dans l'exemple ci-dessus, une variable de type tableau dynamique est déclarée. Ce tableau comporte 2 dimensions. La première instruction fixe la taille du tableau. Après le paramètre tableau, 'SetLength' accepte en fait autant de paramètres que le tableau comporte de dimensions. Le premier paramètre donne la taille dans la première dimension (que nous assimilerons ici à des lignes) : 10 lignes. La deuxième valeur donne le nombre de colonnes par ligne, à savoir 20. Le résultat est donc un tableau de 10 cellules par 20. La deuxième instruction donne un exemple d'utilisation d'un tel tableau : les deux dimensions sont données dans un seul crochet, ce qui peut paraître bizarre du fait que TabTest2 est plus un tableau de tableaux qu'un tableau à deux dimensions : c'est une commodité offerte par le langage. Il faudrait normalement écrire :

```
TabTest2[6][18] := 108;
```

écriture qui sera bien entendu acceptée par Delphi 5. La troisième instruction redimensionne le tableau en plus petit. C'est la même instruction que la première, avec des tailles différentes. La quatrième instruction vous montre la puissance de ce genre de tableaux : chaque cellule de la première dimension étant elle-même en fait un tableau dynamique à une dimension, il est possible de donner une taille différente à chacune de nos « lignes ». On obtient un tableau qui n'a plus rien de rectangulaire, mais qui pourra à l'occasion se montrer très utile. La dernière instruction se contente de donner une valeur à l'une des cases. Le texte en commentaire rappelle que cette case existe dans la deuxième ligne mais pas dans la première.
Si vous ne voyez pas vraiment l'intérêt de ces tableaux dynamiques, sachez que tout ceci est un petit avant-goût de la puissance des *pointeurs*, que nous verrons un peu plus tard. Si vous n'avez pas accès aux tableaux dynamiques à cause de votre version de Delphi, sachez que ces *pointeurs* vous permettrons de réaliser la même chose, en plus compliqué toutefois.

## VII-G - Enregistrements

### VII-G-1 - Vue d'ensemble sur les enregistrements

Les enregistrements sont des types puissants de Pascal Objet : ils sont entièrement personnalisables. Le principe est de rassembler en un seul bloc des données diverses qu'il aurait autrement fallu stocker dans des endroits séparés. Les enregistrements permettront en quelque sorte de définir des moules à partir desquels seront obtenues des variables et des constantes.
Imaginons par exemple que vous ayez besoin dans une application de manipuler des informations relatives à un unique élément. Nous prendrons par exemple un logiciel : vous manipulerez entre autres son nom, sa version, sa langue, son occupation en espace disque. Pour manipuler l'un de ces éléments, vous pourriez déclarer 4 variables destinées chacune à stocker une information sur ce logiciel, mais cela deviendrait rapidement long et fastidieux. Il serait bien plus avantageux de n'avoir à déclarer qu'une seule variable capable de stocker toutes ces informations. Les enregistrements répondent à cette demande. Ils permettent, via la création d'un nouveau type, de créer des variables « à tiroirs ». Un type enregistrement se déclare comme suit :

```
type
  nom_de_type = record
    declarations_de_membres
  end;
```

*nom_de_type* désigne le nom par lequel le nouveau type enregistrement sera accessible. le mot Pascal réservé **record** débute un bloc de déclaration de membres, qui est terminé par le mot réservé **end** suivi comme d'habitude d'un point-virgule.
Ce bloc de déclarations de membres permet de décrire ce que contiendra une variable ou constante de ce type. Ce bloc est presque identique à un bloc de déclaration de variables, mis à part le mot réservé **var** qui est alors omis, et bien entendu le fait qu'on ne déclare absolument pas des variables mais qu'on décrit plutôt la structure du type : ce qu'il contiendra et sous quelle forme.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La déclaration de type ci-dessous déclare un type enregistrement pouvant contenir les informations d'un logiciel :

```
type
  TLogiciel = record
    Nom,                { nom du logiciel }
    Version,            { version du logiciel }
    Langue: string;     { langue du logiciel }
    Taille: integer;    { taille exprimée en kilo-octets }
  end;
```

La présentation suggérée ici est à mon sens la meilleure car très lisible et revélatrice de la structure du bloc. Mis à part cette considération purement esthétique, un nouveau type nommé 'TLogiciel' (Rappel : par convention, les noms de types personnalisés, et en particulier les types enregistrement, commencent par un T majuscule, suivi d'un descriptif. C'est, au même titre que la présentation du code source, la convention adoptée par les créateurs de Delphi).
Concrètement, ce nouveau type permettra d'utiliser quatre membres : 'Nom', 'Version', 'Langue' et 'Taille' de types donnés dans la déclaration du type. On pourra ensuite déclarer des variables ou des constantes de ce type (note : il est possible mais vraiment peu recommandable de déclarer directement une variable de type enregistrement sans avoir créé ce type dans un bloc de déclaration de type. Il y a de bonnes raisons à cela : si vous voulez les connaître et voulez quand même utiliser cette possibilité, allez voir dans l'aide en ligne de Delphi.
En ce qui concerne les variables de type enregistrement, la variable en tant que telle ne nous intéresse pas vraiment, c'est plutôt son contenu qui nous intéresse. Pour accéder à un membre, il faut le *qualifier*, c'est-à-dire utiliser le nom d'une constante ou d'une variable qui contient un tel membre, la faire suivre d'un point et enfin du nom du membre. Ce membre se comporte alors respectivement comme une constante ou une variable du type déclaré pour ce membre dans la déclaration du type. Voici un exemple pour éclairer tout cela :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TLogiciel = record
    Nom,
    Version,
    Langue: string;
    Taille: integer;
  end;
var
  Log1, Log2: TLogiciel;
begin
  Log1.Nom := 'Delphi';
  Log2 := Log1;
end;
```

L'exemple ci-dessus déclare le type TLogiciel, puis deux variables de ce type. Mis à part la nouvelle forme de déclaration des types enregistrement, le principe reste le même qu'avec les autres types, à savoir qu'on définit au besoin un nouveau type, puis qu'on l'utilise dans la définition de variables ou de constantes (Nous verrons les constantes après cet exemple). Les deux instructions de la procédure sont des affectations. Dans la première, on affecte une chaîne au membre 'Nom' de la variable 'Log1' (cette variable est de type TLogiciel donc contient bien un membre nommé 'Nom'). 'Log1' étant une variable, 'Log1.Nom' est alors utilisable comme une variable, de type 'string' en l'occurence. On peut donc lui affecter une valeur chaîne.
La deuxième instruction est plus simple : c'est une affectation toute simple. La valeur d'une variable est affectée à une autre. Cette instruction affecte les valeurs de chacun des membres. On aurait pu faire la même chose en écrivant :

```
Log2.Nom := Log1.Nom;
Log2.Version := Log1.Version;
Log2.Langue := Log1.Langue;
Log2.Taille := Log1.Taille;
```

ce qui a pour inconvénient majeur d'être plus long et de poser un problème si on modifie le type 'TLogiciel'. Imaginez en effet qu'on ajoute un membre nommé 'Auteur'; il faudrait, dans l'exemple ci-dessus, ajouter une affectation, alors que l'affectation de Tab1 à Tab2 ne changera pas.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'exemple de type enregistrement donné est assez simple. Rien n'interdit d'utiliser des membres de types plus complexes. L'exemple ci-dessous est basé sur TLogiciel, mais introduit des changements intéressants.

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TLangueLogiciel = (llInconnue, llFrancais, llAnglais, llAutre);
  TLogiciel = record
    Nom,
    Version,
    Auteur: string;
    Langue: TLangueLogiciel;
    LangueNom: string;
    Taille: integer;
  end;
var
  Log1: TLogiciel;
begin
  Log1.Langue := llAnglais;
end;
```

Dans cet exemple, un nouveau type énuméré a été créé pour spécifier la langue d'un logiciel. On prévoit le cas où la langue ne serait pas parmi les cas prévus en donnant la valeur 'llAutre'. Dans ce cas, un membre 'LangueNom' de 'TLogiciel' sera utilisé pour stocker une chaîne décrivant la langue. Tout le reste n'est que très conventionnel.
En ce qui concerne les constantes de type enregistrement, l'exemple suivant devrait vous faire comprendre comment on procède pour les déclarer :

```
const
  LogParDef: TLogiciel =
    (Nom: '';
     Version: '';
     Auteur: '';
     Langue: llInconnue;
     LangueNom: '';
     Taille: 0);
```

On donne la valeur de chaque membre au moyen de : membre : valeur. La liste des valeurs est délimitée par des point-virgules et est mise entre parenthèses.
Le type enregistrement n'est évidemment pas un type ordinal, mais par contre, il peut être utilisé dans des tableaux (y compris dynamiques) et dans d'autres types enregistrements. Par exemple :

```
type
  TLangueLogiciel = (llInconnue, llFrancais, llAnglais, llAutre);
  TLangue = record
    Code: TLangueLogiciel;
    Nom: string;
  end;
  TLogiciel = record
    Nom,
    Version,
    Auteur: string;
    Langue: TLangue;
    Taille: integer;
  end;
  TTabLogiciel = array of TLogiciel;
```

Cet exemple montre déjà des types bien élaborés. Les deux premiers types sont respectivement énumérés et enregistrement. Le troisième ('TLogiciel') utilise comme type d'un de ses membres un type enregistrement. Si nous avions une variable 'Log1' de type 'TLogiciel', il faudrait écrire 'Log1.Langue.Code' pour accéder au code de la langue du logiciel. Le dernier type est encore un peu plus difficile : c'est un tableau dynamique, dont les éléments sont de type 'TLogiciel'. Imaginons que nous ayons une variable de ce type nommée 'Logiciels' : pour accèder au code de la langue du troisième logiciel de ce tableau, il faudrait écrire 'Logiciels[2].Langue.Code'. car 'Logiciels[2]' est de type 'TLogiciel' et 'Logiciels[2].Langue' est donc de type 'TLangue'.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce dernier type vous parait peut-être bien délicat, mais imaginez qu'avec ce type, on peut gèrer sans trop de peine une liste de logiciels, avec leurs particularités respectives, et tout cela avec *une seule variable*.

## VII-G-2 - Manipulation avancée des enregistrements

Il peut rapidement devenir fastidieux d'écrire le préfixe « *variable.* » devant les membres d'un enregistrement. Pour cela il existe une technique : l'utilisation d'un bloc **with**. Ce dernier permet, à l'intérieur d'un bloc spécifique, de manipuler les membres d'un enregistrement comme s'ils étaient directement accessibles (sans avoir à écrire « *variable.* »). Ce bloc est considéré comme une instruction unique. Voici sa syntaxe :

```
with enregistrement do
  instruction;
```

Le mot Pascal réservé **with** commence l'instruction. Vient ensuite une constante, un paramètre ou une variable de type enregistrement, puis le mot réservé do suivi d'une unique instruction. Cette instruction peut utiliser les membres de la donnée entre 'with' et 'do' sans passer par « *variable.* ». Mais il est possible de remplacer *instruction* par un *bloc d'instructions*. Un bloc d'instructions se comporte comme une instruction unique, mais est composé de plusieurs instructions. On utilise pour le créer les mots Pascal réservés **begin** et **end** entre lesquels on écrit les instructions, comme on le ferait pour une procédure ou une fonction. Voici un exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Log1: TLogiciel;
begin
  with Log1 do
    begin
      Nom := 'Delphi';
      Version := '5';
      Auteur := 'Borland/Inprise';
      Langue.Code := llAnglais;
      Langue.Nom := '';
      Taille := 275000;
    end;
end;
```

Dans l'exemple ci-dessus, TLogiciel est tel qu'il a été créé dans les exemples précédents. La procédure ne contient qu'une seule instruction : un bloc **with**. Ce bloc permet d'accèder aux membres de la variable Log1 sans qualifier ses membres. Le bloc est utilisé pour initialiser la variable. Il faut bien comprendre ici que chaque instruction à l'intérieur du bloc **with** devrait être adaptée pour fonctionner en dehors de ce bloc, mais que ce sont bel et bien des instructions réunies en une seule grâce aux deux mots réservés **begin** et **end**.
Il est dés lors possible, du fait que les instructions à l'intérieur d'un bloc **with** sont de vraies instructions, d'écrire un bloc **with** à l'intérieur d'un autre. Il faudra donner comme paramètre une donnée qui n'a rien à voir avec 'Log1' ou un membre de 'Log1' (**with** n'accepte que des enregistrements). Voici par exemple ce qu'il est possible de faire dans notre cas :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Log1: TLogiciel;
begin
  with Log1 do
    begin
      Nom := 'Delphi';
      Version := '5';
      Auteur := 'Borland/Inprise';
      with Langue do
        begin
          Code := llAnglais;
          Nom := '';
        end;
      Taille := 275000;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    end;
end;
```

Je n'en dirai pas plus, sauf qu'il faudra éviter d'utiliser ce type de bloc à tort et à travers : il faudra avoir au minimum 2 à 3 manipulations sur les membres d'un enregistrement pour passer par un bloc **with**.

## VII-H - Types et paramètres de procédures et fonctions

Tout ce verbiage sur les types nous a (presque) fait oublier que les types sont utilisés à un autre endroit important : dans les déclarations de fonctions et procédures, pour donner les types des paramètres et le type de résultat pour une fonction.
Et ici se pose un petit problème : une partie seulement des types que nous venons de voir sont acceptés pour les paramètres et pour le résultat d'une fonction sous Delphi 2 (Dans Delphi 5, tous les types sont autorisés). Je n'ai pas encore pu faire de tests sous Delphi 2, 3 et 4, donc il convient de se limiter aux types ordinaux et aux chaînes de caractères (ce qui inclue les types énumérés et les booléens).

## VII-I - Conclusion et retour sur terre

Au terme de ce chapitre consacré aux types de données, vous admettrez certainement que je vous ai ennuyé. C'est sûrement très vrai et je m'en excuse.
Ces types de données, pourtant, vous allez souvent les utiliser dans les programmes que vous écrirez. Le chapitre qui suit est davantage concentré sur les instructions en Pascal que sur les déclarations de toutes sortes. Vous apprendrez surtout qu'un programme est rarement exécuté de manière linéaire, mais que l'exécution peut « sauter » des instructions ou en répeter certaines.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# VIII - Structures de programmation en Pascal

Jusqu'à présent, lorsque nous avons écrit des instructions (dans une procédure ou dans une fonction, peu importe), celles-ci étaient exécutées les unes après les autres. Cela ne posait pas de problème puisque les exemples étaient choisis pour cela, mais rapidement, il va nous falloir structurer les instructions pour répondre à des besoins spécifiques tels des conditions sur une ou plusieurs valeurs, des parcours d'éléments de listes (par exemple ajouter 1 à tous les éléments d'un tableau dont les dimensions dépendent d'une donnée calculée pendant le programme, ...
Pour répondre à l'ensemble de nos besoins, un certain nombre de structures élémentaires permettent de répondre à tout. Il faudra parfois être astucieux et combiner ces structures pour en tirer un fonctionnement adéquat, mais il y aura moyen répondre quasiment à n'importe quel besoin. Ces structures se répartissent en deux familles : les structures conditionnelles et les structures itératives.
Chacune des structures que nous allons voir dans ce chapitre prend la place d'une unique instruction. Cette instruction structurée autour de mots réservés et d'une syntaxe particulière à chaque bloc contient la plupart du temps une ou plusieurs autres instructions ou blocs d'instructions (qui contiennent eux-mêmes des instructions). Il est à noter que tous ces blocs peuvent s'imbriquer les uns dans les autres, pour donner des comportements plus élaborés que ceux des blocs de base : ceci sera l'objet d'un paragraphe.
Ce chapitre est des plus importants, pourtant, malgré la pléthore de notions intéressantes et importantes qu'il présente, sa longueur également risque de vous paraître... importante. J'ai tenté de résoudre ce problème en insérant quelques exercices et manipulations.
Chacune des structures évoquées dans ce chapitre sera présentée indépendament des autres, pourtant, ces structures pourront être « imbriquées » les unes dans les autres et c'est précisément cet art de manipuler ces structures qui vous permettra de programmer efficacement.

## VIII-A - Structures conditionnelles

Le langage Pascal offre deux structures conditionnelles différentes : les structures **if** et les structures **case**. Elles sont employées à des endroits différents : tout dépend de ce que l'on veut faire. Les blocs **if** permettent de tester la valeur d'un booléen et d'exécuter une ou des instructions suivant que la valeur est vraie ou fausse. Les blocs **case**, bien que plus complexes, sont plus puissants puisqu'ils permettent de spécifier plusieurs cas, et même un cas général qui complète les autres cas. Ces deux structures vont faire chacune l'objet d'une étude détaillée.

### VIII-A-1 - Blocs 'if'

Les blocs **if** existent sous deux formes. La première, plus simple que l'autre, est :

```
if valeur_booleenne then
  instruction;
```

Ce bloc fonctionne de façon relativement simple (principe du si...alors) : si 'valeur_booleenne' vaut 'true', alors *instruction* est exécutée. *Valeur_booleenne* représente n'importe quoi qui donne un booléen en définitive, que ce soit directement un booléen, une comparaison (à l'aide des opérateurs de comparaison), le résultat d'une fonction, ou une combinaison d'un ou plusieurs de ces éléments avec les opérateurs booléens. *instruction* désigne au choix une instruction unique, ou alors un bloc d'instructions délimité par les désormais habituels **begin** et **end**.
Voici comme d'habitude un exemple utilisable sous Delphi (toujours au même endroit) :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DitBonjour: Boolean;
begin
  DitBonjour := false;
  if DitBonjour then
    ShowMessage('Bonjour !');
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cet exemple est des plus simples : une variable booléenne est déclarée et initialisée. Un bloc if utilise la valeur de cette variable comme condition. Si la valeur de départ est 'true', le message est affiché (sinon, il ne l'est pas). Essayez cet exemple sous Delphi en changeant la valeur de départ de 'DitBonjour' de 'false' à 'true'.
Cet exemple manquant un peu de piment, nous allons un peu le compliquer :

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MotDePasse = 'machin';
var
  Reponse: string;
begin
  Reponse := '';
  InputQuery('Attention', 'Entrez le mot de passe', Reponse);
  if Reponse = MotDePasse then
    ShowMessage('Mot de passe correct');
end;
```

Cet exemple fait usage de la fonction 'InputQuery'. Cette fonction, qui montre une fenêtre contenant une zone d'édition, du texte, et deux boutons 'OK' et 'Annuler', permet de poser une question à l'utilisateur. Voici celle qui apparaîtra sur votre écran :

Elle admet 3 paramètres : le premier est le texte apparaissant dans la barre de titre de la fenêtre. Le second est le texte affiché juste au-dessus de la zone d'édition. Le troisième est un peu particulier dans le sens où on doit absolument transmettre une variable et pas une valeur. Cette variable, de type chaîne, est utilisée de deux manières : la valeur de la variable est affichée dans la boite de dialogue lorsque celle-ci est montrée à l'utilisateur. La valeur de la variable sera ensuite modifiée par la fonction et recevra le texte entré par l'utilisateur si ce dernier clique sur 'OK'. La fonction renvoie un résultat booléen : si l'utilisateur clique sur 'OK', la fonction renvoie 'true', sinon elle renvoie 'false'.
Hormis l'usage de cette fonction, la procédure initialise d'abord la variable 'Reponse'. Ainsi, rien ne sera affiché dans la zone d'édition de la boite de dialogue. Vient ensuite un bloc if : le contenu de 'Reponse' est comparé à la constante 'MotDePasse'. Si les deux chaînes sont identiques, alors un message indique à l'utilisateur que le mot de passe est correct (si l'utilisateur donne un mauvais mot de passe, ou s'il annule, les deux chaînes seront différentes et rien ne se passera).
L'exemple ci-dessous est encore un peu plus délicat :

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MotDePasse = 'machin';
var
  Reponse: string;
  DonneReponse: boolean;
begin
  Reponse := '';
  DonneReponse := InputQuery('Attention', 'Entrez le mot de passe', Reponse);
  if not DonneReponse then
    ShowMessage('Vous deviez indiquer un mot de passe !');
  if DonneReponse and (Reponse = MotDePasse) then
    ShowMessage('Mot de passe correct');
  if DonneReponse and (Reponse <> MotDePasse) then
```

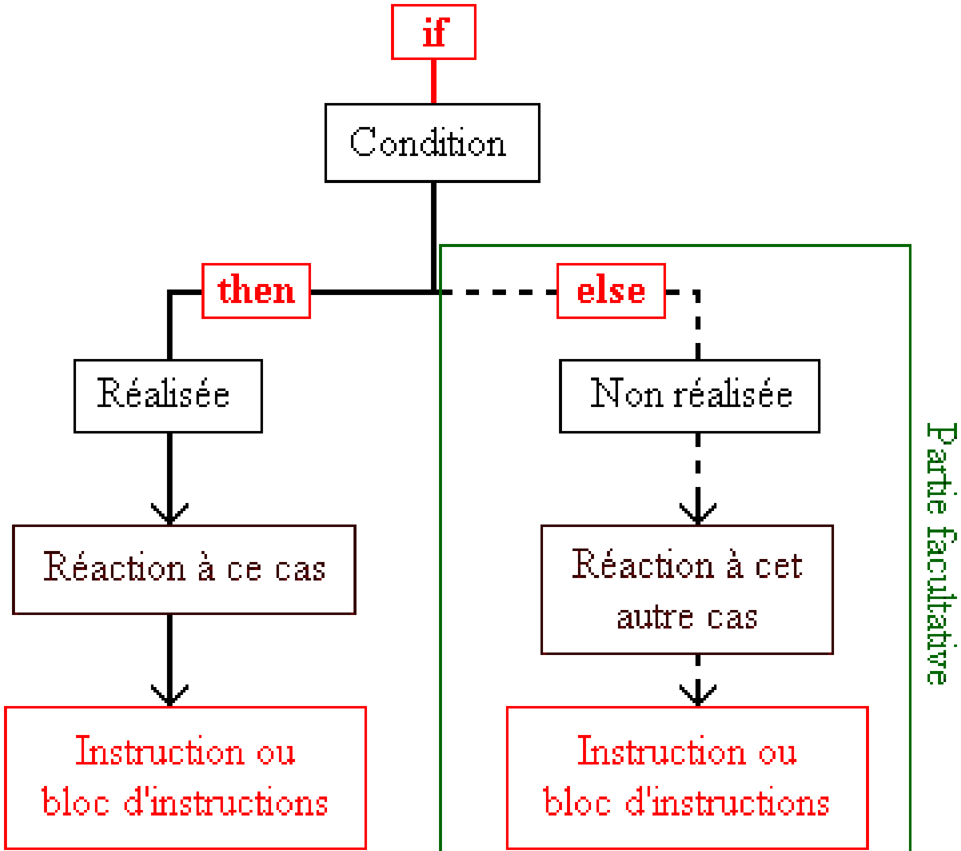

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    ShowMessage('Mot de passe incorrect');
end;
```

Le premier bloc if utilise un opérateur booléen pour obtenir la négation de 'DonneReponse' : on montre le message d'erreur seulement lorsque 'not DonneReponse' vaut 'true', c'est-à-dire lorsque 'DonneReponse' vaut 'false' (habituez-vous dés à présent à ces raisonnements logiques, ils sont très fréquents et utiles dans les blocs **if**). Le deuxième bloc **if** utilise une condition plus fine puisqu'il y a un opérateur booléen entre deux valeurs booléennes : la premiere est la valeur d'une variable booléenne, tandis que la seconde est le résultat de la comparaison entre 'Reponse' et 'MotDePasse'. Le message n'est donc affiché que si l'utilisateur donne une réponse, et si sa réponse correspond au bon mot de passe. Le troisième bloc **if** est du même genre mais le message n'est affiché que lorsque l'utilisateur donne un mot de passe éronné.
Il est maintenant opportun de voir la deuxième forme des blocs **if**. Cette forme permet de réagir avec le principe suivant : si...alors...sinon :

```
if valeur_booleenne then
  instruction_si_vrai
else
  instruction_si_faux;
```

*instruction_si_vrai* et *instruction_si_faux* sont chacun soit une instruction seule soit un bloc d'instructions. Les instructions incluses dans *instruction_si_vrai* sont exécutées si la *valeur_booleenne* est 'true', tandis que celles incluses dans *instruction_si_faux* sont exécutées dans le cas contraire (si *valeur_booleenne* est 'false'). Cette forme permettra de réagir différemment suivant les deux valeurs en une seule instruction. (Il est à noter que la même chose serait possible en utilisant deux blocs **if** qui testeraient successivement une valeur et sa négation par l'opérateur **not**. L'utilisation de **if...then...else** sera pourtant préférée à l'utilisation de deux blocs if)
Voici le schémas de fonctionnement d'un bloc **if** :





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce schéma comporte ce qui se rapporte au langage en rouge (les mots-clés sont en gras), la partie facultative (else) marquée en vert, ainsi que le principe général indiqué en noir : suivant une condition, il y a branchement vers une instruction ou bloc d'instruction différente suivant que la condition est réalisée ou pas.
Le dernier exemple a été réécrit ci-dessous en utilisant la nouvelle syntaxe (avec partie 'else') :

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MotDePasse = 'machin';
var
  Reponse: string;
  DonneReponse: boolean;
begin
  Reponse := '';
  DonneReponse := InputQuery('Attention', 'Entrez le mot de passe', Reponse);
  if DonneReponse then
    begin
      if Reponse = MotDePasse then
        ShowMessage('Mot de passe correct')
      else
        ShowMessage('Mot de passe incorrect');
    end
  else
    ShowMessage('Vous deviez indiquer un mot de passe !');
end;
```

Cet exemple montre plein de nouveautés : deux blocs if...then...else sont utilisés, et sont même imbriqués l'un dans l'autre, ce qui est tout à fait autorisé. le premier bloc teste la valeur de 'DonneReponse'. Si 'DonneReponse' est vraie, un nouveau bloc if est rencontré : selon que l'utilisateur donne ou pas le bon mot de passe, le message affiché change. Enfin, si 'DonneReponse' est faux, un message d'erreur est affiché.
Vous vous étonnez peut-être de voir qu'on a écrit un bloc d'instructions pour le premier if, et pourtant, une seule instruction (un autre bloc if) y est présente. Ceci est peu recommandable dans le cas général puisque le begin et le end ne servent alors à rien. Ils sont bel et bien indispensables dans les premières versions de Delphi, mais deviennent facultatifs dans Delphi 5. Le bloc if complet devient alors :

```
if DonneReponse then
  if Reponse = MotDePasse then
    ShowMessage('Mot de passe correct')
  else
    ShowMessage('Mot de passe incorrect')
else
  ShowMessage('Vous deviez indiquer un mot de passe !');
```

Dans ce cas, il est assez facile de voir que le bloc entier est une seule et unique instruction : pas un seul point-virgule dans tout le bloc.
Notez enfin qu'il est possible d'enchaîner plusieurs de ces blocs en utilisant la partie **else** de l'instruction. Il sera alors possible d'écrire quelque chose du genre :
**if ... then ... else if ... then ... else if ... then ... else ...**
Ceci permettra d'envisager successivement plusieurs situations, dont une seule sera considérée. Il sera possible de donner une situation "poubelle" avec un else final, qui s'exécutera dans le cas où aucun des autres cas ne se présenterait. Voici un exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Reponse: string;
begin
  Reponse := '';
  InputQuery('Attention', 'oui ou non ?', Reponse);
  if (Reponse = 'oui') or (Reponse = 'OUI') then
    ShowMessage('D''accord, c''est oui !')
  else if (Reponse = 'non') or (Reponse = 'NON') then
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
      ShowMessage('Tant pis, c''est non...')
    else
      ShowMessage('Vous pourriez donner votre avis !');
  end;
```

Voici maintenant deux exercices. Ces exercices sont faits pour vous faire écrire tout d'abord un bloc **if**, puis des fonctions utilisant des blocs **if**. Aucun d'entre eux ne nécessite d'écrire des blocs d'instructions, des instructions seules sont suffisantes. Vous pourrez cependant compliquer à loisir les exercices si cela vous fait envie. Réussir à résoudre ces exercices n'est pas essentiel mais le fait d'essayer de les résoudre et de bien comprendre la solution proposée est par contre absolument indispensable.

Exercice 1 : (voir la **solution et les commentaires**).
En vous servant de la désormais habituelle procédure 'TForm1.Button1Click', écrivez les instructions qui permettront de réaliser ce qui suit :
*Données du problème :*

- la fonction 'sqrt' (« Square Root », « racine carrée » en français) admet un unique paramètre de type 'extended' et renvoie un nombre de type 'extended' qui est la racine carrée du paramètre (Exemple : sqrt(4) donnera 2).
- la fonction 'StrToFloat' (Chaine vers nombre à virgule) admet un unique paramètre de type chaîne, et renvoie un nombre de type 'extended' qui est le nombre représenté par le paramètre (Exemple : StrToFloat('1.2') donnera 1.2).

Fonctionnement souhaité :

1. Un nombre est demandé à l'utilisateur (en utilisant 'InputQuery'), on admet ici que c'est toujours un nombre qui sera rentré. Si c'est autre chose qu'un nombre, une erreur se produira, mais ceci ne concerne pas cet exercice et est admis pour le moment.
2. Si ce nombre est négatif, un message s'affiche alors, indiquant cela à l'utilisateur, sinon, la procédure affiche la racine carrée de ce nombre (mes excuses en passant à ceux qui n'aiment pas les maths).

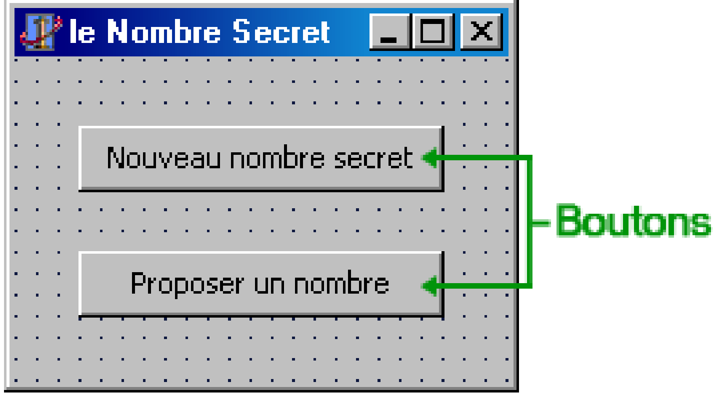
Exercice 2 : (voir la **solution et les commentaires**).
Dans chacune des questions suivantes, afin de pouvoir vérifier votre travail, il faudra utiliser la procédure 'TForm1.Button1Click'. Le principe du test consistera à déclarer et initialiser une variable (en lui donnant une valeur que vous changerez pour examiner les divers cas possibles), qui sera transmise à la fonction à tester. Le résultat sera affiché à l'écran comme nous l'avons déjà fait de nombreuses fois. Chacune des fonctions demandées doit être écrite dans l'unité 'Principale' du projet 'PremierEssai'.

1. Ecrire une fonction « NbNegatif » qui prend un nombre comme paramètre. Le résultat de la fonction est une chaîne de caractères dépendant de ce nombre : si le nombre est strictement négatif (non nul), la fonction renvoie « négatif », sinon, elle renvoie « positif ».
2. Ecrire une fonction « SigneNombre » qui prend un nombre comme paramètre. Le résultat de la fonction est une chaîne de caractères dépendant de ce nombre : si le nombre est strictement positif, le résultat est « plus », si le nombre est strictement négatif, le résultat est « moins », enfin, si le nombre est 0, le résultat est « zéro ».

**Mini-projet n°1**
(voir **Indications, Solution et Téléchargement**)
Vous avez maintenant les compétences requises pour réaliser un premier mini-projet. Celui-ci consiste en la réalisation d'un petit logiciel que vous connaissez certainement : la recherche d'un nombre secret. Vous devez créer un nouveau projet et en faire une application répondant aux exigences ci-dessous :

1. L'interface est, pour cette fois, imposée. Elle doit être similaire à ceci (seuls les textes sont à respecter) :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




2 Le bouton « Nouveau nombre secret » fixe un nombre secret entier entre 0 et 100. Ce nombre est choisi par l'ordinateur, et n'est ni demandé ni montré à l'utilisateur.
3 Le bouton « Proposer un nombre » demande un nombre entier à l'utilisateur. Si le nombre entier n'est pas compris entre 0 et 100, l'utilisateur en est informé. Sinon, la position par rapport au nombre secret est indiquée :
« Le nombre secret est inférieur/supérieur à (le *nombre proposé par l'utilisateur*) »
4 Lorsque l'utilisateur trouve le nombre secret, un message l'en informe : « bravo, le nombre secret était (*indiquer le nombre secret*) », puis le nombre secret est changé et l'utilisateur est informé de ce changement et invité à rejouer : « Nombre secret changé, vous pouvez rejouer ».
Ce mini-projet étant le premier, il est possible que vous ayez du mal à le réaliser. Dans ce cas, n'hésitez pas à consulter les indications, puis le guide pas à pas, ou en dernier recours à télécharger le mini-projet terminé (auquel vous avez accès même si vous avez la version téléchargée du site).
*Rappels et indications indispensables :*

- Pour pouvoir réagir au clic sur un bouton, il faut faire un double-clic dessus dans Delphi, puis écrire les instructions dans la procédure qui est créée à cet effet (Ecrire la procédure et la déclarer sans passer par cette étape du double-clic ne suffit en aucun cas).

## VIII-A-2 - Blocs 'case'

Les blocs **case** fonctionnent sur un autre principe : elle permet d'examiner la valeur d'une donnée et de décider d'une instruction éventuelle à exécuter suivant les cas. Les blocs **case** permettront aussi parfois de simplifier des blocs **if** trop complexes, mais le principe est différent : il s'agit de choisir parmi plusieurs cas possibles et non de prendre une décision comme dans un bloc **if**. Voici la syntaxe générale d'un bloc **case** :

```
case variable_ordinale of
  cas1: instruction1;
  [cas2: instruction2;]
  {...}
  [casn: instructionn;]
  [else instruction;]
end;
```

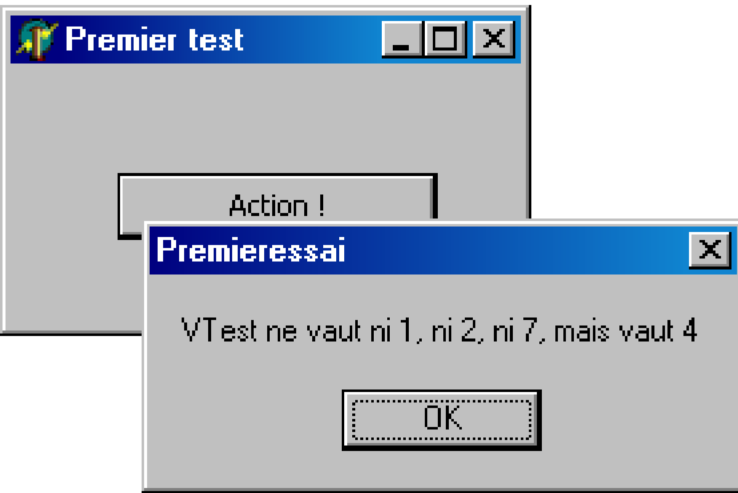
(Ne vous fiez pas à l'aspect un peu barbare de cette syntaxe)
Un bloc **case** permet d'exécuter au plus une des instructions ou bloc d'instructions présents dans le bloc (ce qui signifie que si l'un des cas est réalisé, l'instruction ou bloc d'instructions qui lui correspond sera exécutée, mais que rien ne sera exécuté si aucun des cas n'est réalisé). Les *cas1, cas2 ... casn* permettent de spécifier des valeurs, ou des intervalles de valeurs, ou une liste de ces derniers séparés par des virgules. Si la valeur de *variable_ordinale* est dans l'un de ces cas, l'instruction ou le bloc d'instructions correspondant est alors exécuté (celle ou celui qui suit immédiatement l'énoncé du cas). Vous pouvez en outre spécifier un cas "complémentaire", désigné par **else**, et qui permet de donner une instruction ou un bloc d'instruction exécuté si aucun des autres cas n'est réalisé (notez qu'il n'y a pas de ':' entre **else** et l'instruction ou le bloc d'instructions correspondant). Comprenez bien ici que l'instruction


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

présente après **else** n'est pas exécutée si un des autres cas est exécuté, mais exécutée dans le cas contraire : ceci permet de s'assurer que toutes les valeurs possibles pour la donnée seront couvertes.
Voici un premier exemple d'un bloc **case** :

```
function Random_1_10: Integer;
begin
  Result := Trunc(Random(10) + 1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  VTest: Integer;
begin
  VTest := Random_1_10;
  case VTest of
    1: ShowMessage('VTest vaut 1');
    2: ShowMessage('VTest vaut 2');
    7: ShowMessage('VTest vaut 7');
    else
      ShowMessage('VTest ne vaut ni 1, ni 2, ni 7, mais vaut ' + IntToStr(VTest));
  end;
end;
```

La fonction 'Random_1_10' génère simplement un nombre quelconque entre 1 et 10. Venons-en à la procédure 'TForm1.Button1Click'. Une variable 'VTest' est déclarée et initialisée à une valeur entre 1 et 10. Vient ensuite un bloc **case** qui traite quelques valeurs (1, 2 et 7) et qui traite les autres dans un cas **else**. Chaque cas ne comporte ici qu'une simple instruction, mais on aurait pu mettre à chaque fois un bloc d'instruction contenant d'autres structures (**with** et **if**, pour ne citer que celles que vous connaissez). Le détail des instructions est simple pour les trois premiers cas, et utilise 'IntToStr' et une concatènation pour construire un message dans le dernier cas (cas "complémentaire"). Essayez ce morceau de code sous Delphi : vous verrez que dans 3 cas (1, 2 et 7), vous avez une fenêtre indiquant la valeur, sinon vous aurez un message indiquant que la valeur n'est ni 1, ni 2, ni 7, et qui donnera ensuite la valeur de 'VTest'. Par exemple :




L'utilisation des blocs **case** est possible, comme nous l'avons vu, avec tous les types ordinaux. Ceci est particulièrement intéressant avec les types énumérés car on peut alors choisir un comportement suivant chacune des constantes d'un de ces types, ou suivant des groupes de valeurs. Examinons un petit exemple :

```
type
  TTypeSupport = (tsInconnu, tsDisq35, tsDisqueDur,
    tsCDRom, tsDVD, tsGraveur, tsZIP);

procedure TestSupport(Supp: TTypeSupport);
begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    case Supp of
      tsDisq35, tsZIP:
        ShowMessage('Type de média : Disquettes');
      tsCDRom, tsDVD, tsGraveur:
        ShowMessage('Type de média : Compact-Discs');
      else
        ShowMessage('Type de média : (non amovible)');
    end;
  end;

  procedure TForm1.Button1Click(Sender: TObject);
  begin
    TestSupport(tsDisq35);
  end;

  procedure TForm1.Button2Click(Sender: TObject);
  begin
    TestSupport(tsGraveur);
  end;

  procedure TForm1.Button3Click(Sender: TObject);
  begin
    TestSupport(tsDisqueDur);
  end;
```

(Il est préfèrable de télécharger le code source du projet en **cliquant ici**)
Cet exemple montre l'utilisation d'un type énuméré avec une structure **case**. 3 cas sont envisagés. Les deux premiers donnent une liste de valeurs de type TTypeSupport (qui est le type du paramètre "Supp" examiné). Dans chacun de ces deux cas, si la valeur de "Supp" correspond à une des valeurs données, l'instruction correspondante au cas est exécutée et l'exécution continue après le bloc **case** (contrairement à d'autres langages comme le C où il aurait fallu interrompre l'exécution du bloc au moyen d'une instruction "break;"). Le dernier cas complémentaire **else** donne une réaction pour tous les cas non examinés ailleurs dans le bloc.
Cet exemple utilise trois boutons : chacun lance une fonction identique avec un paramètre différent pour tester les différents cas. Le bloc **case** examine la valeur de "Supp" et décide, suivant que le périphérique accepte les disquettes, les CD ou rien, d'afficher un message à l'utilisateur. Il est à noter que comme dans l'exemple précédent, la donnée examinée (à savoir ici le paramètre "Supp") doit être utilisée comme une constante à l'intérieur du bloc **case**.
Globalement, les blocs **case** sont plus rarement utilisés que les blocs **if**. Ces derniers blocs sont en effet plus intéressants que les blocs **case**. En effet, lorsque vous devrez examiner non pas une condition, mais deux conditions combinées par un ET logique, les blocs **case** deviendront inadéquats.
Vous en savez maintenant assez sur les blocs **case**, ainsi que sur les blocs **if**. Ces structures dites « conditionnelles » sont à la base de la programmtion en Pascal. Il va maintenant falloir examiner un autre type de bloc d'instructions : les structures dites « itératives ».

## VIII.B - Structures itératives

En programmation, il est souvent utile d'exécuter un certain nombre de fois la même instruction ou le même bloc d'instructions en y apportant une légère variation. La répétition controlée permet de résoudre de nombreux problèmes, comme le parcours de tous les éléments d'un tableau par exemple. Imaginons un tableau de 100 éléments :

```
var
  TabTest: array[1..100] of integer;
```

Imaginons maintenant que nous ayons besoin d'ajouter, disons 2 à chacune des 100 valeurs contenues dans ce tableau. Nous pourrions écrire :

```
TabTest[1] := TabTest[1] + 2;
...
TabTest[100] := TabTest[100] + 2;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ce qui aurait effectivement pour effet d'ajouter 2 à chaque valeur du tableau. Mais ce genre d'écriture prendrait 100 lignes, du temps, et ne serait pas très passionnante. Imaginons maintenant, pour enfoncer le clou, que 'TabTest' devienne :

```
var
  TabTest: array[1..Max_V] of integer;
```

où 'max_v' est une constante. Il sera alors impossible d'écrire les max_v lignes, puisqu'il faudrait supposer la valeur de 'max_v', ce qui empécherait sa modification ultérieure, rendant la constante parfaitement inutile. Pour nous sortir de ce mauvais pas, le langage Pascal met à notre disposition des structures permettant de résoudre le problème de façon très simple. C'est ce qu'on appelle des structures itératives.

## VIII-B-1 - Blocs 'for'

***Note aux programmeurs en C ou C++***
*La boucle 'for' du 'Pascal est nettement moins puissante que celle que vous connaissez en C (ou en C++). Il est ici hors de question de donner une condition de sortie pour la variable de contrôle ou de contrôler l'incrémentation de cette variable. Pour faire cela, il vous faudra employer les structures 'while' et 'repeat' de Pascal Objet.*

La boucle 'for' est certainement la structure itérative la plus simple à comprendre pour un débutant en programmation. Cette boucle permet de répéter une instruction ou un bloc d'instructions en faisant varier la valeur d'une variable entre deux valeurs minimales et maximales. Le bloc a besoin pour fonctionner d'une variable de type ordinal (on utilisera souvent un type 'integer') qui parcourra toutes les valeurs comprises entre les deux valeurs minimales et maximales données. Voici la syntaxe du bloc :

```
for variable_ordinale := valeur_minimale to valeur_maximale do
  instruction;
```

Dans ce bloc, *variable_ordinale* est une variable de type ordinal, que l'on appellera « variable de contrôle », *valeur_minimale* et *valeur_maximale* sont deux valeurs de même type ordinal que *variable_ordinale*. L'*instruction* (ou le bloc d'instruction) suivant le mot réservé **do** constitue le corps de la boucle : la ou les instructions qui y sont présentes seront exécutées un nombre de fois égal à *valeur_maximale* - *valeur_minimale* + 1 (dans le cas ou ces valeurs sont des entiers). Il va de soi que *valeur_minimale* doit être inférieure (mais pas forcément strictement) à *valeur_maximale*. Si ce n'est pas le cas, l'intérieur du bloc ne sera pas exécuté.
Examinons un premier petit exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  indx: integer;
begin
  for indx := 1 to 3 do
    ShowMessage('Salut !');
end;
```

Exécutez ce morceau de code : le message 'Salut !' sera affiché 3 fois à chaque clic sur le bouton. Une variable 'indx' est déclarée et utilisée comme variable de contrôle de la boucle 'for'. Les valeurs minimales et maximales choisies sont respectivement 1 et 3. L'équivalent de

```
for indx := 1 to 3 do
  ShowMessage('Salut !');
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

est

```
indx := 1;
ShowMessage('Salut !');
indx := 2;
ShowMessage('Salut !');
indx := 3;
ShowMessage('Salut !');
```

Mais nous n'utilisons pas ici l'une des possibilités les plus intéressantes de la boucle 'for' : la variable de contrôle est accessible depuis l'intérieur de la boucle, mais en tant que constante seulement, il est interdit de la modifier : comprenez bien ceci, on regarde mais on ne touche pas ! Si vous tentez de modifier une variable de contrôle à l'intérieur d'une boucle 'for', la compilation devrait normalement vous signaler une erreur, et si elle ne le fait pas, votre programme a toutes les chances d'aller s'écraser contre un arbre (virtuel, bien entendu). Pour simplifier, imaginez que la variable de contrôle doit être utilisée comme un simple paramètre d'une fonction ou procédure : on peut utiliser sa valeur mais en aucun cas prétendre à la modifier.
Ces recommandations faites, et, je l'espère, bien comprises, revenons à notre exemple de tableau de 100 éléments :

```
var
  TabTest: array[1..100] of integer;

procedure TForm1.Button1Click(Sender: TObject);
var
  indx: integer;
begin
  for indx := 1 to 100 do
    TabTest[indx] := TabTest[indx] + 2;
end;
```

Cet exemple est des plus importants car il montre quelle utilisation on peut faire non seulement d'une boucle **for**, mais également de la variable qui est mise à jour à chaque itération (chaque exécution du bloc d'instruction qui constitue l'intérieur de la boucle). La variable inx est ici utilisée comme variable de contrôle, puis utilisée à l'intérieur de la boucle. Notez tout d'abord qu'en aucun cas on ne tente de modifier sa valeur? La variable est utilisée pour désigner une case du tableau. Ainsi, lorsque la variable parcourera les valeurs successives de 1 à 100, les cases successives de 1 à 100 seront modifiées comme désiré, à savoir que la valeur 2 leur sera ajoutée (il est à noter ici qu'aucune des valeurs contenues dans le tableau n'a été initialisée, on suppose en effet pour l'exemple qu'une autre partie du programme s'en est chargé avant).
Il est possible, dans une boucle **for**, d'utiliser comme valeurs limites (minimales et maximales) des valeurs de paramètres, de variables, de constantes, de résultats de fonctions, ou de calculs réalisés à partir de ces derniers. C'est très pratique lorsqu'on ne connait pas les valeurs limites au moment de l'écriture d'un programme, il suffira alors de donner la valeur sous forme d'une expression qui donnera la valeur désirée. Voici un petit exemple théorique pour illustrer cela :
Essayons maintenant un exemple plus intéressant : le calcul d'une factorielle. Je rappelle pour le groupe des non-matheux (dont je fais désormais partie :) que la factorielle d'un nombre entier *n* (notée « n! ») s'obtient par la multiplication suivante :
n! = 1 x 2 x 3 x ... x (n - 1) x n

1. Ecrivez la fonction **for** capable de calculer la factorielle d'un entier compris entre 1 et 69. Cette fonction utilisera une boucle **for** dont la variable de contrôle sera nommée « indx » et le résultat « Fac_n ». L'unique paramètre sera nommé « n » (type entier) et le résultat de type entier. Vous aurez besoin d'utiliser la valeur du paramètre « n » en tant que valeur maximale dans la boucle. Pensez bien à initialiser « Fac_n » avant de débuter le calcul et à filtrer les valeurs de « n » pour n'accepter que celles qui conviennent (la fonction renverra -1 en cas d'erreur). Jetez ensuite un oeil sur la correction donnée ci-dessous :

```
function Facto(n: integer): integer;
var
  indx, Fac_n: integer;
begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    result := -1;
    if (n >= 1) and (n &lt;= 69) then
      begin
        Fac_n := 1;
        for indx := 2 to n do
          Fac_n := Fac_n * indx;
        result := Fac_n;
      end;
  end;
```

Cette fonction est déjà un peu plus difficile à écrire que toutes les précédentes. En effet, il s'agit d'inclure le calcul de la factorielle dans un bloc **if** qui garantit que le nombre dont on calcule la factorielle est compris entre 1 et 69 : c'est ce qui est fait ci-dessus. Le calcul proprement dit est effectué à l'aide d'une boucle **for**. Le principe est le suivant : on doit faire une multiplication qu'on ne peut pas poser directement puisqu'on ne connait pas la valeur de 'n'. On doit pour cela passer par une décomposition du calcul en petites étapes : des multiplications simples par un seul nombre. La solution est alors de multiplier successivement par toutes les valeurs entre 2 (multiplier par 1 n'est pas très intéressant) et la valeur de 'n'. 'Fac_n' est alors initialisé à 1, puis multiplié par les valeurs successives de 'indx' entre 2 et n, ce qui donne bien la factorielle de n. Ne surtout pas oublier à la fin d'un tel calcul de fixer le résultat de la fonction à la valeur calculée (Note importante : nous aurions pu tout aussi bien utiliser directement 'result' en lieu et place de 'Fac_n' dans la fonction, mais j'ai délibérément évité pour ne pas trop vite embrouiller les choses ;=).
C'est déjà tout pour ce paragraphe consacré aux boucles **for**. Les exemples d'utilisation de ces boucles sont innombrables et vous aurez l'occasion d'en voir certain au cours des mini-projets qui suivront dans le guide. Sachez cependant qu'en programmation, l'utilisation des boucles **for** est moins répandu qu'on ne pourrait croire, car elles imposent un nombre d'itérations (de passages) au moins virtuellement connu (*Valeur_maximale* - *Valeur_minimale* + 1). Il est souvent utile de répèter une instruction ou un bloc d'instructions jusqu'à ce qu'une condition soit remplie : c'est le rôle des deux structures **while** et **repeat**.

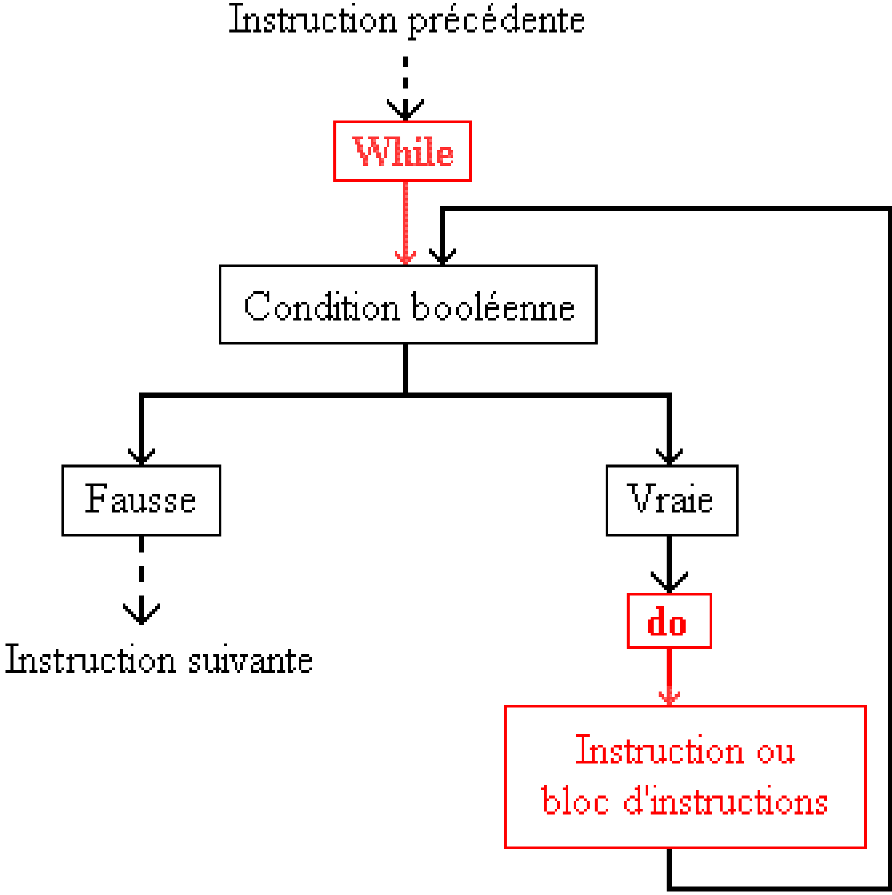
## VIII-B-2 - Blocs 'while'

Les blocs **while**, qu'on appelera aussi boucle **while**, sont plus puissants mais plus délicates à manipuler que les boucles **for**, tout en étant réservées à d'autres usages. Le principe d'un bloc **while** est de tester une valeur booléenne et d'exécuter une instruction ou un bloc d'instruction si elle est vraie. La boucle se termine dés que la valeur booléenne est fausse (l'instruction ou le bloc d'instructions qui constitue le corps de la boucle peut ne pas être exécuté du tout).
Voici la syntaxe de ce bloc :

```
while condition_booleenne do
  instruction; { ou bloc d'instruction }
```

Voici maintenant le schéma de fonctionnement d'une boucle **while** :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




La condition booléenne employée dans ce bloc peut être n'importe quelle expression dont le type est booléen : un booléen seul, une ou des opérations logiques entre booléens, des comparaisons de valeurs, des résultats de fonctions, des paramètres... Tout est permis du moment que l'ensemble forme un booléen. Nous aurons l'occasion de travailler cette partie au fur et à mesure de la progression dans le guide car les boucles **while** sont très largement employées en Pascal.
Le test sur la condition effectué, deux actions sont possibles : si le booléen est faux (false), alors le bloc se termine, sinon, si le booléen est vrai (true), alors l'instruction ou le bloc d'instruction est exécuté. A la fin de l'exécution de cette instruction ou de ce bloc d'instructions, la condition booléenne est évaluée à nouveau et ainsi de suite. Une conséquence immédiate à comprendre est que contrairement aux boucles **for** où la variable de contrôle devait être manipulée comme une constante, il faudra ici faire bien attention à modifier le résultat de la condition booléenne à l'intérieur de la boucle, ou sinon, on se retrouvera avec une boucle infinie qui plantera l'ordinateur.
Voici un exemple de ce qu'il ne faut pas faire :

```
while i > 0 do
  a := a + 2;
```

Vous voyez où est l'erreur ? Le problème consiste en le fait que la condition (i > 0) sera évaluée une fois : si elle est fausse, alors tout ira bien, sinon, l'instruction s'exécutera, ne modifiant pas i. La valeur de i n'étant pas modifiée, la condition sur i sera encore vraie, ce qui boucle un cycle sans fin, entrainant un blocage du programme et probablement un plantage de l'ordinateur.
Mais que cela ne vous empêche pas d'utiliser les boucles **while** ! Elles sont très utiles et souvent indispensables. Voici le même exemple, mieux écrit :

```
i := 10;
while i > 0 do
  begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    a := a + 2;
    i := i - 1;
  end;
```

Dans l'exemple ci-dessus, on ne se lance pas tête baissée dans l'exécution de l'instruction (a := a + 2). On commence par initialiser i, et on prend bien soin de le modifier à l'intérieur de la boucle : l'effet ici sera d'enlever 1 à i à chaque itération de la boucle, ce qui aura pour conséquence de faire prendre la valeur 0 à i au bout de 10 itérations, arrètant là l'exécution de la boucle. Nous avons en fait recréé ici l'équivalent d'une boucle **for** avec une boucle **while**, ce qui montre bien que cette dernière est plus puissante.
Ce dernier exemple n'utilise pas le principal intérêt des boucles **while**, car on peut encore prévoir dans cet exemple le nombre d'itérations. Considérons un autre exemple plus pertinent :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Reponse: string;
const
  RepQuit = 'q';
begin
  Reponse := '';
  while Reponse <> RepQuit do
    begin
      InputQuery('', 'quelque chose à dire ?', Reponse);
      if Reponse <> RepQuit then
        ShowMessage(Reponse);
    end;
end;
```

Dans cet exemple, un clic sur le bouton initialise 'Reponse' à la chaine vide, puis lance une boucle **while**. Cette boucle s'arrète à la condition que la valeur de 'Reponse' soit la chaîne « q ». Ce qui se passe dans la boucle est assez simple : on demande à l'utilisateur de taper quelque chose, que l'on stocke justement dans 'Reponse', puis on affiche ce que l'utilisateur vient de taper. Le seul moyen de quitter la boucle est de taper la chaîne « q ». Le bloc **if** permet d'éviter que cette chaîne soit répètée à l'utilisateur. La boucle se termine alors car la condition 'Reponse <> RepQuit' devient fausse.
Comme pour les boucles 'for', d'autres exemples viendront en leur temps au fur et à mesure du guide.
Nous allons maintenant nous attaquer à une manipulation guidée pas à pas : suivez les indications ci-dessous sous Delphi pour suivre les explications (si vous n'arrivez pas à réaliser toutes ces opérations, n'ayez pas peur : vous pouvez télécharger le projet créé pendant la manipulation ci-dessous) :
L'objectif de cette manipulation est de recréer le projet « Nombre secret » d'une manière plus rapide et plus agréable autant pour nous que pour l'utilisateur. Je préfère vous prévenir : cette manipulation va peut-être vous paraître difficile. L'essentiel ici est de comprendre l'utilisation de la boucle **while** et de l'utilité des imbrications de blocs que nous allons construire.

1. Créez un nouveau projet (que vous enregistrerez immédiatement, ainsi que la seule unité et sa fiche associée, en trouvant vous-même des noms adéquats), placez un unique bouton sur la fiche principale, et dimensionnez tout cela à votre goût. Saisissez comme texte du bouton : « Jouer ! » (Pour ceux qui ne se rappeleraient plus comment on fait, il faut sélectionner le bouton à l'aide d'un simple clic, puis aller dans l'inspecteur d'objets, dans la page 'Propriétés' ('Properties' en anglais), et modifier la ligne intitulée 'Caption'). De la même manière, changez le titre de la fenêtre à « Le nombre secret ».
2. Créez la procédure répondant à un clic sur le bouton (en effectuant un double-clic sur ce dernier). Le principe va être ici de tout faire en une seule procédure : la première étape sera de générer un nombre secret, la deuxième de lancer la phase interactive avec l'utilisateur (la seule qu'il verra), tout en pensant à garder à cet utilisateur une porte de sortie. Nous aurons besoin pour écrire cette procédure d'une boucle **while** et d'un gros bloc **if**.
3. Le nombre secret sera stocké dans une variable interne nommée 'NbSec'. Déclarez cette variable de type integer :

```
var
  NbSec: integer;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

puis écrivez l'instruction qui génèrera un nombre secret entre 0 (inclus) et 100 (inclus) :

```
NbSec := Trunc(Random(101));
```

4 Comme dans tout programme où l'utilisateur intervient, il va nous falloir deux variables de plus : 'Reponse' de type chaîne de caractères et 'NbPropose' de type integer. Déclarez ces deux variables. Le bloc **var** de votre procédure doit ressembler à cela :

```
var
  NbSec, NbPropose: integer;
  Reponse: string;
```

5 N'oublions pas d'initialiser 'Reponse', cela nous servira à entrer dans la boucle que nous allons construire. Initialisez donc la variable 'Reponse' à la chaîne vide :

```
Reponse := '';
```

6 La boucle s'arrètera lorsque l'utilisateur entrera « q ». Créez une constante contenant ce caractère sous le nom « RepQuit » :

```
const
  RepQuit = 'q';
```

7 Ecrivons maintenant le squelette de la boucle. La condition booléenne est des plus simples.

```
while Reponse <> RepQuit do
  begin

  end;
```

8 Nous allons maintenant réfléchir en terme d'itération : ce que nous allons écrire à l'intérieur de la boucle sera probablement exécuté plusieurs fois de suite. Il faudra faire en sorte que tout soit réglé à la perfection et que l'utilisateur ne voit rien de ce que nous ferons. La première chose consiste donc à lui demander une proposition. Pour cela, un appel de 'InputQuery' est nécessaire :

```
InputQuery('Proposition d''un nombre', 'Veuillez indiquer une proposition (''q'' pour arrêter)',
 Reponse);
```

9 Il va maintenant falloir faire quelque chose de cette réponse : on ne sait pas si l'utilisateur désire quitter ('q'), ou a proposé un nombre. Pour simplifier, nous admettrons que toute réponse différente de 'q' est une proposition de nombre. Mais revenons à notre problème : deux cas sont possibles, il va donc falloir construire un bloc **if** et ceci à l'intérieur du bloc **while** déjà entamé (rappelez-vous bien que l'art de bien programmer passe par l'art de bien combiner les différentes structures élémentaires dont vous disposez). Voici le squelette du bloc if à écrire :

```
if Reponse = RepQuit then
  begin

  end
else
  begin

  end;
```

10 Occupons-nous du cas où l'utilisateur a entré 'q' : un message l'informant qu'il a perdu, en lui donnant la valeur du nombre secret, est ce qui paraît le plus adapté. Entrez donc dans le premier bloc d'instruction (celui qui suit juste le **then**) l'instruction :

```
ShowMessage('Vous avez perdu, le nombre secret était ' + IntToStr(NbSec));
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il est inutile d'en faire plus : nous n'écrirons plus rien après le bloc **if**, et à la prochaine itération, la condition (Reponse <> RepQuit) n'étant plus vérifiée, la boucle se terminera, terminant également la procédure puisque nous n'écrirons rien après la boucle **while**. Important : Vous retrouverez souvent cette technique qui consiste à isoler quelques instructions comme les initialisations au début d'une procédure, puis à lancer une boucle ou une condition, et ainsi de suite : ceci permet en quelque sorte de « protéger » des instructions de plusieurs cas nuisibles. Ici, nous éliminons le cas où l'utilisateur désire quitter et nous allons maintenant nous concentrer sur le cas où il a fait une proposition.

11 Il nous reste à écrire un bloc d'instructions : celui qui suit le **else**. Dans ce cas, on a une proposition de l'utilisateur, à laquelle on doit répondre. Commencez par écrire l'instruction qui convertira la réponse en entier :

```
NbPropose := StrToInt(Reponse);
```

Pour vous donner un point de repère, voici le code source de la procédure tel qu'il devrait être écrit à la fin de cette étape :

```
procedure TForm1.Button1Click(Sender: TObject);
const
  RepQuit = 'q';
var
  NbSec, NbPropose: integer;
  Reponse: string;
begin
  NbSec := Trunc(Random(101));
  Reponse := '';
  while Reponse <> RepQuit do
    begin
      InputQuery('Proposition d''un nombre',
        'Veuillez indiquer une proposition (''q'' pour arrèter)', Reponse);
      if Reponse = RepQuit then
        begin
          ShowMessage('Vous avez perdu, le nombre secret était ' +
            IntToStr(NbSec));
        end
      else
        begin
          NbPropose := StrToInt(Reponse);
          { comparaison au nombre secret }
        end; {if}
    end; {while}
end;
```

Prenez au début cette sage habitude qui consiste à décrire en commentaires chaque **end** de fin de bloc, ou vous finirez rapidement comme moi au début par vous perdre dans vos instructions lorsqu'il faudra fermer 5 ou 6 **end** à la suite !

12 Voilà, nous pouvons maintenant comparer cette proposition et le nombre secret. Mais diable, il va encore falloir distinguer plusieurs cas !!! (Ne vous affolez surtout pas, j'ai un peu fait exprès d'imbriquer plusieurs blocs pour vous montrer les complications que cela peut rapidement apporter. Pour vous aider, rappelez-vous bien qu'à l'intérieur d'un bloc, vous n'avez absolument pas à vous soucier de l'extérieur : concentrez-vous sur l'écriture du bloc en lui-même avant de l'intègrer au programme)
L'instruction qui va suivre la conversion de 'Reponse' en entier est donc un bloc **if**. Ce bloc, il va falloir le construire, mais là, nous avons de l'avance puisque nous avons déjà écrit quelque chose de similaire. Le bloc en question va être écrit là ou est placé le commentaire { comparaison au nombre secret }. Voici le bloc qu'il vous faut écrire :

```
if (NbPropose < 0) or (NbPropose > 100) then
  ShowMessage('Le nombre secret est compris entre 0 et 100')
else if NbPropose < NbSec then
  ShowMessage('Le nombre secret est supérieur à ' + IntToStr(NbPropose))
else if NbPropose > NbSec then
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        ShowMessage('Le nombre secret est inférieur à ' + IntToStr(NbPropose))
      else
        begin
          ShowMessage('bravo, le nombre secret était ' + IntToStr(NbPropose));
          Reponse := RepQuit;
        end;
```

Voici les explications de ce bloc : le premier cas traite les valeurs non acceptées : en dessous de 0 ou au dessus de 100. Les deuxième et troisième cas traitent les cas où le nombre secret est différent du nombre proposé, mais où ce dernier est entre 0 et 100. Le dernier cas s'occupe du cas où le nombre secret est trouvé. Vous vous interrogez peut-être sur l'instruction 'Reponse := RepQuit'. vous avez raison de vous poser la question, mais réfléchissez bien : après avoir informé l'utilisateur qu'il a gagné, il faut arrèter la boucle. On pourrait utiliser un moyen que vous ne connaissez pas encore, à savoir 'break', mais il est plus commode de s'assurer que la prochaine itération n'aura pas lieu en donnant à 'Reponse' la valeur qui termine justement la boucle.

13. C'est terminé ! Si vous avez perdu le fil, **téléchargez le projet** terminé, et reprenez la manipulation à loisir. J'ai choisi volontairement un exercice difficile pour vous, l'objectif n'étant pas que vous soyez capable de reproduire ce que nous venons de faire, mais que vous compreniez bien l'intérêt qu'il y a à imbriquer les blocs les uns dans les autres. Remarquez au passage que la procédure que nous venons d'écrire est d'une longueur remarquable. C'est promis, plus d'exercices aussi tordus avant un petit moment.

## VIII-B-3 - Blocs 'repeat'

Les blocs **repeat** sont une variante des blocs **while**. Certains connaisseurs vont peut-être bondir en lisant cette phrase, mais je maintiens que le principe de base est le même, mais pour des emplois différents. Alors qu'une boucle **while** permet de répèter un bloc d'instruction tant qu'une condition est satisfaite, un bloc **repeat** permet de répèter un bloc d'instructions tant qu'une condition (toujours booléenne) n'est pas remplie : cette condition sera nommée « condition de sortie » de la boucle. Une autre différence est que le bloc **while** peut ne pas exécuter les instructions contenues dans le bloc, alors qu'un bloc **repeat** exécutera toujours au moins une fois les instructions contenues dans le bloc. La condition de sortie est testée après chaque itération, alors que dans un bloc **while**, c'est avant chaque itération.
Pour clarifier les choses, et avant même de connaître la syntaxe des blocs **repeat**, voici un tableau qui regroupe les différences des deux structures de bloc **while** et **repeat** :

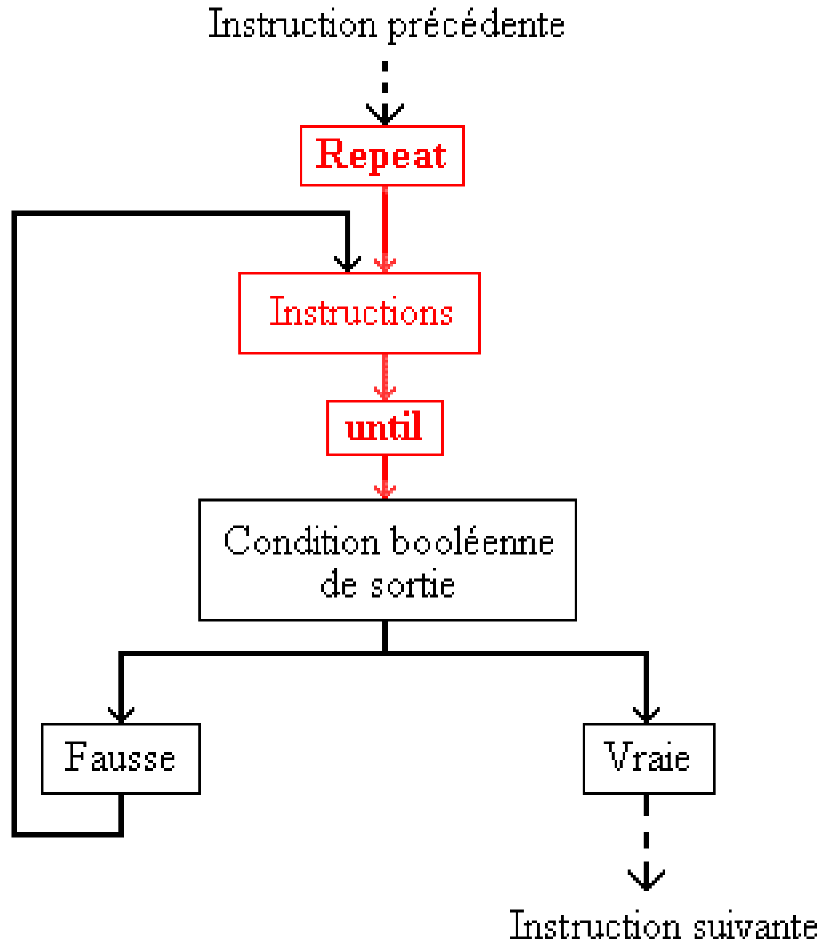
| Type de bloc : | Sortie du bloc : | Nombre minimum d'itérations : |
|---|---|---|
| **while** | condition fausse | 0 |
| **repeat** | condition vraie | 1 |

Voici maintenant la structure de ce bloc :

```
repeat
  instruction_1;
  ...
  instruction_n;
until condition_de_sortie;
```

Le bloc commence par le mot Pascal réservé **repeat** (qui a donné son nom au bloc) et se termine par le mot **until** suivi d'une condition booléenne de sortie. Entre ces deux mots réservés, peuvent être écrites autant d'instructions que l'on veut : c'est une particularité de ce bloc, qui est à lui seul un bloc d'instructions (le **begin** et le **end** sont inutiles (donc interdits) et remplacés par les mots **repeat** et **until**. Vous comprendrez mieux cela en regardant les exemples donnés ci-dessous.
Revenons un instant sur la condition booléenne. Cette condition est la condition qui doit être respectée pour qu'une autre itération soit lancée. Comprenez bien que la première itération s'effectuera toujours, puis ensuite seulement la condition de sortie est vérifiée. Si cette condition est vraie, la boucle se termine, si elle est fausse, une autre itération est lancée avec vérification de la condition en fin d'itération, et ainsi de suite.
Voici également, comme pour le bloc **while**, le schémas de fonctionnement d'un bloc **repeat** :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Examinez ce schémas à loisir et comparez-le à celui des boucles **while** : vous remarquerez que les instructions sont exécutées avant le test de la condition booléenne dans le cas des boucles **repeat**. Remarquez également que ce n'est pas la même valeur pour cette condition qui décide de la sortie de la boucle.
Tout comme pour une boucle **while**, il faudra faire en sorte de ne pas créer une boucle infinie. Pour cela, faire en sorte que la condition de sortie soit réalisée est indispensable à un endroit donné dans la boucle (mais ce changement peut ne pas être explicite et dépendre d'un paramètre indépendant de vous, comme atteindre la fin d'un fichier par exemple.
Voici comme premier exemple le petit programme perroquet déjà construit avec une boucle **while** au paragraphe précédent : il a été réécrit ici en utilisant une boucle **repeat**, plus adaptée ici :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Reponse: string;
const
  RepQuit = 'q';
begin
  Reponse := '';
  repeat
    InputQuery('', 'quelque chose à dire ?', Reponse);
    if Reponse &lt;> RepQuit then
      ShowMessage(Reponse);
  until Reponse = RepQuit;
end;
```

Vous verrez si vous comparez les deux versions de la procédure que les différences sont assez mineures : on a changé de type de boucle, ce qui ne change en rien ni les déclarations ni l'initialisation de 'Reponse'. La condition 'Reponse <> RepQuit' est déplacée en fin de boucle pour satisfaire la syntaxe de la boucle **repeat**, mais c'est sa


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

négation logique que l'on prend, car on doit non plus donner une condition pour continuer, mais pour ne pas continuer, c'est-à-dire arrèter. La négation logique de 'Reponse <> RepQuit' est 'not (Reponse <> RepQuit)', ou plus simplement 'Reponse = RepQuit'. Les deux instructions présentes à l'intérieur de la boucle sont inchangées.
Pourquoi, me direz-vous, perdre mon temps à vous débiter un exemple déjà traité, alors que je pourrais vous en trouver un autre ? C'est tout simplement car du point de vue logique, la deuxième version (en utilisant une boucle **repeat**) est plus correcte que la première. En effet, le principe du programme est de répèter tout ce que dit l'utilisateur, et il faudra donc bien pour cela lui poser une première fois la question (l'appel à InputQuery) : du point de vue théorique, la boucle doit être exécutée au moins une fois : c'est la boucle **repeat** qui s'impose alors.
Dans vos programmes, il vous faudra également réfléchir de cette manière : une itération est-elle indispensable ou non ? Si la réponse est oui, il vous faudra utiliser une boucle **repeat**, et une boucle **while** dans l'autre cas. La condition booléenne ne pose pas de problème car la condition d'arrèt utilisée dans une boucle **repeat** est habituellement la négation de la condition qui apparaitrait dans une boucle **while** (condition de « non arrèt »).
Les exemples utilisant des boucles **while** n'étant pas faciles à trouver, d'autres viendront dans la suite du guide ou dans ce paragraphe si j'en trouve d'intéresants.

## VIII-B-4 - Contrôle avancé des boucles

Lorsqu'on programme en utilisant des boucles, il se présente des situations où l'on aurait besoin de sauter la fin d'une itératon, de sortir immédiatement de la boucle en cours, ou même de terminer la procédure ou la fonction en cours d'exécution. Dans ces trois situations, on a besoin de « casser » le rythme normal de l'exécution du programme. Pour chacun de ces besoins, le langage Pascal a prévu une instruction spécifique. Les trois instructions correspondantes seront :

- Continue;
- Break;
- Exit;

Les deux premières instructions (Continue et Break) ne doivent être présentes qu'à l'intérieur d'une boucle : leur présence en dehors est interdite. La troisième peut être présente n'importe où à l'intérieur d'une procédure ou d'une fonction. Il est à noter que ces trois éléments sont des appels de procédures très spéciales sans paramètres et non pas des mots réservés Pascal.
L'instruction 'Continue;', lorsqu'elle est exécutée, termine immédiatement l'itération qui vient d'être entamée, et passe à la suivante : comprenez bien qu'elle ne termine pas la boucle (pour cela, utilisez 'Break;'). Cette instruction équivaut à la rencontre de la fin du bloc d'instructions pour les boucles **for** et **while**, et à la rencontre du mot **until** pour les boucles **repeat**.
Voici un exemple simple, dont le code source sera probablement exploré plus tard dans le guide (car encore un peu trop difficile actuellement) : quelles lettres de lecteurs sont disponibles sur votre ordinateur ? Si vous avez connu Windows 3.1, vous vous rappelez certainement qu'on devait choisir dans une liste déroulante ne présentant que les lecteurs valides. Cette liste, que vous pourrez utiliser avec Delphi (le composant s'appelle 'DriveComboBox') est construite au moyen d'une boucle **for**. Par un moyen que vous n'avez pas encore à connaître, la liste des lettres disponibles est obtenue (c'est Windows qui donne ces informations). Pour chaque lettre, une vérification est faite : si elle est disponible, elle est ajoutée à la liste, sinon, l'itération est stoppée au moyen de 'Continue'.
L'instruction 'Break;', quant à elle, termine complètement la boucle dans laquelle elle est présente. Souvenez-vous de la manipulation ci-dessus : on a à un moment donné fixé la valeur de la variable 'Reponse' à 'RepQuit' pour s'assurer de quitter la boucle. Imaginons que la boucle s'allonge d'une centaine de lignes (ca arrive parfois...), que pour une raison X ou Y, on vienne à modifier la condition, et pas l'instruction qui est censée provoquer la sortie de boucle : panique garantie ! La solution consiste à remplacer cette affectation de fortune par l'instruction 'Break;', qui terminera immédiatement la boucle et passera à la suite.
L'instruction 'Exit;', enfin, constitue le siège éjectable d'une procédure ou fonction. Cette manière de quitter une procédure est décriée par de nombreuses personnes, pretextant qu'il ne s'agit là que d'une manière commode de pallier à un algorithme défectueux. N'écoutez ces récriminations que d'une seule oreille, car vous auriez bien tort de vous priver d'utiliser 'Exit;' (après tout, le code source livré avec Delphi comporte de nombreux appels à 'Exit;'). 'Exit;' permet en fait d'éliminer rapidement des cas indésirables dés le début d'une procédure ou d'une fonction. Encore une fois, il va falloir patentier un peu pour les exemples, mais je peux vous assurer qu'ils viendront (Mini-projet à la fin du chapitre 8 par exemple).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Plus généralement, l'ensemble de ce chapitre constitue une base pour tout ce qui va suivre. Les structures et les instructions vues ici seront abondamment employées dans vos programmes, et vous apprendrez à les maîtriser au fur et à mesure de leur utilisation. En guide de conclusion de chapitre, il est important pour vous connaître, mais pas forcément de bien maîtriser chacune des structures vues dans ce chapitre. Vous aurez tout le temps de découvrir l'utilité de chacune de ces structures quand le besoin apparaîtra.
A ce stade du guide, vous commencez à avoir une connaissance appréciable du langage Pascal, même si quelques domaines sont encore inexplorés, comme les pointeurs ou les objets. Ce chapitre-ci est terminé (son écriture m'aura pris trois semaines, pour un résultat à mon sens assez moyen).
Avez-vous remarqué que nous sommes pour l'instant réduits, sous Delphi, à n'utiliser que bien peu de choses ? C'était nécessaire tant que vous ne connaissiez pas assez le langage Pascal, ce qui n'est plus le cas. Le prochain chapitre sera consacré bien plus à Delphi qu'au langage, même si des notions très importantes de ce dernier y seront vues. L'objectif en sera ambitieux : la connaissance de quelques composants, de leur utilisation, de leurs propriétés et de leurs évènements.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# IX - Manipulation des composants

Votre connaissance du langage Pascal augmentant, il est désormais possible de nous attaquer à un morceau de choix : l'utilisation des composants. Par composant, on entend ici en fait les fiches, qui sont (techniquement seulement) des composants, ainsi que tous les composants que vous pourrez être amenés à manipuler.
Ce chapitre sera encore un peu théorique et approfondira des notions déjà survolées auparavant. Vous apprendrez ainsi plus en détail ce que sont les composants, comme on les utilise, comment on accède à leurs propriétés, ce que sont les événements et comment on les utilise. Quelques exemples tenteront de rompre la monotonie de ce genre de cours. Un mini-projet est proposé vers la fin du chapître.

## IX-A - Introduction

Nous avons déjà parlé dés le début du guide de l'interface de vos applications. Ces interfaces seront constituées d'un ensemble bien structuré d'éléments pour la plupart visibles : boutons, cases à cocher, textes, menus... mais aussi d'autres non visibles sur lesquels nous reviendront. Chacun de ces éléments individuels se nomme « composant ». Un composant est en fait une sorte de boite noire qui a plusieurs têtes. La partie la plus facile à saisir d'un composant est assurément son coté visuel, mais ce n'est pas le seul aspect d'un composant : il contient tout un tas d'autres choses telles que des propriétés, des événements et des méthodes. Les propriétés sont des données de types très divers, dont certaines sont des constantes et d'autres des variables. Les événements sont tout autre : ils permettent d'exécuter une procédure dans certaines circonstances en réponse à un evènement concernant le composant (ces circonstances dépendent de l'evènement), comme par exemple lors du clic sur un bouton (vous devez commencer à vous familiariser avec celui-là, non ?). Les méthodes sont tout simplement des procédures et des fonctions internes au composant.
Au niveau du code source, chaque composant est une variable. Cette variable est de type objet. Vous n'avez nul besoin pour l'instant de savoir ce qu'est et comment fonctionne un objet, car ceci fera l'objet d'un long chapitre. Ce qu'il est intéressant pour vous de savoir pour l'instant, c'est qu'un objet ressemble beaucoup (en fait, pas du tout, mais on fera semblant de le croire, hein ?) à un enregistrement (type **record**) amélioré. Un objet peut contenir des données, comme les **record**, mais aussi des procédures, des fonctions, et encore certaines autres choses spécifiques aux objets dont nous parlerons plus tard et qui permettent entre auters de faire fonctionner les événements.
Au risque d'avoir l'air de changer de sujet (ce qui n'est pas du tout le cas), avez-vous parfois regardé cette partie du code source de l'unité 'Principale.pas' (fiche vierge créée avec une nouvelle application) située dans l'interface ? Jetez-y un coup d'oeil, vous devriez voir quelque chose du genre :

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Cet extrait de code, malgré son apparente complexité, fait bien peu de choses : il déclare un nouveau **type** (bloc type) nommé 'TForm1' et une variable nommée 'Form1' qui est justement de type 'TForm1'. La déclaration du type 'TForm1' devrait vous rappeler celle d'un type enregistrement (**record**), mis à part qu'ici on utilise le mot **class** suivi d'autre chose entre parenthèses. Une autre différence est que des mots réservés (**private** (« privé » en anglais) et **public**) sont présents à l'intérieur de cette déclaration, ne vous en occupez surtout pas pour l'instant.
La variable nommée 'Form1' est la fiche. Comprenez par là que cette variable contient tout ce qui a trait à la fiche : sa partie visible que vous pouvez voir en lancant l'application aussi bien que sa partie invisible, à savoir ses propriétés et ses événements. Le type de cette variable 'Form1' est 'TForm1'. Ce type est défini dans l'unité car il est spécifiquement créé pour cette fiche. Tout ajout de composant à la fiche viendra modifier ce type et donc modifiera la variable 'Form1' (mais pas sa déclaration).
Essayez par exemple de poser un bouton sur une fiche vierge. L'extrait de code devient alors :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

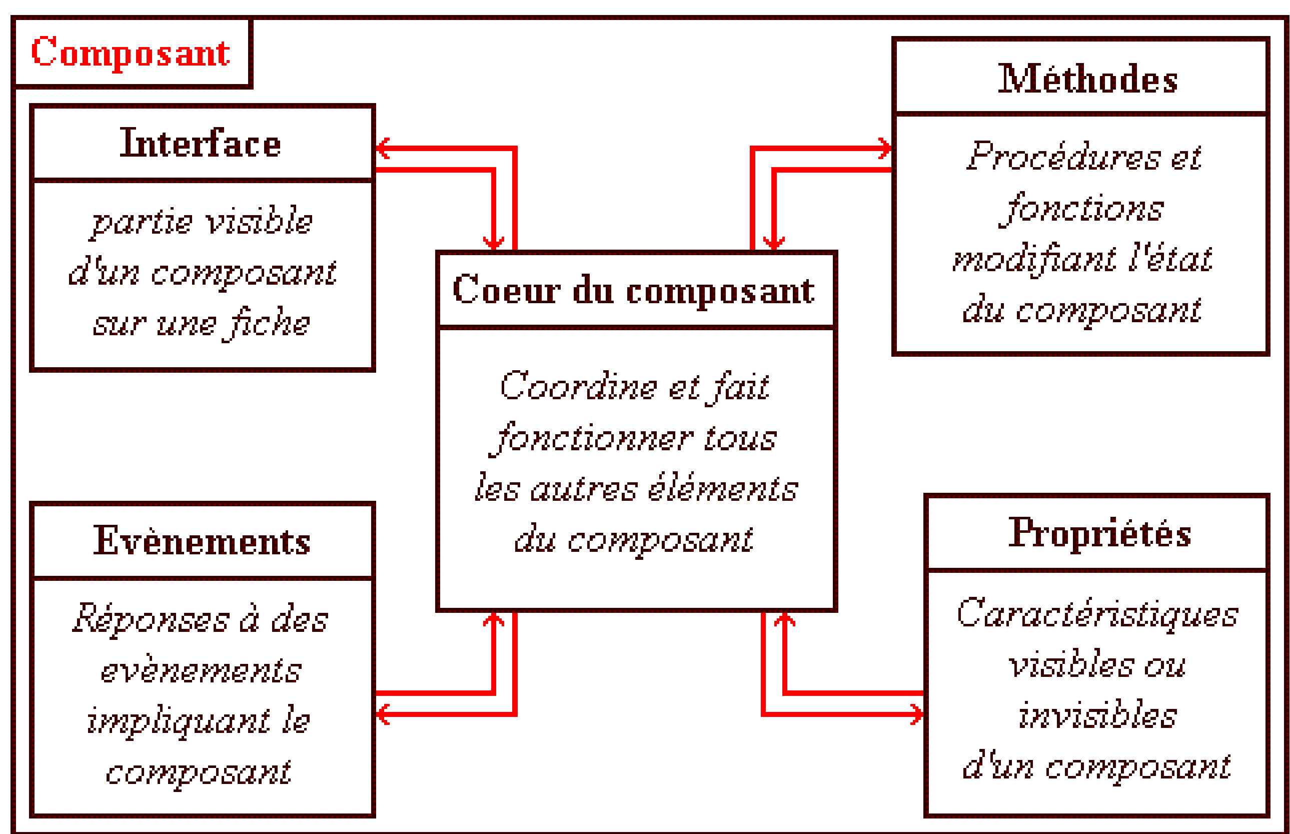
Le seul changement par rapport au premier code source est la ligne :

```
Button1: TButton;
```

Cette ligne anodine déclare en fait une variable « interne » au type 'TForm1' (au même sens que dans un enregistrement). La variable est nommée 'Button1' et est de type 'TButton'. Cette variable est le bouton, c'est-à-dire bien plus que ce que vous voyez à l'écran quand vous lancez l'application. Tout comme la variable 'Form1', la variable 'Button1' contient bien d'autres choses que sa partie visible : ce qui fait fonctionner les composants.

## IX-B - Aperçu de la structure interne d'un composant

Il est ici hors de question de rentrer déjà dans les détails de la structure d'un composant, c'est quelque chose que nous verrons plus tard. Il est cependant intéressant pour vous de connaître au moins la constitution théorique d'un composant : c'est quelque chose qui vous aidera mieux, j'en suis certain, à maîtriser ces « boites noires ».
Voici un schémas qui donne une représentation d'un composant :




Vous pouvez voir sur ce schémas qu'un composant a plusieurs aspects regroupés autour d'un coeur central.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Un premier aspect, souvent le plus connu, est l'interface (le coté visible) du composant. C'est ainsi que lorsque vous placez un composant 'Button' sur une fiche, vous voyez un bouton apparaître. Ce bouton n'est après tout qu'une image qui est la partie visible d'un très très gros iceberg.
- D'un autre coté, on voit les propriétés d'un composant. Ces éléments, dont nous avons déjà un peu parlé sont pour une partie présentes dans l'inspecteur d'objets lorsque le composant est sélectionné, et pour l'autre partie utilisables depuis le code source uniquement (nous verrons comment et pourquoi en temps et en heure).
- Un troisième aspect d'un composant est ses événements. Ces événements donnent simplement la *possibilité* (et non l'obligation comme certains le croient au début) de réagir en réponse à une intervention sur le composant : clic, double clic, mouvement de souris, appui d'une touche, activation, etc... autant de procédures qui pourront être créées et reliées automatiquement au composant par Delphi. Ces procédures seront exécutées à chaque fois que l'evènement se produira.
- Le dernier aspect d'un composant est constitué par ses méthodes. On emploie le terme de *méthode* pour désigner les procédures et fonctions internes à un composant (et plus généralement à tout objet). Chacune de ces méthodes a un rôle à jouer dans le composant, et dans lui seul. Les méthodes sont les petits bras articulés qui vont faire le travail à l'intérieur du composant.

Afin d'illustrer tout cela, je vais reprendre un exemple trouvé dans l'aide en ligne d'un autre langage (Visual Basic pour ne pas le citer...) : celui d'un ballon. Notre ballon est un objet, et dans notre cas un composant (c'est plus qu'un simple objet). Reprenons maintenant les 4 aspects d'un composant en pensant à notre ballon.

- L'interface, ici, sera évidemment un morceau de caoutchouc. Mais n'importe quel bout de caoutchouc n'est pas un ballon, il a une forme, une composition, certaines particularités physiques qui font un ballon d'un morceau de caoutchouc. Nous allons pouvoir modifier ce ballon « virtuel » à volonté.
- C'est ici qu'interviennent les propriétés : couleur du ballon, contenu du ballon (air, hélium, eau, ...), gonflé ou dégonflé, etc... Ces caractéristiques vont pouvoir être règlées pour nous permettre d'avoir le ballon dont nous aurons besoin : ballon rouge gonflé à l'hélium par exemple.
- Comme evènement, je prendrai la rencontre fortuite de notre ballon et d'une épingle : c'est un evènement. Dans la réalité, notre ballon n'aurait pas d'autre choix que d'éclater : se dégonfler en émettant un bruit d'explosion. Eh bien, pour notre ballon virtuel, c'est la même chose : il va nous falloir changer l'état de notre ballon (changer la valeur de la propriété « gonflé » et emettre un son). Prenons un exemple moins barbare : la personne qui tient le ballon le lâche. Plusieurs possibilités sont à envisager alors : si le ballon est gonflé d'hélium, il va s'envoler, si il est vide, il va tomber. Nous devrons gèrer plusieurs scénarios et donc faire une analyse de l'état de notre ballon avant de le modifier.
- Les méthodes, enfin, sont tout ce qui va nous permettre de modifier notre ballon : le faire monter, le faire tomber, lui faire emettre un bruit d'explosion, se dégonfler, se gonfler (même si en réalité ce n'est pas physiquement très correct, personne ne sera présent dans votre programme pour gonfler les ballons !).

## IX-C - Manipulation des propriétés d'un composant

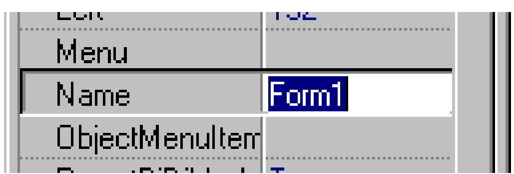
Cette partie vous explique les moyens de modifier les propriétés d'un composant. Ces moyens sont au nombre de deux. Delphi, tout d'abord, vous propose de faire ces modifications de façon visuelle. Le code Pascal Objet, d'autre part, vous permet de faire tout ce que fait Delphi, et bien d'autres choses.

### IX-C-1 - Utilisation de Delphi

Delphi est un fantastique outil de programmation visuelle, vous ais-je dit au début de ce guide. C'est le moment d'expliquer comment tirer partie de cela : Delphi vous propose divers outils pour modifier facilement les propriétés des composants.
Lorsque vous avez une fiche sous Delphi, vous pouvez la redimensionner et la déplacer à volonté : vous manipulez sans le savoir les quatre propriétés nommées 'Height', 'Width', 'Top' et 'Left' de cette fiche. Ces propriétés, vous pouvez également les modifier depuis l'inspecteur d'objets, qui a l'avantage de présenter d'autres propriétés non accessibles ailleurs. C'est la même chose lorsque vous placez un composant sur une fiche : vous pouvez le dimensionner à volonté sur la fiche comme dans un logiciel de dessin, mais vous pouvez aussi utiliser l'inspecteur d'objets pour modifier les mêmes propriétés que pour la fiche.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour déplacer ou dimensionner une fiche, il vous suffit de le faire comme n'importe quelle autre fenêtre sous Windows. Pour dimensionner un composant, il faut le sélectionner d'un simple clic, puis déplacer les poignées (petits carrés noirs tout autour du composant) avec la souris. Pour déplacer un composant, laissez faire votre intuition : prenez le composant à la souris en cliquant une fois dessus et en maintenant le bouton gauche de la souris, puis déplacez-le. Pour le poser, relâchez simplement le bouton de la souris. Un autre moyen consiste à sélectionner le composant, puis à maintenir enfoncée la touche 'Ctrl' du clavier tout en utilisant les fléches de direction, ce qui déplace la sélection d'un pixel dans l'une des 4 directions désignée.
Au niveau de la sélection, vous pouvez sélectionner plusieurs composants en en sélectionnant un premier, puis en sélectionnant les autres en maintenant la touche 'Shift' ('Maj' en français) du clavier. Vous pouvez également dessiner un rectangle avec la souris en cliquant et en maintenant enfoncé le bouton de la souris à un emplacement sans composant, puis en allant relâcher le bouton à l'angle opposé du rectangle : tous les composants ayant une partie dans ce rectangle seront sélectionnés. Pour annuler une sélection, appuyez sur 'Echap' jusqu'à ce que toutes les poignées de sélection aient disparu.
En ce qui concerne les tailles et positions des éléments, vous disposez également sous Delphi d'une palette d'alignements prédéfinis accessible par le menu « Voir », choix « Palette d'alignement ». Cette palette permet, en sélectionnant par exemple plusieurs boutons, de répartir également l'espace entre ces boutons, de centrer le groupe de boutons, et bien d'autres choses que vous essaierez par vous-même.
Pour les autres propriétés, il est nécessaire d'utiliser l'inspecteur d'objets. Vous pouvez directement modifier une propriété commune à plusieurs composants en les sélectionnant au préalable : l'inspecteur d'objets n'affichera alors que les propriétés communes à tous les composants sélectionnés.
Une propriété des plus importantes dont vous devez maintenant apprendre l'existence et le fonctionnement est la propriété 'Name' ('Nom' en français). Cette propriété est une propriété spéciale utilisable uniquement sous Delphi et qui décide du nom de la variable qui stockera le composant. Prenons par exemple la seule fiche d'un projet vide : sa propriété 'Name' devrait être « Form1 » (voir capture d'écran ci-dessous).




Vous vous souvenez certainement de l'extrait de code dont nous avons parlé dans l'introduction, cet exemple où était déclarée une variable 'Form1'. C'est la propriété 'Name' qui décide du nom de cette variable. Changez donc ce nom en un nom plus adéquat, tel « fmPrinc » ('fm' comme 'Form': habituez-vous dés à présent à rencontrer beaucoup de ces abréviations en deux lettres minuscules désigant le type de composant et donc différentes pour chaque type de composant. je vous conseille d'utiliser ces abréviations dés que vous les connaissez, ou de créer les vôtres et de vous y tenir absolument). Examinez ensuite le code source, le voici tel qu'il devrait être :

```
type
  TfmPrinc = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmPrinc: TfmPrinc;
```

Vous voyez que le nom de la variable à changé, mais aussi que le type utilisé a aussi changé. Le nom de ce type est en fait généré en fonction du nom que vous indiquez pour la propriété 'Name' : un T majuscule est simplement ajouté devant pour signifier que c'est un Type. habituez-vous à rencontrer ce T majuscule dans beaucoup de situations, et même si possible à l'utiliser vous-mêmes comme je le fais dans le guide et en dehors.
Placez maintenant trois boutons sur la fiche. L'extrait de code devrait maintenant être :

```
type
  TfmPrinc = class(TForm)
    Button1: TButton;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    Button2: TButton;
    Button3: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmPrinc: TfmPrinc;
```

Vous avez certainement remarqué que trois lignes ont fait leur apparition, déclarant chacune une variable « interne » au type 'TfmPrinc'. Chacune de ces variables est un des boutons que vous venez de placer sur la fiche. Le type employé pour chacune des trois variables est 'TButton', qui est le type de composant utilisé pour les boutons. Renommez ces trois boutons (toujours depuis l'inspecteur d'objets, jamais depuis le code source !) en les nommant à votre convenance, et en utilisant comme abréviation de deux lettres : 'bt'. Voici le code source qui vous indique les noms que j'ai personnellement utilisés.

```
type
  TfmPrinc = class(TForm)
    btBonjour: TButton;
    btSalut: TButton;
    btCoucou: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmPrinc: TfmPrinc;
```

Il n'est pas encore très important pour vous de connaître cette partie du code source, mais il est tout de même essentiel de savoir que cette propriété 'Name' désigne un nom de variable et doit donc contenir un identificateur valide, respectant la règle du préfixe de deux minuscules désignant le type de composant. Nous allons encore creuser un petit peu le sujet en créant une procédure répondant à un évènement (comme vous l'avez certainement déjà fait plusieurs fois) : double-cliquez sur l'un des trois boutons sous Delphi, une procédure est alors générée. N'écrivez rien dans cette procédure, ce n'est pas l'objet ici. Retournez plutôt voir le morceau de code déclarant le type 'TfmPrinc'. Vous devez voir une nouvelle ligne.

```
type
  TfmPrinc = class(TForm)
    btBonjour: TButton;
    btSalut: TButton;
    btCoucou: TButton;
    procedure btBonjourClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmPrinc: TfmPrinc;
```

Cette ligne déclare la procédure nommée 'TfmPrinc.btBonjourClick', mais en ne mettant qu'une partie du nom : 'btBonjourClick'. C'est en fait le vrai nom de la procédure. Jusqu'ici, je vous ai laissé croire (mais qui viendra s'en plaindre ?) que 'TForm1.btBonjourClick' était le nom de la procédure, ce qui est impossible puisque ce n'est pas un identificateur. 'btBonjourClick', en revanche, est un identificateur valide. Quid alors de 'TfmPrinc.' ? La réponse est assez simple : c'est le nom « qualifié » de la procédure. Ce nom comporte le nom du type auquel il appartient : 'TfmPrinc', suivi d'un point (.) pour indiquer qu'on va maintenant désigner un élément *intérieur* à ce type, à savoir ici


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

la procédure nommée 'btBonjourClick'. C'est pour cela que la déclaration à l'intérieur du type 'TfmPrinc' ne comporte pas ce préfixe ('TfmPrinc.') : on est en fait déjà à l'intérieur du type !
Un autre fait intéressant qu'il vous faut connaître est que dans cette partie du code source que nous venons d'étudier succintement, tous les composants, ainsi que toutes les procédures associées à des événements seront déclarées automatiquement par Delphi. La gestion de cette partie du code source incombe exclusivement à Delphi, vous n'aurez donc pas le droit d'y toucher (sous peine de faire des bétises parfois irréparables). Cela ne vous épargne pas de connaître l'existence et de comprendre ce qui s'y trouve. Nous aurons au cours du guide l'occasion de revenir sur cette partie.
Pour revenir sur un terrain plus stable, de nombreuses autres propriétés sont disponibles dans l'inspecteur d'objets. Leur nombre et leur diversité peut effrayer, mais c'est un gage de puissance et de flexibilité des composants. Pour l'instant, nous n'allons pas manipuler ces propriétés, puisque vous l'avez normalement déjà fait au fur et à mesure des autres chapitres du guide, mais si le coeur vous en dit, n'hésitez pas à vous lancer et à vous tromper : en cas d'erreur, fermez le projet sans enregistrer, puis créez-en un nouveau !

## IX-C-2 - Utilisation du code source

L'utilisation de l'inspecteur d'objets est certes intuitive, facile et sympathique, mais d'une efficacité très limitée. Voici les limites principales :

- Certaines propriétés sont inaccessibles depuis l'inspecteur d'objets.
- On ne peut pas modifier les propriétés pendant l'exécution de l'application.
- On ne peut manipuler que certains composants (nous verrons cela bien plus tard : certains composants, du fait de leur absence dans la palette des composants, ne peuvent être manipulés que depuis le code source de l'application).

De plus, il ne faut pas oublier, mais cela, vous ne le savez peut-être pas encore si vous êtes novice en programmation, que l'inspecteur d'objets est une facilité offerte aux programmeurs, mais qu'avant l'existence de Delphi, il fallait bien faire tout ce que fait maintenant l'inspecteur d'objet, et le faire depuis le code source : c'était parfois pénible, souvent fastidieux et source d'erreurs.
Il paraît donc inconcevable de se limiter à l'inspecteur d'objets pour la manipulation des propriétés : c'est très souvent dans le code Pascal d'une application que s'effectueront le plus grand nombre de manipulations de propriétés.
Une propriété se manipule, selon les cas, comme une variable ou une constante. En fait, une propriété est un élément spécial dont vous n'avez pour l'instant pas à connaître les détails, mais qui permet la lecture des données (on peut lire la valeur de la propriété, l'écriture de ces données (on peut modifier la valeur de cette propriété) ou les deux à la fois. La presque totalité des propriétés est soit en lecture seule soit en lecture/écriture.

- Les propriétés en écriture seule sont rarissimes, vous n'en rencontrerez probablement jamais, l'intérêt de ce genre de propriété reste à prouver.
- Les propriétés en lecture seule sont fréquentes : elles permettent de se renseigner sur l'état d'un composant, sans pouvoir agir sur cet état (Dans l'aide de Delphi, ces propriétés sont souvent signalées par une petite flèche bleue). Ces propriétés se manipulent comme n'importe quelle constante.
- Les propriétés en lecture/écriture sont les plus courantes : elles permettent tout autant de s'informer que de modifier l'état du composant en modifiant la valeur de la propriété.

Pour utiliser une propriété d'un composant dans du code Pascal, il faut séparer deux cas, selon qu'on est ou pas dans une procédure interne à la fiche qui contient le composant.

- Si on est dans une procédure interne à la fiche qui contient ce composant (le composant et la procédure dans laquelle on souhaite manipuler une propriété de ce composant doivent être déclarés dans le même type), ce qui est le cas le plus fréquent, il faut écrire

  1. le nom du composant auquel elle appartient (le nom de la variable qui stocke ce composant : cette variable, dans notre cas, est interne au type qui est utilisé pour déclarer la variable qui stocke la fiche)
  2. un point (.)
  3. le nom de la propriété


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ainsi, pour désigner la propriété 'Caption' d'un bouton nommé (la valeur de la propriété 'name' est le nom du composant) 'btBonjour', on devra écrire :

```
btBonjour.Caption
```

On pourra utiliser cette expression comme une variable puisque la propriété 'Caption' est en lecture/écriture pour les boutons (composants de type 'TButton').

- Dans les autres cas, c'est-à-dire dans les procédures non internes à la fiche, mais présentes dans la même unité, ou dans les procédures d'autres unités, il faudra en écrire un peu plus. C'est ici une notion des plus importantes : depuis l' « intérieur » de la fiche (c'est-à-dire dans une procédure interne à cette fiche), on peut utiliser tout ce que contient la fiche. Lorsqu'on est à l'extérieur de la fiche, il va d'abord falloir y entrer, puis ce sera le même procédé que ci-dessus. Voici ce qu'il faudra écrire :

1. le nom de la fiche, c'est à dire le nom de la variable qui contient la fiche (c'est également le contenu de la propriété 'Name' de la fiche)
2. un point (.)
3. le nom du composant auquel elle appartient (le nom de la variable qui stocke ce composant : cette variable, dans notre cas, est interne au type qui est utilisé pour déclarer la variable qui stocke la fiche)
4. un point (.)
5. le nom de la propriété

Ainsi, depuis une autre fiche par exemple, pour désigner la propriété 'Caption' d'un bouton nommé 'btBonjour' d'une fiche nommée 'fmPrinc', on devra écrire :

```
fmPrinc.btBonjour.Caption
```

Pour résumer ces divers cas qui peuvent paraître compliqués au premier abord, il faut simplement s'assurer, lorsqu'on veut utiliser un élément, que ce soit un composant ou une propriété, qu'on y a accès directement, c'est-à-dire sans accèder à une (autre) fiche. Beaucoup de choses dans le style de programmation employé sous Delphi sont conçues sur ce principe : pour accèder à une information écrite sur un papier, on doit d'abord trouver ce papier en ouvrant le classeur qui le contient, et aussi parfois trouver le classeur dans une armoire, trouver le bureau qui contient l'armoire... ces dernières étapes étant bien entendues inutiles si on a déjà le classeur ouvert devant soi lorsqu'on a besoin de l'information sur la feuille de papier !
Avant de passer aux incontournables exemples (je sens que certains d'entre vous en auront absolument besoin), je tiens à revenir sur ce point qui apparaît pour séparer les noms de fiches, de composants et de propriétés. Le point, en Pascal Objet, est un opérateur qui permet d'accèder au contenu de ce qui le précède : lorsqu'on écrit 'fmPrinc.', on va s'adresser ensuite à quelque chose d'interne à fmPrinc. De même, lorsqu'on écrit 'btBonjour.', on s'adresse ensuite à quelque chose d'interne au bouton. Lorsqu'on est à l'extérieur de la fiche 'fmPrinc', on doit écrire 'fmPrinc.btBonjour.' pour s'adresser à quelque chose d'interne au bouton. Pour accèder à la propriété 'Caption' de la fiche, il suffira d'écrire alors 'fmPrinc.Caption'. Lorsqu'on sera en plus à l' « intérieur » de cette fiche, 'Caption' désignera directement la propriété 'Caption' de la fiche.
Pour revenir à notre exemple de la feuille de papier et du classeur, imaginons tout d'abord que vous ayez le classeur devant vous : vous indiquerez « page824.information75 » pour accèder à l'information sur la page (en admettant évidemment que vous connaissiez le nom du papier et le nom de l'information, ce qui revient à savoir quelle information vous cherchez et sur quel papier vous comptez la trouver. Imaginons maintenant que vous soyez dans le mauvais bureau : vous devrez alors écrire tout cela : « mon_bureau.armoire_archives.classeur_clients.page824.information75 ». C'est, je vous l'accorde, parfois un peu pénible, mais c'est la clé d'une bonne organisation : vous saurez toujours comment trouver l'information désirée sous Delphi à partir du moment où vous saurez où elle est.
Venons-en à un exemple très pratique : créez un projet vierge et placez deux boutons sur la feuille principale. Nommez la feuille « fmPrinc » et les deux boutons : « btTest1 » et « btTest2 » (utilisez la propriété 'name' en sélectionnant la fiche, puis chaque bouton). Donnez une tête présentable à tout cela en modifiant également les propriété 'Caption', 'Width', 'Height', 'Top' et 'Left' des boutons et de la fiche (selon votre humeur du moment). Double-cliquez sur le bouton 'btTest1', ce qui va générer une procédure locale à la fiche 'fmPrinc', nommée 'btTest1Click' (rien de très



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

nouveau jusqu'à maintenant, mis à part que vous pouvez certainement remarquer que le nom de cette procédure ne doit rien au hasard, puisqu'il est composé du nom du composant et de 'Click'). Faites de même pour le second bouton ('btTest2').
Nous allons modifier les propriétés 'Caption' des deux boutons et de la fiche depuis ces procédures, ce qui signifie qu'un clic sur l'un des boutons changera les textes affichés sur les boutons ou dans le titre de la fenêtre. Utilisez l'extrait de code ci-dessous pour compléter vos procédures :

```
procedure TfmPrinc.btTest1Click(Sender: TObject);
begin
  Caption := 'Titre de fenêtre';
end;

procedure TfmPrinc.btTest2Click(Sender: TObject);
begin
  btTest1.Caption := 'Titre de bouton 1';
  btTest2.Caption := 'Titre de bouton 2';
end;
```

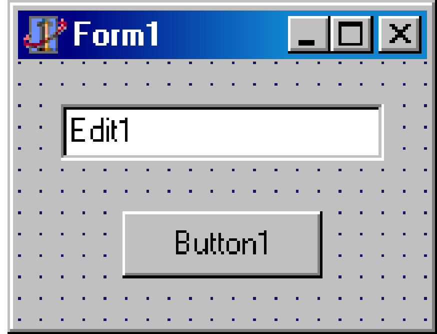
Exécutez l'application et cliquez sur chacun des boutons en prettant attention à l'effet produit : un clic sur l'un des boutons change le titre de la fenêtre, ce qui correspond à la propriété 'Caption' de la fiche. Un clic sur l'autre bouton, en revanche, modifie le texte affiché sur chacun de ces deux boutons : leur propriété 'Caption' est modifiée. Notez que cliquer plusieurs fois sur les boutons n'a pas d'effet visible puisque les valeurs des propriétés ne changent plus. Malgré cela, le code des procédures est bien exécuté.
Revenons maintenant à l'extrait de code ci-dessus : la ligne :

```
Caption := 'Titre de fenêtre';
```

modifie la propriété 'Caption' de la fiche. Comme la procédure fait partie de la fiche (deux moyens de s'en assurer : la procédure est déclarée à l'intérieur du type 'TfmPrinc' plus haut dans le code, et le nom de la procédure est précédé de 'TfmPrinc.'), nous n'avons rien à faire de plus pour accèder à l'intérieur de la fiche : on utilise directement 'Caption', et ceci comme une variable, à laquelle on affecte une chaîne de caractères. L'exécution de cette petite ligne va suffir pour changer le titre de la fenêtre en ce que nous voudrons. Le code de la deuxième procédure, comme vous l'avez vu en exécutant l'application, modifie le texte des deux boutons. Pour cela, on a utilisé deux instructions similaires, une pour chaque bouton. Pour modifier la propriété 'Caption' du bouton 'btTest1', on doit d'abord s'adresser au composant 'btTest1', placé sur la fiche. On est sur la fiche, donc on s'adresse directement au composant, puis à la propriété :

```
btTest1.Caption := 'Titre de bouton 1';
```

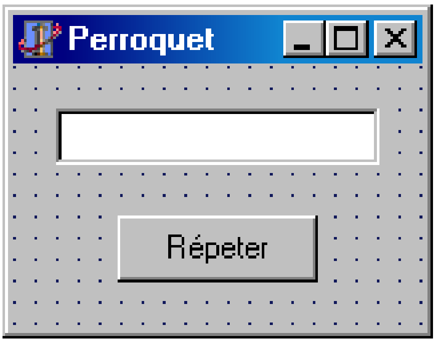
Passons maintenant à quelque chose de plus intéressant. Créez un nouveau projet, puis placez sur la seule fiche deux composants : un composant Edit (zone d'édition, l'icône est ) et un bouton classique. Placez-les pour obtenir quelque chose du style :





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Modifiez les propriétés Caption de la fiche et du bouton en mettant respectivement « Perroquet » et « Répeter ». Nommez la fiche 'fmPrinc', le bouton 'btRepeter' et la zone d'édition (on l'appelera désormais "l'Edit" pour faire plus court) 'edMessage'. La propriété 'Text' (de type string) d'un composant Edit (type TEdit) permet de lire et écrire le texte contenu (mais pas forcément entièrement affiché, faute de manque de place à l'écran) dans la zone d'édition. Utilisez l'inspecteur d'objets pour effacer le texte indésirable qui doit s'y trouver : « edMessage » (nous expliquerons plus tard pourquoi ce texte a changé quand vous avez modifié la propriété 'Name').
Votre interface doit maintenant être quelque chose du style :




Evitez maintenant de visualiser dans votre navigateur le code source présent ci-dessous, réalisez l'exercice et regardez ensuite la solution pour éventuellement vous corriger. Le principe du programme simple que nous allons créer va être de répeter ce qui est écrit dans la zone d'édition lors d'un clic sur le bouton 'Répeter'. Générez la procédure répondant au clic sur le bouton (double-clic sur celui-ci). La première instruction va consister à attribuer à une variable chaîne le texte écrit dans la zone d'édition. Déclarez une variable 'Msg' de type Chaîne. Tapez maintenant (en tant que première instruction) l'instruction qui affecte à la variable 'Str' la valeur de la propriété 'Text' de la zone d'édition (nommée 'edMessage'). La deuxième instruction, plus simple consiste à afficher ce message par un moyen habituel déjà souvent utilisé.
Vous avez terminé ? Voici la solution :

```
procedure TfmMain.btRepeterClick(Sender: TObject);
var
  Msg: string;
begin
  Msg := edMessage.Text;
  ShowMessage(Msg);
end;
```

La seule difficulté est dans la première ligne : la propriété 'Text' du composant 'edMessage' s'obtient par 'edMessage.Text', car la procédure est interne à la fiche, et peut donc s'adresser directement aux composants. Notez que l'on aurait pu faire plus court en se passant de la variable Msg et en écrivant l'instruction unique :

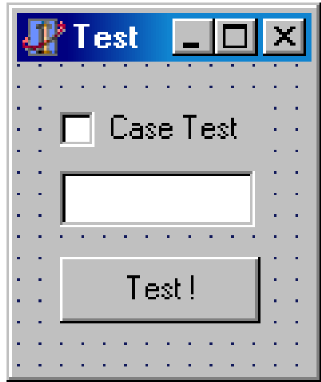
```
ShowMessage(edMessage.Text);
```

car edMessage.Text est bien de type **string** et peut donc être transmis en tant que paramètre à 'ShowMessage'.
Nous avons fait la manipulation dans les deux sens : écriture (exemple avec 'Caption') et lecture (à l'instant). Dans un programme Pascal Objet sous Delphi, nous manipulerons les propriétés à longueur de temps. Il est donc primordial de vous entraîner un peu. Voici un autre exemple plus complexe.

Créez un projet vierge, placez un composant CheckBox (Case à cocher, son icône est ), une zone d'édition, et un bouton. Le but de cet exercice est de contrôler l'état de la case à cocher avec la zone d'édition et le bouton. Avant cela, il vous faut une propriété de plus : la propriété 'Checked' de type Booléen des cases à cocher (type TCheckBox) permet de lire ou fixer l'état de la case (cochée ou non cochée). Lorsque 'Checked' vaut 'true', la case apparaît cochée, et non cochée lorsque 'Checked' vaut 'false'. En outre, la propriété 'Caption' décide du texte présent à coté de la case. Ceci permet de (dé)cocher la case en cliquant sur ce texte (fonctionnement classique de Windows, rien de nouveau ici).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Nommez la case à cocher 'cbTest', le bouton 'btModif' et la zone d'édition 'edEtat', la fiche principale (et unique), comme presque toujours, sera nommée 'fmPrinc'. Donnez des textes acceptables à tout cela pour obtenir l'interface ci-dessous :




Générez la procédure de réponse au clic sur le bouton. Nous allons réaliser deux tests similaires : nous comparerons le texte écrit dans la zone d'édition à 'false' ou 'true', et s'il est égal à l'un des deux, la case à cocher sera mise à jour en conséquence. En outre, le texte de la zone d'édition sera effacé à chaque clic sur le bouton pour qu'on puisse y retaper facilement du texte.
Voici le code source :

```
procedure TfmPrinc.btModifClick(Sender: TObject);
begin
  if edEtat.Text = 'false' then
    cbTest.Checked := false;
  if edEtat.Text = 'true' then
    cbTest.Checked := true;
  edEtat.Text := '';
end;
```

Cet exemple montre bien à quel point on peut assimiler les propriétés à des variables (mais pas les confondre, attention, ce ne sont PAS des variables). 'edEtat.Text' par exemple est d'abord utilisé deux fois en lecture pour être comparé à une autre chaîne, puis en écriture en étant fixé à la chaîne vide. La propriété 'Checked' de 'cbTest' est modifiée comme le serait une variable booléenne, à ceci près qu'en plus, un effet visuel est produit : la coche apparait ou disparait.
D'innombrables exemples seront présents dans la suite du guide, puisque la manipulation des composants est un passage obligé sous Delphi. Pour l'instant passons à un sujet très lié et non moins intéressant : la manipulation des méthodes des composants.

## IX-D - Manipulation des méthodes d'un composant

Il n'est pas ici question, comme pour les propriétés, d'utiliser l'interface de Delphi : tout ce qui touche aux méthodes touche à l'exécution des applications, c'est-à-dire que toutes les manipulations s'effectuent depuis le code source.
Nous avons déjà un peu parlé des méthodes, en disant qu'il s'agissait de procédures ou fonctions internes aux composants. Ces méthodes font tout le travail dans un composant. Même la manipulation des propriétés cache en fait la manipulation des méthodes, mais nous reparlerons de ca plus tard. Pour l'instant, il s'agit de savoir utiliser ces méthodes. Le principe est des plus simples pour utiliser une méthode : on s'en sert comme les propriétés, en s'assurant simplement qu'on y a accès en la précédant par le nom du composant auquel elle appartient. Si ces méthodes ont des paramètres, on les donne comme pour des procédures ou des fonctions.
Un petit exemple amusant pour commencer : créez un projet vierge et placez deux boutons 'Button1' et 'Button2' sur la fiche principale (et toujours unique, mais cela ne va plus durer très longtemps). Dans la procédure réagissant au clic sur 'Button1', entrez l'instruction ci-dessous :

```
Button2.Hide;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

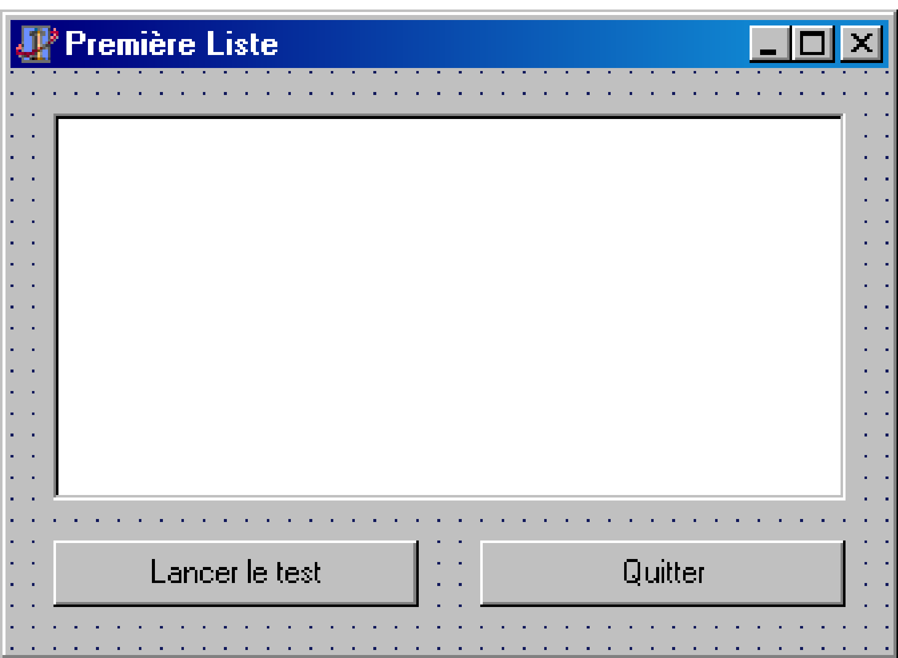
puis exécutez l'application. Lors d'un clic sur 'Button1', 'Button2' disparaît : il se « cache » ('Hide' en anglais). Vous remarquerez qu'aucune propriété n'a été modifiée par notre code puisqu'aucune affectation n'est écrite. Pourtant, la méthode 'Hide' des composants Button cache le bouton, et effectue donc en quelque sorte la même chose que si l'on avait changé la propriété 'Visible' du bouton (propriété qui décide si un composant est visible ou non).
Puisqu'on en est à l'amusement, mettez le code ci-dessous à la place du précédent :

```
Hide;
Sleep(5000);
Show;
```

En exécutant cet exemple, l'application « disparait » pendant 5 secondes (5000 millisecondes, d'où le 5000 dans le code) puis réapparait comme avant. L'explication est la suivante : la méthode 'Hide' nommée directement s'applique à la fiche : la fiche se cache donc. Puis la procédure 'Sleep' fait « dormir » l'application pendant 5 secondes. Enfin, après ces cinq secondes, la méthode 'Show' fait réapparaître la fiche.
Une méthode des fiches qui ne manquera pas de vous intéresser est la méthode 'Close'. Cette méthode ferme la fiche dont on appelle la méthode 'Close'. Il est alors intéressant de savoir que l'application se termine lorsque l'on ferme la fiche principale. Essayez donc, à la place du code précédent un peu fantaisiste, de mettre l'instruction unique :

```
Close;
```

Exécutez l'application. Un clic sur le bouton ferme la fiche, et donc quitte l'application : vous venez de créer votre premier bouton 'Quitter' !
Passons à un exemple plus croustillant : vous voyez régulièrement sous Windows des listes d'éléments. Un moyen de créer ces listes sous Delphi (on parle ici des listes simples, et non pas des listes complexes telles que la partie droite de l'explorateur Windows) passe par le composant ListBox (type TListBox, dont l'icône est ). Sur le projet déjà entamé, placez un composant ListBox, redimensionnez le tout pour obtenir ceci :




Commençons par le plus simple : assurez-vous que le bouton 'Quitter' quitte effectivement l'application (en générant la procédure répondant au clic et en écrivant comme unique instruction « close »). Le bouton 'Lancer le test' nous permettra d'effectuer divers tests de plus en plus élaborés.
Avant cela, un petit complément sur les ListBox s'impose. Les ListBox (zones de liste) sont composées d'un cadre, dans lequel apparaissent un certain nombre de lignes. Ces lignes comportent le plus souvent du texte. Un composant ListBox comporte une propriété 'Items' qui centralise tout ce qui a trait à la gestion des éléments de la liste. 'Items'


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

n'est pas une simple donnée : c'est un objet. Pour l'instant, nul besoin encore pour vous d'en connaître plus sur les objets, sachez simplement qu'un objet se manipule comme un composant : il possède propriétés et méthodes. Cette propriété 'Items' permet donc l'accès à d'autres propriétés et méthodes internes. La méthode 'Add' de 'Items' permet d'ajouter une chaîne de caractères à une zone de liste. Elle admet un unique paramètre de type chaîne de caractères qui est ajouté à la liste. Pour appeler cette méthode, on doit écrire :
*composant_liste*.Items.Add(*Chaîne)*
Utilisons donc cette méthode pour ajouter un élément dans la liste lors d'un clic sur le bouton 'Lancer le test'. Servez-vous de l'extrait de code ci-dessous pour complèter la procédure de réponse au clic sur ce bouton :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('Ca marche !');
end;
```

La seule instruction présente dans cette procédure s'adresse à 'ListBox1' (que nous aurions dû renommer en 'lbTest' par exemple). Elle ajoute la chaîne « Ca marche ! » à la liste à chaque clic sur le bouton (essayez !).
Cet exemple ne permet cependant pas de vider la liste. Pour cela, il faudrait utiliser la méthode 'Clear' de 'Items'. Essayez donc de remplacer l'ancien code par celui-ci :

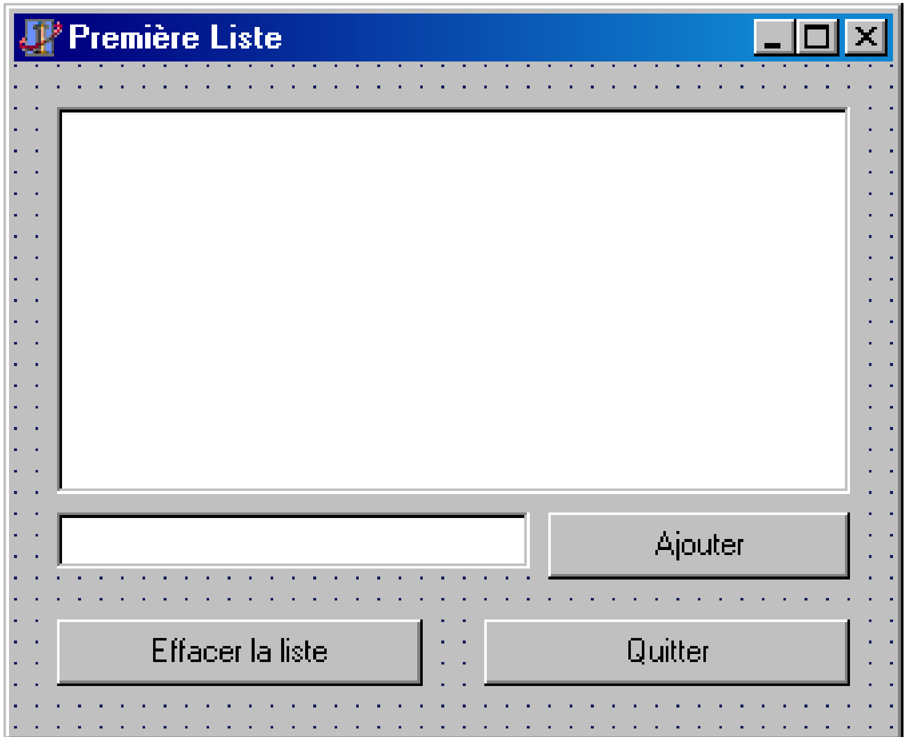
```
ListBox1.Items.Clear;
ListBox1.Items.Add('Ca marche aussi !');
```

La liste est vidée avant l'ajout de la chaîne, ce qui donne un effet visuel moins intéressant, mais a pour mérite de vous avoir appris quelque chose.
Maintenant que vous connaissez le nécessaire, passons à un exercice grandeur nature : un mini-projet.
**Mini-projet n°2**
(voir **Indications, Solution et Téléchargement**)
L'objectif de ce mini-projet est de vous faire créer une application exploitant les notions que vous venez de découvrir, à savoir :

- Gestion de composants, propriétés et méthodes.
- Utilisation d'une zone d'édition, d'une liste.

Si vous avez suivi le début de ce chapitre, vous avez toutes les connaissances nécessaires à la création de ce mini-projet. Des indications, une solution guidée pas à pas ainsi que le téléchargement sont proposés **ici**. Voici maintenant l'énoncé du problème.
Le but de ce projet est, à partir de l'interface que voici, composée d'une liste, d'une zone de saisie et de trois boutons, de permettre les actions mentionnées ci-dessous :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




- Ajout de texte dans la liste : pour cela, on entre le texte dans la zone d'édition puis on clique sur le bouton. Le texte est alors ajouté à la liste et la zone de saisie est vidée (le texte y est effacé).
- Effacement de la liste : en cliquant sur le bouton adéquat, le contenu de la liste sera effacé.
- Fermeture de l'application, en cliquant sur 'Quitter'.

Bon courage !

## IX-E - Evénements

Depuis le début de ce guide, je ne cesse de vous parler des événements, qui sont d'une importance considérable. En effet, les événements sont le coeur du style de programmation employé sous Delphi : les événements ne sont pas seulement une possibilité offerte aux programmeurs, mais presque un passage obligé puisque c'est le style de programmation privilégié sous un environnement graphique tel que Windows. Nous consacrerons donc un paragraphe entier à l'étude du style de programmation à l'aide d'événements, puis nous passerons à leur utilisation en pratique.

### IX-E-1 - Style de programmation événementiel

Aux débuts de l'informatique, l'utilisation des ordinateurs se résumait souvent à concevoir un programme, le coder, l'exécuter et traiter les résultats. Depuis, l'apparition des PC a permis de se ménager une certaine liberté quant à l'utilisation d'un ordinateur. L'apparition de Windows, dans le monde du PC que vous connaissez, a favorisé un style non directif : on n'oblige en aucun cas l'utilisateur à suivre une démarche précise et prévisible : on sait seulement qu'il sera susceptible d'accomplir un certain nombre d'actions, auxquelles on est apte à réagir. Pour bien marquer la différence entre les deux grands styles de programmation, à savoir la programmation directive et la programmation événementielle, choisissons de réfléchir à un petit programme destiné à calculer des additions.
Premier programme, en style directif : vous demandez un premier nombre, puis un second, puis vous donnez le résultat. L'utilisateur ne peut rien faire d'autre que vous donnez les informations et attendre le résultat, il n'a aucune liberté.
Second programme, en style non directif : vous donnez à l'utilisateur deux endroits pour rentrer ses deux nombres, un espace ou sera affiché le résultat, et un bouton qui effectuera le calcul. L'utilisateur n'est alors pas astreint à respecter l'ordre normal : il peut entrer un premier nombre, en entrer un second, puis constater qu'il s'est trompé et corriger


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

le premier, quitter sans effectuer d'addition, et calculer le résultat quand bon lui semble. Ce style paraît avantageux mais oblige le programmeur à montrer plus de vigilence : l'utilisateur peut sans le vouloir sauter des étapes dans le processus logique, en oubliant par exemple de donner l'un des deux nombres (évidemment, avec cet exemple, c'est peu vraisemblable, mais imaginez une boite de dialogue de renseignements sur une personne, où l'on omettrait un renseignement indispensable, comme la date de naissance).
Autre exemple directif : la procédure d'installation d'un logiciel : on vous guide dans l'installation divisée en étapes, dont certaines vous proposent quelques choix, mais sans vous permettre de passer directement à la copie des fichiers sans passer par le choix des programmes à installer. Le style directif s'impose alors, et on peut reconnaître son utilité car on voit mal un programme d'installation commencer à choisir les fichiers à installer avant de rentrer un numéro de série pour le logiciel.
Dernier exemple non directif : Windows en lui-même : vous êtes complètement libre, sous Windows, de lancer telle ou telle application, et de faire ainsi ce que vous voulez : à un moment vous travaillerez sous votre tableur préféré, et à l'autre vous travaillerez à l'extermination d'envahisseurs extra-terrestres, deux occupations que vous auriez pu choisir dans l'ordre inverse : vous êtes libre, c'est le style non directif.
Sous Delphi, c'est le style non directif qui est privilégié, avec aussi bien entendu la possibilité de créer un programme très directif. La programmation des applications s'en ressent, et en permier lieu par l'utilisation des événements, ce qui lui donne le nom de programmation événementielle. Ces événements permettent de laisser l'utilisateur libre, et de simplement réagir selon ce qu'il fait : clic sur un bouton, entrée de texte dans une zone de saisie, clic de souris, etc...
Il existe des événements de diverses nature. Un événement s'applique la plupart du temps à un composant (mais aussi parfois à une application entière) : c'est le composant ou la fiche qui déclenche l'événement. Delphi permet d'assigner une procédure à chacun de ces événements, destinée à réagir à l'événement comme il se doit.
Dans une application, de très nombreux événements se produisent. Il n'est pas question de réagir à TOUS ces événements : lorsqu'on n'affecte pas de procédure à un événement, le traitement de l'événement est fait pas Windows. Par exemple, lorsque vous n'affectez pas de procédure à l'événement « clic de la souris » d'un bouton, un clic produit toujours l'effet classique d'un bouton, à savoir l'enfoncement puis le relâchement, mais rien de plus.
Ce qu'il est important de comprendre, c'est qu'on fait généralement fonctionner un programme basé sur une interface en répondant à un éventail d'événements adaptés à l'application. Un programme qui effectue des additions n'a que faire des mouvements du pointeur de la souris, tandis qu'un logiciel de dessin aura tendance à s'intéresser de très près à ces mouvements, transmis au programmeur sous forme d'événements. Nous avons le plus souvent fait fonctionner nos programmes de test en cliquant sur des boutons, ce qui nous a permis d'éxécuter à chaque fois une ou des procédures.
C'est ainsi désormais que nous allons faire « vivre » nos programmes : nous construirons l'interface, puis nous lui donnerons vie en répondant à certains événements ciblés.

## IX-E-2 - Utilisation des événements

Venons-en maintenant à l'utilisation des événements. Ces derniers sont listés dans l'inspecteur d'objets, dans l'onglet « Evénements ». Comme pour les propriétés, la liste est spécifique à chaque composant : chaque composant, de même que chaque fiche, a sa liste d'événements. Il est possible d'assigner une procédure à un des événements d'un composant en suivant le processus suivant :

1. Sélectionnez le composant qui déclenche l'événement
2. Allez dans l'inspecteur d'objets (F11) dans l'onglet 'événements'.
3. Effectuez un double-clic sur la zone blanche en face du nom de l'événement que vous souhaitez traiter. La procédure sera générée et appelée à chaque occurence de l'événement.

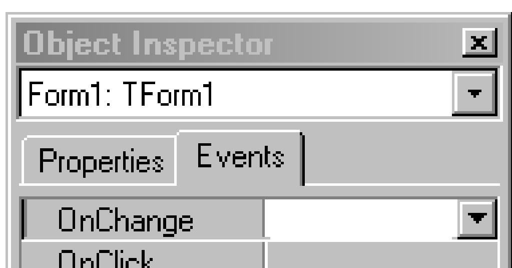
Les événements déjà attachés à des procédures montrent le nom de cette procédure en face de leur nom. Le nom de la procédure est constitué par défaut du nom du composant, puis du nom de l'événement auquel on a retiré le préfixe 'On' présent dans tous les noms d'événements.
Suivant les événements, les paramètres de la procédure changeront. Une procédure d'événement a toujours un paramètre 'Sender' de type 'TObject' que nous décrirons plus tard mais qui ne sert à rien dans une bonne partie des cas.
Une fois que vous disposez de la procédure (son « squelette »), vous pouvez y inscrire des instructions, y déclarer des constantes, des variables. Cependant, il y a certains points sur lesquels vous devez faire attention :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Lorsque vous compilez le projet, les procédures vides (sans déclarations ni instructions ni commentaires) associées aux événements sont automatiquement supprimées car parfaitement inutiles. La référence à cette procédure dans l'inspecteur d'objets est également supprimée car la procédure n'existe plus.
- Pour supprimer une procédure associée à un événement, c'est-à-dire pour ne plus répondre à cet événement, il ne faut pas supprimer la procédure entière car Delphi serait dans l'incapacité de gérer cela. La procédure à suivre est la suivante :

    1. Supprimez toutes les instructions, commentaires et déclarations de la procédure : ne laissez que le squelette, c'est-à-dire la ligne de déclaration, le **begin** et le **end.**
    2. A la prochaine compilation, la procédure sera supprimée, de même que la référence à celle-ci dans l'inspecteur d'objets.

A la lumière de ces explications, vous comprenez certainement mieux maintenant le pourquoi de la procédure de réponse au clic sur les boutons : double-cliquer sur un bouton revient à double-cliquer dans l'inspecteur d'objets sur la zone blance associée à l'événement nommé « OnClick ». Essayez pour vous en convaincre.
La quasi-totalité des événements que vous rencontrerez sous Delphi commencent par le préfixe « On » : c'est un signe de reconnaissance des événements. Chaque composant dispose de son propre ensemble d'événements, mais d'un composant à un autre de même type (deux boutons par exemple), la liste des événements disponibles sera la même, mais la réponse (la procédure associée) sera différente.
Par exemple, le type de composant Edit propose l'événement OnChange, alors que les boutons ne proposent pas cet événement. Si vous utilisez plusieurs composants Edit, vous pourrez associer une procédure à l'événement OnChange de chacun.
Permier exemple pratique : dans un projet vierge, posez un composant Edit sur la fiche principale. Dans l'inspecteur d'objets, trouvez l'événement OnChange de ce composant :




Double-cliquez sur la zone blanche : une procédure est générée : le code source devrait en être :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin

end;
```

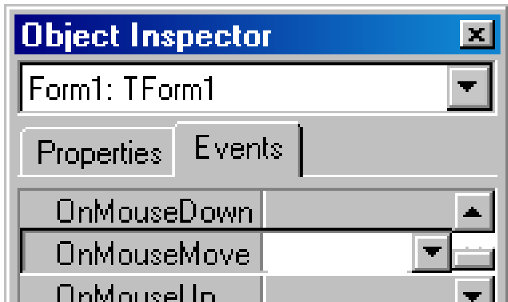
Et entrez l'instruction suivante :

```
Caption := Edit1.Text;
```

Lancez l'application et tapez quelque chose dans la zone d'édition : vous devez constater que le texte dans la barre de titre de l'application est désormais le même que celui que vous tapez dans la zone de saisie. L'explication est assez simple : nous avons utilisé l'événement 'OnChange'. Cet événement se produit à chaque fois que le texte de la zone de saisie (propriété 'Text') change. Dés que vous tapez quelque chose, ou effacez du texte, la procédure 'Edit1Change' est appelée. C'est cette procédure qui fait le changement : elle assigne à 'Caption' (propriété de la fiche) la valeur de la propriété 'Text' de la zone d'édition 'Edit1', ce qui permet aux deux textes d'être identiques puisqu'à chaque modification, l'un est fixé identique à l'autre.
Lorsque vous modifiez le texte dans la zone d'édition, cela change tout d'abord la valeur de la propriété 'Text'. Ce changement fait, l'événement 'OnChange' est déclenché, et appelle la procédure qui lui est attaché, qui modifie la


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

barre de titre de l'application. L'impression donnée par ce programme est de pouvoir fixer le titre de l'application : l'utilisateur n'a pas conscience des événements déclenchés par la saisie au clavier.
Difficile de faire un choix parmi les nombreux événements pour vous les montrer... Jettons un oeil à l'événement 'OnMouseMove' : cet événement est déclenché à chaque fois que la souris bouge « au dessus » du composant, en nous transmettant des informations fort utiles, comme les coordonnées de la souris et l'état des touches Shift, Control, Alt et des boutons de la souris. Cet événement a donc tendance à se produire des dizaines de fois par secondes : il faudra bien faire attention à ne pas y mettre d'instructions qui prendront trop de temps d'éxécution, sous peine de résultats assez imprévisibles.
Ajoutez une deuxième zone d'édition au projet précédent, et nommez les deux zones 'edX' et 'edY'. Veillez à dimensionner la fiche pour pouvoir balader la souris sans qu'elle souffre de claustrophobie. Nous allons utiliser l'événement 'OnMouseMove' de la fiche pour suivre les coordonnées de la souris. Allez donc chercher l'événement 'OnMouseMove' de la fiche :




Double-cliquez sur la zone blanche et entrez les deux instructions que vous pouvez voir ci-dessous :

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  edX.Text := 'X = ' + IntToStr(X);
  edY.Text := 'Y = ' + IntToStr(Y);
end;
```

Exécutez l'application : lorsque vous bougez la souris, les coordonnées relatives à la fiche s'inscrivent dans les deux zones d'édition. Le fonctionnement de l'application est simple : à chaque mouvement de la souris, l'événement 'OnMouseMove' est déclenché et transmet les coordonnées X et Y de la souris. Ces coordonnées sont transformées en chaînes de caractères, ajoutées à un texte de présentation et inscrites dans les deux zones d'édition. Vous remarquerez certainement que lorsque le pointeur passe sur les zones d'édition, les coordonnées ne sont plus mises à jour : le pointeur n'est en effet plus sur la fiche mais sur l'une des zones d'édition contenues dans la fiche.

## IX-F - Composants invisibles

Certains composants ont une particularité des plus intéressante : ils sont invisibles. Non pas que leur propriété 'Visible' soit fixée à 'False', non, ils ne sont purement et simplement invisibles. Ces composants, lorsqu'ils sont placés sur une fiche, sont représentés par un petit cadre montrant l'icône qui les représente dans la palette des composants (nous parlerons dans ce guide de pseudo-bouton). Un exemple est "MainMenu" : lorsque vous placez un tel composant sur une fiche, une sorte de bouton est placé sur la fiche, mais si vous lancez immédiatement le projet sans rien modifier, ce bouton n'est pas visible : le composant que vous venez de placer est invisible.
Ce pseudo-bouton placé sur la fiche n'est visible que sous Delphi : c'est une sorte d'interface entre vous et le composant qui est vraiment invisible. Cette interface est faite pour vous permettre de sélectionner le composant, de changer ses propriétés, ses événements. Déplacer ce pseudo-bouton n'a aucun effet sur le composant : il est non visuel et n'a donc pas d'emplacement. Déplacez ces pseudo-boutons si vous le souhaitez pour bien les ranger quelque part sur la fiche si vous le voulez : ils ne seront de toute façon pas visibles à l'exécution de l'application.
L'utilité de tels composants ne paraît pas forcément évidente au néophyte que vous êtes peut-être : n'ai-je pas dit que les composants permettaient de créer une interface ? Et une interface est visible, non ? Et bien comme partout, il y a des exceptions : les composants invisibles rendent de nombreux services. Les plus courants sont ceux présents sur l'onglet 'Dialogs' : ces composants invisibles permettent l'accès à des fenêtres standard de Windows telles que les


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

fenêtres d'ouverture ou d'enregistrement de fichiers, de choix de couleurs : vous pilotez ces fenêtres très simplement via un composant, au lieu de devoir recréer ces fenêtres standards au moyen de fiches. Il n'est pas concevable de montrer ces fenêtres standards à l'intérieur des fiches, ce qui force le choix de composants invisibles. Vous pouvez utiliser ces fenêtres à volonté : les montrer, les cacher, sans que l'utilisateur ne les ait sous les yeux en permanence. Un dernier composant invisible dont vous devez connaître l'existence est le composant "ImageList". Ce composant, invisible, permet de stocker une liste d'images numérotées. Ce composant seul ne sert à rien, mais certains composants comme les menus, les arbres ou les listes d'icônes en ont absolument besoin. Vous apprendrez donc à vous servir de ce composant au chapitre 8, qui comporte plusieurs parties dédiées au fonctionnement de ces composants invisibles.

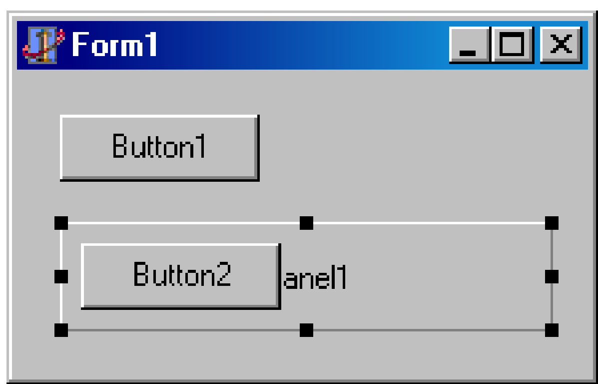
## IX-G - Imbrication des composants

Contrairement à ce que l'on pourrait penser, placer un ensemble de composants sur une fiche ne suffit pas toujours à constituer une interface viable : il faut parfois constituer des interfaces défiant cette optique. Pensez pour bien comprendre ce point de vue à un classeur à onglets tel que la boite de dialogue d'options de Word : plusieurs pages sont présentes sur la même fiche, la plupart sont cachés pour ne laisser visible que le contenu de l'onglet sélectionné. Tout ceci serait evidemment possible sans regrouper les composants, mais au prix d'efforts et de contraintes dissuasives (il faudrait masquer et afficher manuellement un nombre incroyable de composants à chaque changement d'onglet actif, ce qui serait non seulement long et pénible, mais aussi une source intarissable d'erreurs et de bugs).
Le regroupement des composants est une des facettes de l'imbrication des composants : il est en effet possible d'imbriquer certains composants dans d'autres. Il existe à ce niveau deux familles de composants : ceux qui peuvent contenir d'autres composants, et ceux qui ne le peuvent pas. Parmi ceux qui peuvent en contenir d'autres, ceux auquels on ne pense pas sont... les fiches ! En effet, vous placez depuis le début de ce guide des composants sur les fiches, qui SONT des composants (spéciaux, il est vrai, mais non moins des composants tout à fait dignes de cette appellation). Parmi ceux qui ne peuvent contenir d'autres composants, je citerai les boutons, les cases à cocher, les zones d'édition, les listes, et beaucoup d'autres qui constituent en fait la majorité des composants.
Les composants qui peuvent en contenir d'autres, dits composants « conteneurs », sont assez peu nombreux, mais d'un intérêt majeur : on compte, en dehors des fiches les composants "Panel", "GroupBox", "PageControl" et "ToolBar" pour ne citer que les plus utilisés.
Regrouper les composants au sein de composants conteneurs a plusieurs avantages :

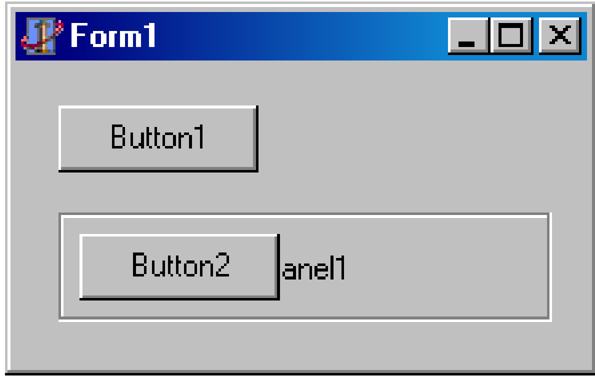
- Avantage visuel : le composant "Panel" permet par exemple de créer des cadres en relief, mettant en évidence le regroupement des composants.
- Avantage logique : lorsque vous construirez un classeur à onglet sous Delphi, il sera plus naturel de sélectionner un onglet, puis d'y placer les composants qui devront apparaître lorsque cet onglet est actif. Changez d'onglet sous Delphi et seul le contenu de cet onglet est visible, laissant de coté le reste des composants des autres onglets.
- Avantage de conception : imaginez ce que serait l'interface d'une fiche si tous ses composants étaient visibles simultanément : ce serait inregardable, un fouillis incompréhensible.
- Avantage de placement : les composants contenus dans d'autres se déplacent avec leur conteneur, ce qui permet de déplacer facilement tout un ensemble d'un endroit à un autre.

Nous n'allons étudier brièvement ici qu'un de ces composants : le composant "Panel" (qui sera étudié plus en détail au chapitre 8). Ce composant est un conteneur, qui peut donc recevoir d'autres composants. Faites l'expérience : placez un "Panel" sur une fiche, puis placez deux composants identiques (des boutons par exemple), l'un sur la fiche (cliquez pour cela en dehors du "Panel" au moment de placer le bouton) et un sur le "Panel" (cliquez pour cela sur le panel pour placer le bouton. Vous devriez obtenir quelque chose de ce genre :

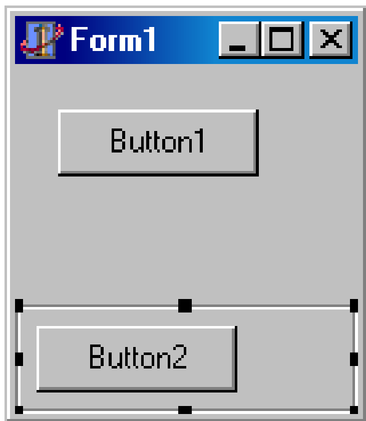

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Essayez maintenant de déplacer le bouton placé à l'intérieur du panel : vous constaterez qu'il vous est impossible de franchir les bords du panel. Le panel constitue en fait une sous-zone de la fiche que vous pouvez agencer comme vous le voulez, et que vous pouvez ensuite placer ou aligner où vous voulez. Le composant Panel a un autre intérêt : avec les propriétés "BevelInner", "BevelOuter", "BevelWidth" et "BorderStyle", il est possible de constituer une bordure personnalisée des plus appréciable. Attention à ne pas tomber dans l'excés à ce niveau, ce qui est en général le propre des débutants, il suffit souvent d'une bordure très fine comme ci-dessous :




Essayez maintenant de déplacer le panel et non le bouton : vous voyez que le bouton reste à l'intérieur du panel, au même emplacement relatif au panel. Fixez maintenant la propriété "Align" du panel à "alBottom" : le panel est maintenant bloqué en bas de la fiche. Redimensionnez la fiche, et vous aurez l'agréable surprise de voir votre panel rester en bas de celle-ci, et se redimensionner pour coller parfaitement au bas de la fenêtre, comme sur la capture ci-dessous :




Grâce à un tel composant, il sera facilement possible de créer des boutons restant à leur place en bas d'une fenêtre, quelle que soit la taille de celle-ci. Même si dans certains cas l'utilisateur n'aura pas à redimensionner la fiche, vous, le programmeur, aurez peut-être besoin de le faire, sans avoir à vous soucier des détails du style emplacement des boutons.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour plus de détails sur l'utilisation des composants conteneurs, consultez les parties du chapitre 8 consacrées à ces composants.

## IX-H - Types de propriétés évolués

Cette partie présente succintement certains types particuliers de propriétés, ainsi que des caractéristiques importantes. Les notions abordées ici ne le sont que pour permettre d'aborder plus facilement le chapitre 8. Ces notions sont expliquées en détail dans le long chapitre consacré aux objets.

### IX-H-1 - Propriétés de type objet

Petit retour obligé ici sur les propriétés. La connaissance de l'utilisation des méthodes étant indispensable ici, ce paragraphe a volontairement été placé en dehors de celui réservé aux propriétés.
La plupart des propriétés que vous avez vu jusqu'ici sont de types connus par vous, à savoir des types simples comme des booléens, des entiers, des chaînes de caractères ou des types énumérés. Certaines propriétés, cependant, sont plus complexes car elles sont elles-mêmes de type objet ou même composant. Ces propriétés affichent dans l'inspecteur d'objets une valeur entre parenthèses du style "(TStrings)" et un bouton ... La propriété "Items" des composants "ListBox", que vous avez déjà eu l'occasion de manipuler, en est un bon exemple : en cliquant sur le petit bouton, on accède à une interface spécifique d'édition de la propriété.
Ces propriétés sont certes un peu moins faciles à manipuler que les propriétés simples, mais cela veut simplement dire qu'elles ont plus de puissance. On manipule en fait ces propriétés comme on manipulerait un composant. Sous Delphi, une interface d'édition sera proposée pour vous simplifier la vie. Au niveau du code, on manipule ces propriétés comme tout composant, à savoir qu'elles ont des propriétés et des méthodes mais en général pas d'evénements. Il est possible d'utiliser de manière classique ces "sous-propriétés" et ces "sous-méthodes" (on oubliera très vite ces vilaines dénominations pour ne plus parler que de propriétés et de méthodes), comme vous l'avez fait avec la propriété "Items" des composants "ListBox".
Pour accèder à une propriété ou à une méthode d'une propriété objet, il suffit d'employer la syntaxe suivante qui doit vous rappeler celle utilisée avec le composant "ListBox" :

```
composant.propriete_objet.propriete := valeur; // modification d'une propriété
composant.propriete_objet.methode(paramètres); // appel d'une méthode
```

Ces propriétés sont généralement employées dans des composants faisant intervenir des listes, à savoir les "ListBox" que vous connaissez déjà, les "ComboBox" que vous connaissez certainement aussi (liste déroulante), les arbres (pensez à la partie gauche de l'Explorateur Windows), les listes d'icônes ou d'éléments a plusieurs colonnes (pensez à la partie droite de l'explorateur windows), les Mémo (pensez au bloc-notes : chaque ligne est un élément d'une liste). Nous aurons de nombreuses fois l'occasion d'utiliser ces propriétés, il n'était question ici que de les présenter brièvement.

### IX-H-2 - Propriétés de type tableau, propriété par défaut

Certaines propriétés que vous serez amenés à manipuler sont de type tableau. Vous connaissez déjà les tableaux et vous savez donc déjà comment on accède aux éléments d'un tableau. Les propriétés tableau ne sont pas de vrais tableaux, mais sont très proches des tableaux, à savoir qu'on accède à leurs éléments de la même façon que pour de vrais tableaux : par l'utilisation des crochets [] à la suite du nom du tableau. Ici, il faudra simplement remplacer le nom du tableau par le nom d'une propriété tableau, ce qui fait une différence théorique uniquement étant donné que les noms de propriétés et de tableaux sont des identificateurs.
Pour clarifier la situation, voici un petit rappel avec un tableau classique :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Tableau: array[1..10] of integer;
begin
  Tableau[2] := 374;
```

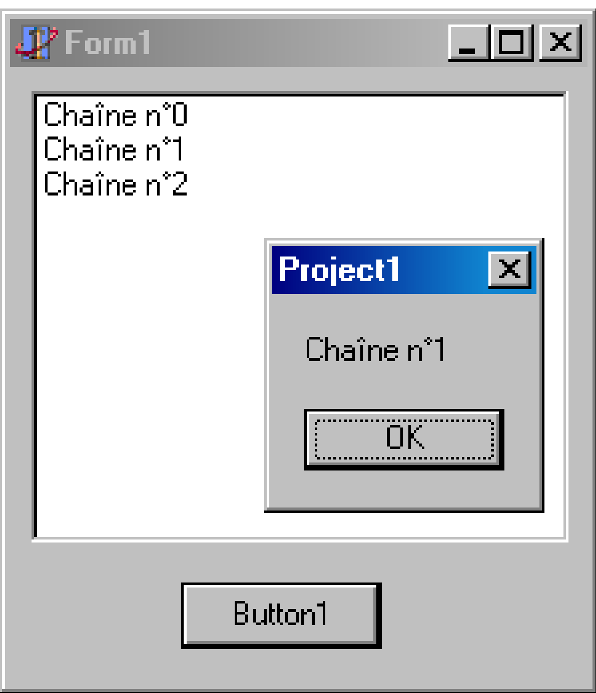

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  ShowMessage(IntToStr(Tableau[2]));
end;
```

Je ne vous ferai pas l'affront d'expliquer ce que fait cet exemple simple. Voici maintenant un autre exemple démontrant l'utilisation d'une propriété tableau. Créez un projet vierge et placez une zone de liste ("ListBox") et un bouton sur l'unique fiche. Nommez la zone de liste "lbTest". Insérez le code présenté ci-dessous dans la procédure de réponse à l'evénement OnClick du bouton :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  lbTest.Items.Clear;
  lbTest.Items.Add('Chaîne n°0');
  lbTest.Items.Add('Chaîne n°1');
  lbTest.Items.Add('Chaîne n°2');
  ShowMessage(lbTest.Items.Strings[1]);
end;
```

Cet exemple est très riche dans le sens où il combine deux notions importantes, à savoir les propriétés objet et les propriétés tableau. Dans cet exemple, la propriété "Items" de "lbTest" est une propriété objet. Elle possède ses propriétés et ses méthodes. Parmi les méthodes, "Clear" permet d'effacer le contenu de la liste, et "Add" permet d'ajouter une chaîne (mais vous savez déjà cela si vous avez suivi ce qui précède).
Ce qui est nouveau, c'est la propriété "Strings" de "Items" : cette propriété est une propriété tableau. Les indices sont des entiers et les éléments du tableau sont des chaînes de caractères, qui sont précisément les chaînes affichées dans la liste. Comme beaucoup de propriétés tableau, "Strings" est indicée à partir de 0, c'est-à-dire que l'élément 0 est le premier du tableau et que l'élément n est le (n+1)-ième. Dans l'exemple ci-dessous, on prends le deuxième élément de la liste (le deuxième ajouté), qui est lbTest.Items.Strings[1], et on l'affiche, ce qui donne :




Les propriétés tableau sont des propriétés des plus utiles, et nous en reparlerons lorsque nous aborderons en détail les composants qui les utilisent. Pour l'instant, il me faut contenter certains lecteurs habitués à Delphi qui auront certainement tiqué à la lecture de l'exemple précédent. En effet, la dernière ligne de la procédure serait plutôt écrite comme cela par ces lecteurs :

```
ShowMessage(lbTest.Items[1]); // le ".Strings" a disparu !
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

C'est une écriture valable, et pourtant comment expliquer qu'on puisse se passer d'un morceau de code tel que ".items", étant donné qu'il fait appel à une propriété ? La réponse est que cette propriété, ne se satisfaisant pas d'être déjà une propriété tableau d'une propriété objet, a une caractéristique supplémentaire : elle est *par défaut*. Cette caractéristique réservée à une seule propriété tableau par composant ou propriété objet permet de raccourcir l'accès à cette propriété en n'écrivant plus explicitemet l'appel à la propriété tableau (qui est fait implicitement dans ce cas).
Dans le cas de la propriété "Items" des composants "ListBox", les deux écritures suivantes sont donc parfaitement équivalentes :
lbTest.Items[1]
lbTest.Items.Strings[1]
La deuxième ligne est simplement plus longue que la première, c'est pourquoi on ne se privera pas d'utiliser exclusivement la première.

## IX-H-3 - Propriétés de type référence

Certaines propriétés, vous le verrez, ne sont pas de vraies propriétés au sens où nous l'entendons depuis le début : ce sont juste des références à d'autres éléments, tels que des composants la plupart du temps. Contrairement au modèle classique montré au début de ce chapitre, la propriété n'est pas entièrement stockée dans le composant, mais se trouve en dehors de celui-ci. A la place, se trouve une simple référence, un lien vers l'élément référencé, à savoir un composant la plupart du temps.
Prenons un exemple : lorsque vous créérez un menu pour votre application, vous utiliserez un composant "MainMenu". Vous serez certainement tentés de placer des images à coté des éléments de menus, comme cela semble être la mode actuellement. Ceci passe par l'utilisation d'un composant "ImageList" invisible. Ce composant doit ensuite être mis en relation avec le menu par l'intermédiaire d'une propriété "Images" de ce dernier. Vous n'avez plus ensuite qu'à piocher dans les images présentes dans le composant "ImageList" pour les placer dans les menus (la création de menus est décrite dans l'**Annexe A du chapitre 8**).
Ce qui nous intéresse ici, c'est la propriété "Images" en elle-même. Cette propriété est de type "TImageList", qui est précisément le type du composant "ImageList". Mais ici, il n'est pas question de stocker le composant "ImageList" dans le composant "MainMenu", puisque le premier est placé sur la fiche et est donc stocké dans celle-ci. La propriété "Images" n'est donc qu'une simple référence au composant "ImageList". Je sais que la nuance peut paraître inintéressante, mais elle est de taille.
Quant à l'utilisation de ces propriétés références, c'est un vrai plaisir. Supposons que notre menu soit nommé "MainMenu1" et notre "ImageList" soit nommée "ImageList1". Les deux écritures suivantes sont alors équivalentes :

```
MainMenu1.Images
ImageList1
```

ce qui signifie simplement que vous pouvez utiliser une référence comme l'élément lui-même. Il faudra cependant faire attention à vérifier qu'une référence est valable avant de l'utiliser.
Nous reviendront en détail sur ces notions de références dans deux chapitres, l'un consacré aux pointeurs et l'autre consacré aux objets. C'est un sujet qui ne peut hélas qu'être effleuré tant qu'on n'a pas vu ces deux notions plus complexes.

## IX-I - Conclusion

Ceci termine ce chapitre consacré à la manipulation des composants. Ce chapitre n'est en fait qu'une entrée en matière pour le chapitre suivant. Vous savez maintenant ce que sont les composants, les propriétés, les méthodes, les événements. Vous savez également comment utiliser ces éléments. Des composants particuliers comme les composants invisibles ou conteneurs ont été abordés. Ce qu'il vous manque maintenant, c'est une visite guidée des différents composants utilisables dans Delphi, avec les propriétés, méthodes et événements importants à connaître pour chacun d'eux. C'est l'objet du chapitre 8.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# X - Découverte des composants les plus utilisés - 1ère partie

L'utilisation des composants ImageList est démontrée dans l'**Annexe 8-A : Création de menus**.
Ceci termine ce chapitre qui n'est qu'une première partie. La deuxième partie, présentée plus tard, sera consacrée à l'étude de composants plus délicats ou complexes à manipuler. Ces composants demandent pour être utilisés des connaissances que vous ne possèdez pas encore.
VIII.3. Mini-projet
Je tiens absolument à terminer ce chapitre par un mini-projet : le premier mini-projet dont vous pourrez réaliser entièrement l'interface et le code source, en répondant à des exigences définies. La nature de ce projet dérangera ceux d'entre vous qui se sont jurés d'oublier les mathématiques, car il est en plein dans ce domaine.
**Accèder au mini-projet** (attention à la tentation : les solutions sont données juste à la fin de l'énoncé.
Le but de ce chapitre un peu particulier est d'offrir la première partie d'une visite guidée des composants les plus utilisés sous Delphi. Pour cette première partie, nous nous limiterons aux composants moins complexes et très couramment utilisés, avec leurs propriétés, leurs méthodes et leurs événements. Cette visite n'est en aucun cas exhaustive étant donné

1. le nombre impressionnant de composants proposés sous Delphi.
2. que certains composants ne sont pas disponibles dans toutes les versions de Delphi.
3. que des milliers de composants sont disponibles sur Internet : aucun de ces composants n'est abordé ici.

La propriété "Name" ne sera jamais mentionnée : elle est déjà connue et valable pour tous les composants. Dans le même ordre d'idée, certains événements standards tels "OnClick" ne seront pas mentionnés à chaque fois.
Pour chaque composant abordé, une présentation générale est donnée : utilisations principales ou recommandables, limitations, avantages et caractéristiques au niveau du code Pascal. Vient ensuite une liste des propriétés, méthodes et événements importants pour ce composant. Chaque fois que ce sera possible, les explications seront illustrées par un exemple et/ou une manipulation. Je rajouterai des exemples si vous en demandez.
Si vous souhaitez qu'un composant non traité ici le soit, merci de **me contacter** ou utiliser la **mailing-list** du guide. Veuillez cependant veiller à ce que le composant soit présent dans les versions de Delphi 2 ou ultérieur, professionnel ou entreprise. Si je traite ce composant, ce sera dans un chapitre différent de celui-ci.
Enfin, dans les titres de parties, les noms des composants sont données sans le 'T' initial qui est présent dans le code source lorsqu'on utilise le composant. Ce sera une convention désormais : lorsque nous parlerons d'un composant sans mentionner le 'T', nous parlerons d'un composant en général, non placé sur une fiche. Lorsque nous préciserons le 'T', ce sera soit pour parler du code source ou d'un composant particulier présent sur une fiche. Cependant, vous vous habituerez vite à utiliser les deux dénominations et la distinction sera bientôt mise de coté (hélas).

## X-A - La fiche : Composant "Form"

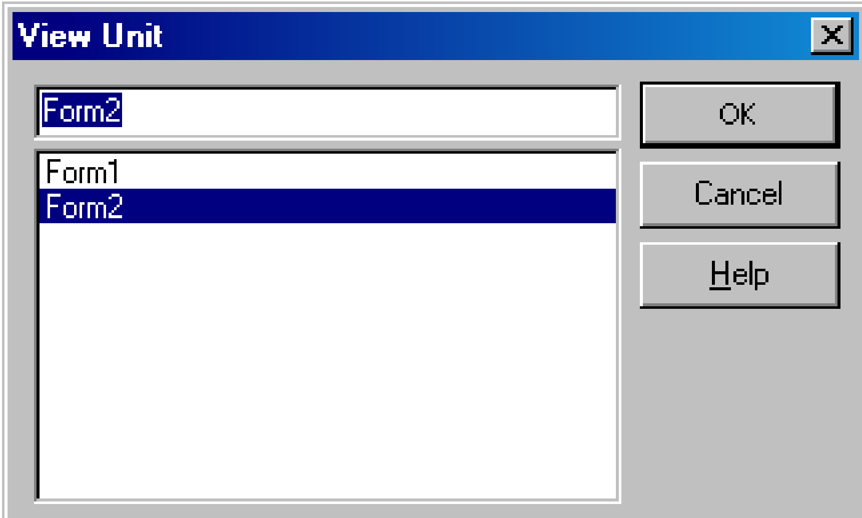
Il est de bon ton de commencer ce chapitre par un composant incontournable : la fiche. Vous travaillez déjà depuis un certain temps dessus sans vraiment penser au composant qu'elle représente, et pourtant, elle a ses propriétés, ses événements, ses méthodes et son interface. C'est LE composant conteneur classique qu'on utilise le plus souvent sans se rendre compte de cette fonctionalité pourtant indispensable.
Voici un petit tableau qui donne les caractéristiques importantes des fiches. Ce genre de tableau sera répété pour chaque composant dans la suite du chapitre.
**Fiche technique :**

| Icône | (aucune) |
|---|---|
| Visibilité | Visible |
| Conteneur | Oui |

Les fiches ne se créent pas depuis la palette des composants, d'où elles brillent par leur absence, mais par une commande spécialisée de Delphi. Cette commande est accessible par le menu "Fichier", choix "Nouveau..." puis "Fiche", ou encore par un bouton des barres d'outils : . Cette fiche est alors ajoutée au projet actuel (un projet doit être ouvert). Pour effacer cette fiche, il faut aller dans le gestionnaire de projets (menu "Voir", choix "gestionnaire de projet") qui sera traité dans un futur chapitre consacré à l'interface de Delphi.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour voir la liste des fiches d'un projet, il faut utiliser la commande "Fiches..." du menu "Voir" ou le raccourci clavier Shift+F12. Ceci a pour effet de faire apparaître une liste des fiches :




Pour en faire apparaître une, sélectionnez-la puis cliquez sur OK.
Voici maintenant une liste des propriétés intéressantes à connaître pour les fiches :
**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fbeaulieu.developpez.com/guide/

| | |
|---|---|
| BorderIcons | Décide des icônes présentes dans la barre de titre de la fenêtre. Pour éditer cette propriété, cliquez sur le + qui la précède et modifier les sous-propriétés de type booléen qui la composent (La propriété BorderIcons est en fait de type ensemble, vous décidez de quel élément cet ensemble est constitué).<br><br>• biSystemMenu affiche le menu système (en haut à gauche).<br>• biMinimize affiche le bouton de réduction en icône.<br>• biMaximize affiche le bouton d'agrandissement maximal.<br>• biHelp affiche un bouton d'aide. |
| BorderStyle | Permet de fixer le style de bordure de la fenêtre (style appliqué pendant l'exécution de l'application seulement).<br><br>• bsDialog crée une bordure fixe |



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | Permet de fixer l'état de la fenêtre :<br><br>• wsMaximized : donne la taille maximale à la fenêtre (même effet que le clic sur le bouton d'agrandissement).<br>• wsMinimized : réduit la fenêtre en icône (même effet que le clic sur le bouton de réduction en icône).<br>• wsNormal : redonne à la fenêtre son état normal, à savoir non maximisé et non icônisée. |
|---|---|

**Evénements :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| OnActivate | Se produit chaque fois que la fiche est activée, c'est-à-dire lorsqu'elle était inactive (bordure souvent grisée) et qu'elle devient active (bordure colorée). Une fiche est également activée lorsqu'elle devient visible et lorsqu'elle est la fiche active de l'application et que celle-ci devient active. |
| OnClick | Permet de réagir au clic de la souris, vous connaissez déjà bien cet événement pour l'avoir déjà expérimenté avec les boutons. Il fonctionne avec de très nombreux composants. |
| OnClose | Se produit lorsque la fiche se ferme, c'est-à-dire lorsqu'elle devient invisible ou que sa méthode Close est appelée. Le paramètre Action transmis permet certaines manipulations. Nous utiliserons cet événement lorsque nous créerons un "splash- |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Se produit lorsque la fiche est montrée, c'est-à-dire lorsque sa propriété Visible passe de False à True. Cet événement se produit notamment lorsque la méthode Show ou ShowModal de la fiche est appelée. |

**Méthodes :**

| | |
|---|---|
| Close | Ferme la fiche. Vous pouvez obtenir le même effet en fixant la propriété Visible à False. |
| Show | Montre une fiche. Vous pouvez obtenir le même effet en fixant la propriété Visible à True. |
| ShowModal | Montre une fiche, en la rendant modale. Une fenêtre modale reste visible jusqu'à ce que sa propriété ModalResult soit différente de 0. Les fenêtres modales sont abordées ci-dessous. |

Les fenêtres modales sont une possibilité intéressante de Delphi. Une fenêtre devient modale lorsqu'elle est montrée au moyen de sa méthode ShowModal. On la ferme ensuite au choix en fixant une valeur non nulle à sa propriété ModalResult, ou en appelant sa méthode Close qui, en fait, affecte la valeur constante "mrCancel" (valeur 2) à ModalResult. Cette propriété permet de renseigner sur les circonstances de fermeture de la fenêtre, ce qui est souvent


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

très utile. Une fenêtre modale se distingue en outre d'une fenêtre normale par le fait qu'elle doit être refermée avant de pouvoir continuer à utiliser l'application. Une utilisation classique en est faite pour créer des boites de dialogue : ces dernières doivent être refermées avant de pouvoir continuer à travailler dans l'application (prenez la boite de dialogue "A propos de..." de Delphi par exemple).
La propriété ModalResult ferme donc une fiche modale lorsqu'elle est fixée différente de 0. Mais c'est une propriété de type énuméré, c'est-à-dire que ses valeurs sont prises parmi un jeu d'identificateurs ayant des noms significatifs. Pour fermer une fiche en donnant une information sur les circonstances de fermeture de la fiche (OK, annulation, ...), on pioche parmi ces valeurs. Voici une liste de valeurs possibles pour ModalResult (la valeur numérique, qui n'a aucune signification, est donnée à titre indicatif entre parenthèses) :

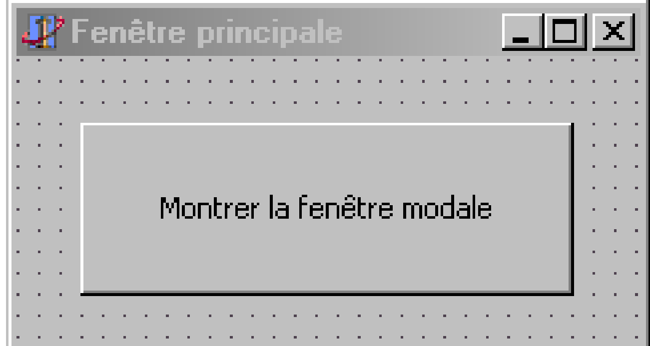
- mrNone (0) : valeur prise par ModalResult lorsque ShowModal est appelée.
- mrOk (1) : signifie que l'utilisateur a validé par un "OK" (ou tout autre moyen qui signifie "OK").
- mrCancel (2) : signifie que l'utilisateur a annulé.
- mrAbort (3) : signifie que l'utilisateur a abandonné.
- mrRetry (4) : signifie que l'utilisateur souhaite réessayer quelque chose (c'est à vous de déterminer quoi !).
- mrIgnore (5) : signifie que l'utilisateur souhaite ignorer.
- mrYes (6) : signifie que l'utilisateur a répondu par l'affirmative à une question.
- mrNo (7) : signifie que l'utilisateur a répondu par la négative à une question.

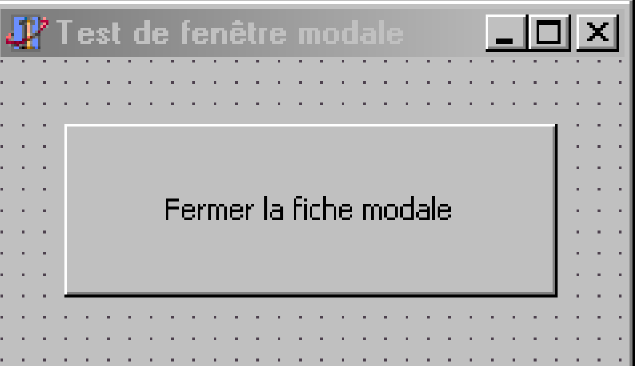
Toutes ces valeurs ne sont bien évidemment pas appropriées dans toutes les situations, elles permettent juste de couvrir un large éventail de circonstances standards de fermeture : souvent seules les valeurs mrOK et mrCancel sont significatives et signifient souvent que l'utilisateur a cliqué sur un bouton "OK" ou "Annuler".

## X-A-1 - Manipulation Guidée

Créez un projet vierge sous Delphi, puis créez une deuxième fiche au moyen de la commande "Nouveau..." du menu "Fichier" (choisissez "Fiche" parmi la liste des éléments proposés). Votre projet contient maintenant deux fiches, l'une nommée "Form1" associée à une unité "Unit1", et l'autre nommée "Form2" associée à une unité "Unit2". L'exercice consiste à montrer la fiche "Form2" en cliquant sur un bouton de la fiche "Form1", et à pouvoir refermer la fiche "Form2" en cliquant sur un bouton de celle-ci.

1. Commencez par enregistrer le projet : "Modales". L'unité "Unit1" doit être nommée "Principale" et "Unit2" nommée "TestModale". La fiche "Form1" doit être nommée "fmPrinc" et "Form2" nommée "fmTestModale".
2. Placez un bouton nommé "btMontreModale" sur la fiche "fmPrinc" (placez le bouton puis renommez-le). Placez un bouton "btFermeFiche" sur la fiche "fmTestModale". Donnez à tout cela une apparence correcte en vous inspirant de la capture d'écran ci-dessous :






3. Il faut maintenant écrire le code qui va faire vivre notre exemple. Il est des plus simple : le bouton de la fiche "fmPrinc" appelera la méthode ShowModal de fmTestModale, tandis que le bouton "btFermeFiche" de fmTestModale se contentera d'assigner une valeur non nulle a la propriété ModalResult de la fiche.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Avant de faire cela, il va falloir accèder à l'unité "TestModale" depuis l'unité "Principale". Ajoutez donc l'unité "TestModale" au bloc "uses" de l'interface de l'unité "Principale" (nous aurions aussi pû utiliser celui de l'implémentation, savez-vous pourquoi ?).

4 Générez la procédure de réponse à l'événement "OnClick" de "fmPrinc.btMontreModale" (on se permettra désormais cette notation pratique empruntée au langage Pascal Objet). Il nous faut ici appeler la méthode ShowModal de la fiche "fmTestModale". Cette dernière est accessible puisque la variable qui la stocke est dans une unité présente dans le bloc "uses" convenable. On doit pour accèder à la méthode "ShowModal" de "fmTestModale" écrire le nom de la fiche, un point, puis la méthode à appeler, soit la très simple instruction présentée ci-dessous :

```
procedure TfmPrinc.btMontreModaleClick(Sender: TObject);
begin
  fmTestModale.ShowModal;
end;
```

5 Générez la procédure de réponse à l'événement OnClick du bouton "btFermeFiche". Nous devons ici modifier la propriété "ModalResult" de la fiche "fmTestModale" à laquelle appartient le bouton "btFermeFiche". On peut donc directement s'adresser à la propriété, puisque le bouton et la propriété sont tous deux des éléments de la fiche. Voici le code source, dans lequel nous affectons la valeur mrOK à ModalResult :

```
procedure TfmTestModale.btFermeFicheClick(Sender: TObject);
begin
  ModalResult := mrOK;
end;
```

6 Lancez alors l'application. Lorsque vous cliquez sur le bouton de la fiche principale, vous faites apparaître la deuxième fiche, qui ne permet pas de revenir à la première tant qu'elle n'est pas fermée. Pour la fermer, un moyen est de cliquer sur le bouton créé à cet effet.

Cet exemple très simple démontre non seulement l'utilisation de plusieurs fiches, mais aussi l'utilisation de fenêtres modales. Vous pouvez modifier cet exemple en remplaçant "ShowModal" par "Show". Dans ce cas, il faudra remplacer l'affectation à ModalResult par un appel à la méthode "Close". Je vous laisse réaliser cette modification simple et l'expérimenter vous-même.
Il est une amélioration que nous allons cependant faire ici. J'ai mentionné auparavant que ShowModal renvoyait la valeur de ModalResult en tant que résultat. En effet, ShowModal est une fonction qui affiche la fiche. Cette fonction se termine lorsque ModalResult devient non nul. Cette valeur est alors renvoyée comme résultat de la fonction. Nous allons tout simplement mémoriser le résultat de l'appel à ShowModal, puis afficher sa valeur. Voici le nouveau code à utiliser dans la procédure de réponse au clic sur le bouton "btMontreModale" :

```
procedure TfmPrinc.btMontreModaleClick(Sender: TObject);
var
  Rep: TModalResult;
begin
  Rep := fmTestModale.ShowModal;
  ShowMessage(IntToStr(Ord(Rep)));
end;
```

La première instruction appelle toujours la méthode ShowModal de fmTestModale, mais mémorise en plus le résultat dans une variable Rep de type TModalResult (c'est le type énuméré qui énumère les constantes mrNone, mrOK, ...). La deuxième instruction prend la valeur ordinale de Rep, puis la transforme en chaîne de caractères, et ensuite l'affiche avec ShowMessage. Le résultat du clic sur le bouton lors de l'exécution de l'application est d'afficher d'abord la fiche fmTestModale, puis immédiatement après sa fermeture une boite de message avec marqué "1". 1 est en effet la valeur ordinale de la constante mrOK. Vous pouvez essayer de changer mrOK par d'autres constantes pour voir leur valeur ordinale.

*Note : Les habitués de Delphi qui liront ceci sauteront une nouvelle fois au plafond en se demandant : "mais pourquoi diable utilise-t-il la fonction "Ord" ?". La réponse, pour tous, est que dans l'état actuel, transmettre une valeur de type TModalResult à une fonction qui veut une valeur de type Integer est interdit. Pour les non-initiés, sachez que cela est en fait*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

*permis, sous certaines conditions. Ceci n'étant pas à l'ordre du jour, nous en reparlerons plus tard.*

Avant de passer à autre chose, voici pour ceux qui le souhaitent la possibilité de **télécharger le projet fini**.

## X-B - Référence des composants

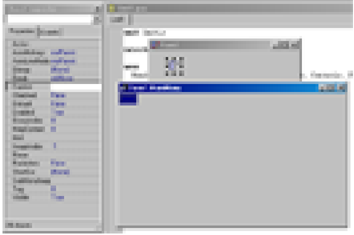
Les composants présentés dans ce paragraphe sont les plus couramment employés : ce sont pour la plupart des composants standards disponibles depuis Windows 3.1. Certains composants très particuliers comme "Frame" ou "ActionList" ne seront pas traités ici. Le composant "Frame" fera l'objet d'un chapitre. Aucun détail sur le composant "ActionList" ne sera donnée, son utilité ne m'ayant pas vraiment été prouvée pour l'instant (Si vous n'êtes pas d'accord, expliquez-moi comment il vous a été utile en **me contactant**).

### X-B-1 - Composant "MainMenu"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Invisible à la création, visible ensuite. |
| Conteneur | Non |

Ce composant permet de munir la fiche d'une barre de menus déroulants comme vous en utilisez très souvent. Ce composant est non visuel lorsqu'on vient de le poser sur une fiche : au lieu d'un composant visible, l'icône du composant vient se placer sur la fiche. A partir de son pseudo-bouton, vous pouvez accèder via un double-clic à une interface spécifique de création de menus. Cette interface permet de créer directement de façon visuelle les menus en utilisant l'inspecteur d'objets et quelques raccourcis clavier. Cette interface permet en fait d'éditer la propriété "Items" du composant (en sélectionnant la propriété, puis en cliquant sur le bouton qui permet d'éditer des propriétés complexes, vous accèdez à la même chose). Voici une capture d'écran de cet éditeur :



Le fonctionnement de cet éditeur et la création de menus forment un sujet très vaste et passionnant qui est traité de façon très détaillée en **Annexe A**. Je vous conseille avant de vous lancer dans le sujet de lire ce paragraphe et celui consacré au composant "ImageList".
Les propriétés importantes du composant "MainMenu" sont décrites ci-dessous :
**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Images | Référence à un composant "ImageList". Permet d'associer à un menu une liste d'images stockées dans un composant "ImageList". |
| Items | Propriété objet. Permet l'accès à l'éditeur de menus. |

## X-B-2 - Composant "TPopupMenu"

**Fiche technique :**

| | |
|---|---|
| Icône | |
| Visibilité | Invisible à la création, peut être visible à l'éxécution |
| Conteneur | Non |

Ce composant permet de créer un menu contextuel. Les menus contextuels, je le rappelle, sont ceux qui apparaissent lorsqu'on clique avec le bouton droit de la souris. Ce genre de menu doit son nom au fait qu'il semble adapté à la zone sur laquelle on clique. En fait, il faut souvent dans la pratique créer plusieurs menus contextuels et les adapter éventuellement (en activant ou désactivant, ou en cachant ou en montrant certains choix) pendant l'exécution de l'application. La création de ce menu se fait dans le même genre d'interface que les menus principaux ("MainMenu"), vous pouvez d'ailleurs consulter l'**Annexe A** qui comprend une section décrivant l'usage des menus contextuels.
**Propriétés :**

| | |
|---|---|
| Alignment | Spécifie l'alignement du menu par rapport à l'endroit où le clic droit a eu lieu. "paLeft" est la valeur habituelle et par défaut. |
| Images | Référence à un composant "ImageList". Permet d'associer au menu une liste d'images stockées dans un composant "ImageList". |
| Items | |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Propriété objet. Permet l'accès à l'éditeur de menus. |

**Evénements :**

| | |
|---|---|
| OnPopup | Se produit juste avant que le menu contextuel soit montré. Utiliser une procédure répondant à cet évenement pour décider à ce moment de l'apparence du menu (éléments (in)visibles, (dés)activés, cochés, ...) |

**Méthodes :**

| | |
|---|---|
| Popup | Permet d'afficher directement le menu contextuel. Vous devez spécifier des coordonnées X et Y dont la signification varie suivant la valeur de la propriété "Alignment" |

Pour utiliser un menu contextuel, il faut (tout est décrit en détail dans l'**Annexe A**) l'associer à chaque composant qui doit le faire apparaître. Ceci est fait en sélectionnant le menu dans la propriété "PopupMenu" (propriété de type référence) des composants (une très grande majorité le propose).

## X-B-3 - Composant "Label"

**Fiche technique :**

| | |
|---|---|
| Icône | A |
| Visibilité | Visible |
| Conteneur | Non |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Un composant "Label" permet d'inclure facilement du texte sur une fiche. Ce texte n'est pas éditable par l'utilisateur et c'est donc un composant que l'on utilise souvent comme étiquette pour d'autres contrôles. Bien que l'on puisse modifier la police utilisée pour l'écriture du texte, il faut toujours freiner ses ardeurs. Ces composants sont en général en grand nombre sur les fiches importantes, mais cela ne pose pas de problème car ils ne prennent presque pas de mémoire.

***Note : (aux programmeurs avertis) :***
*Les composants "Label" ne sont pas des vrais objets au sens de Windows : ils ne possèdent pas de Handle. Ces composants sont en effet directement dessinés sur le canevas de la fiche. Pour utiliser un Label avec un Handle (utile avec ActiveX), il faut utiliser le composant "StaticText".*

Un composant "Label" peut contenir jusqu'à 255 caractères, ce qui le limite à des textes très courts. Les propriétés "AutoSize" et "WordWrap" permettent d'obtenir une bande de texte à largeur fixe sur plusieurs lignes, ce qui sert souvent pour donner des descriptions plus étoffées que de simples étiquettes. Les composants "Label" sont très souvent utilisés et le seront donc dans les manipulations futures.

**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Alignment | Permet d'aligner le texte à droite, au centre ou à gauche. N'est utile que lorsque "AutoSize" est faux et qu'une taille différente de celle proposée a été choisie. Le texte s'aligne alors correctement. |
| AutoSize | Active ou désactive le redimensionnement automatique du Label. "true" ne permet qu'une seule ligne, avec une taille ajustée au texte présent dans le label, tandis que "false" permet plusieurs lignes, non ajustées au contenu du label. |
| Caption | Permet de spécifier le texte à afficher dans le label. |
| FocusControl | Référence à un autre contrôle. Cette propriété permet de choisir le composant qui reçoit le focus (qui est activé) lorsqu'on clique sur le label. Cette propriété permet un certain confort à l'utilisateur puisqu'un clic sur une étiquette pourra par exemple activer le composant étiquetté. |



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Autorise les retours à la ligne pour permettre d'afficher plusieurs lignes à l'intérieur du label. "AutoSize" doit être faux pour permettre l'utilisation de plusieurs lignes. |

## X-B-4 - Composant "Edit"

**Fiche technique :**

| Icône | ab |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Les composants "Edit" permettent de proposer des zones d'édition. Ces zones très souvent utilisées sous Windows ne contiennent qu'une ligne de texte, dont la police peut être règlée, toujours avec la même parcimonie. Ce composant permet à l'utilisateur d'entrer une information quelconque tapée au clavier. Le texte entré dans le composant est accessible via une propriété "Text". Il est possible de fixer une limite à la longueur du texte entré, de masquer les caractères (utile pour les mots de passe), de désactiver la zone ou d'interdire toute modification du texte.

**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| AutoSelect | Permet l'autosélection du contenu lors de l'activation : lorsque le contrôle devient actif, le texte est sélectionné, de sorte qu'il peut directement être modifié en tapant un nouveau texte. Utiliser cette fonction dans les formulaires peut faire gagner un temps précieux. |
| Enabled | Active ou désactive la zone d'édition (l'édition n'est possible que lorsque la zone est activée). |
| MaxLength | Permet de fixer le nombre maximal de caractères entrés dans la zone. Mettre 0 pour ne pas donner de limite (par défaut). |
| PasswordChar | A utiliser lorsqu'on veut masquer les caractères tapés, comme pour les mots de passe. Utiliser le caractère "*" pour masquer (le plus souvent utilisé) et "#0" (caractère n°0) pour ne pas masquer. |
| ReadOnly | Permet d'activer la lecture seule de la zone d'édition. Lorsque |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Contient le texte entré dans la zone d'édition. C'est aussi en changeant cette propriété que l'on fixe le contenu de la zone. |

**Evénements :**

| | |
|---|---|
| OnChange | Se produit lorsqu'un changement de texte est susceptible de s'être produit, c'est-à-dire lorsque le texte s'est effectivement produit, mais aussi dans certaines autres circonstances. La propriété "Modified" de type booléen permet de tester depuis la procédure de réponse aux événements si une modification du texte a eu lieu. Lors de l'exécution de la procédure associée à cet événement, la propriété "Text" est déjà modifiée et vous pouvez connaître le nouveau contenu en consultant cette propriété. Conseil : veillez à ne |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | pas écrire de code trop long à exécuter dans la procédure de réponse à cet événement, car il se produit assez souvent lorsqu'un utilisateur remplit un formulaire constitué de quelques zones d'édition par exemple. |

Une manipulation fera intervenir ce composant à la fin du **§ X-B-7 Composant "CheckBox"**.

## X-B-5 - Composant "Memo"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Le composant "Memo" permet l'édition de texte sur plusieurs lignes. Une utilisation célèbre de ce composant est faite par le bloc-notes de Windows. Ce composant ne traite pas la mise en forme des caractères comme dans WordPad (pour cela, un autre composant, "RichEdit", est tout indiqué). Le composant "Memo", placé sur une fiche, permet donc lors de l'exécution, d'écrire du texte, d'ajouter des lignes en appuyant sur Entrée, d'éditer facilement ce texte (les fonctionnalités de copier-coller ne sont cependant pas automatiques et vous devrez apprendre à les programmer, j'essaierai de traiter un exemple d'utilisation du presse-papier dans un futur chapitre, il est encore un peu tôt pour cela). Concrètement, un mémo stocke le texte sous formes de lignes : chaque ligne est une chaîne de caractères. La propriété objet "Lines" des composants "Memo" permet l'accès aux lignes du mémo et leur manipulation. L'interface de Delphi permet même d'écrire directement dans le mémo en éditant la propriété "Lines".

**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Align | Permet d'aligner le mémo à gauche, droite, en haut, en bas, ou en prenant toute la place du composant qui le contient (la fiche, ou un autre composant conteneur). Ceci permet au mémo de mettre à jour automatiquement certaines de ses dimensions lorsque c'est nécessaire. |
| Lines | Pendant l'exécution de l'application, permet d'accèder aux lignes du mémo, c'est-à-dire à tout le texte qui y est écrit. Lines possède une propriété tableau par défaut qui permet de référencer les lignes par Lines[x] ou x est le n° de ligne. Lines possède également une propriété Count qui donne le nombre de lignes. Les lignes sont numérotées de 0 à Count-1. |
| ReadOnly | Permet d'interdire la modification du texte du mémo (en fixant la valeur à True). |
| WantTabs | Autorise l'utilisateur à utiliser des |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Permet les retours à la ligne automatique comme dans le bloc-note de Windows : lorsque WordWrap vaut True, les lignes trop longues sont découpées en plusieurs de façon à ce que la largeur du texte colle avec celle du mémo. Ceci ne modifie en rien les lignes du mémo, car c'est une option visuelle seulement. Par défaut, WordWrap vaut False et les lignes trop longues sont non découpées. |

**Evénements :**

| | |
|---|---|
| OnChange | Analogue à l'événement OnChange des composant Edit. |

Une manipulation fera intervenir ce composant à la fin du § **X-B-7 Composant "CheckBox"**.

## X-B-6 - Composant "Button"

**Fiche technique :**

| | |
|---|---|
| Icône | |
| Visibilité | Visible |
| Conteneur | Non |

Pas vraiment besoin de présenter longuement ici le composant Button : vous vous en servez depuis un bon moment si vous suivez ce guide depuis le début. Ce composant sert en général à proposer à l'utilisateur une action. Cette


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

action lui est expliquée par un texte très court sur le bouton, du style "OK" ou "Annuler". Lorsque l'utilisateur clique sur le bouton, ou appuie sur Espace ou Entree lorsque celui-ci est sélectionné (ce qui revient à le cliquer), l'événement OnClick se produit, qui offre une possibilité de réaction. C'est en général dans la procédure de réponse à cet événement qu'on effectue l'action proposée à l'utilisateur, comme afficher ou fermer une fiche par exemple pour citer des exemples récents.
Les boutons ont une fonctionnalité en rapport avec les fenêtres modales : ils ont une propriété ModalResult. Cette propriété ne fonctionne pas du tout comme celle des fiches. Elle est constante (fixée par vous), et est recopiée dans la propriété ModalResult de la fiche lors d'un clic sur le bouton. Ceci a comme effet de pouvoir créer un bouton "OK" rapidement en lui donnant "mrOK" comme ModalResult. Lors d'un clic sur ce bouton, la propriété ModalResult de la fiche devient mrOK et la fiche se ferme donc, sans aucune ligne de code source écrite par nous. Nous utiliserons cette possibilité dans de futurs exemples.

**Propriétés :**



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Cancel | Lorsque Cancel vaut True, un appui sur la touche Echap a le même effet qu'un clic sur le bouton (lorsque la fiche est active). Cette fonction réservée au bouton "Annuler" permet d'utiliser le raccourci clavier Echap pour sortir de la fiche. |
| Caption | Texte du bouton. Une seule ligne est autorisée. Si le texte est trop long, seule une partie est visible sur le bouton. |
| Default | Analogue de Cancel mais avec la touche Entrée. Cette fonction est en général réservée au bouton "OK" de certaines fiches très simples que la touche Entrée suffit alors à refermer (pensez tout simplement aux messages affichés par ShowMessage. Même si l'on utilise pas explicitement de fiche, c'est bel et bien une fiche qui est employée avec un bouton "OK" avec sa propriété "Default" fixée à True). |
| Enabled | Comme pour |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Permet de modifier automatiquement la propriété ModalResult de la fiche contenant le bouton lors d'un clic et si la fiche est modale. La gestion de l'événement OnClick devient parfois inutile grâce à cette propriété. |

**Evénements :**

| | |
|---|---|
| OnClick | Permet de répondre au clic sur le bouton. Cet événement se produit aussi lorsque le bouton est actif et que la touche Entrée ou Espace est enfoncée, ou lorsque la touche Echap est enfoncée et que la propriété Cancel vaut True, ou lorsque la touche Entrée est enfoncée et que la propriété Default vaut True. L'appel de la méthode Click déclenche aussi cet événement. |

**Méthodes :**

| | |
|---|---|
| Click | La méthode Click copie |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | la propriété ModalResult du bouton dans celle de sa fiche, et déclenche un événement OnClick. C'est la méthode à appeler lorsqu'on doit déclencher l'événement OnClick depuis le code source. |

Une manipulation fera intervenir ce composant à la fin du § **X-B-7 Composant "CheckBox"**.

## X-B-7 - Composant "CheckBox"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Un composant CheckBox permet de donner à l'utilisateur un choix de type "Oui/Non". S'il coche la case, la réponse est "Oui", sinon, c'est "Non". Au niveau du code, une propriété de type booléen stocke cette réponse : Checked. Il est possible de donner un texte explicatif de la case à cocher dans la propriété Caption. Ce texte apparait à coté de la case. Une modification de l'état de la case déclenche un événement OnClick, que cette modification provienne effectivement d'un clic ou d'une pression sur la touche Espace. La modification depuis le code source de la propriété Checked ne déclenche pas cet événement.

**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Caption | Permet de spécifier le texte qui apparait à coté de la case à cocher. Il est à noter que le composant ne se redimensionne pas pour afficher automatiquement tout le texte. Ce sera donc à vous de prévoir cela si c'est nécessaire. |
| Checked | Permet de connaître ou de modifier l'état de la case : cochée ou pas. |

**Evénements :**

| | |
|---|---|
| OnClick | Déclenché lorsque l'état de la case à cocher change, quelle que soit l'origine du changement. La propriété Checked est déjà mise à jour lorsque la procédure de réponse à cet événement s'exécute. |

## X-B-7-a - Manipulation guidée

L'objectif de cette manipulation est de vous faire manipuler les composants Edit, Memo, Button et CheckBox. Placez un composant de chaque sorte sur la seule fiche d'un projet vierge, et obtenez l'interface ci-dessous en modifiant les propriétés nécessaires décrites dans les paragraphes précédents (pour effacer le texte présent dans le mémo, éditez la propriété Lines et effacez le texte présent dans l'éditeur). Les noms des composants ont été rajoutés en rouge (ces annotations ne doivent pas apparaître dans votre interface !) :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Le but de cette petite application est assez simple : le contenu de la zone d'édition est ajouté au mémo lorsqu'on clique sur le bouton "Ajout". Cet ajout se fait sur une nouvelle ligne quand l'option correspondante est cochée, et sur la dernière ligne sinon. Si le mémo ne contient pas encore de texte, une première ligne est créée. Cet exemple va nous permettre d'utiliser de nombreuses propriétés abordées dans les paragraphes ci-dessus.
L'interface de l'application est déjà faite, il ne nous reste plus qu'à écrire un peu de code. Le seul "moment" où il se passe quelque chose dans notre application est lorsqu'on clique sur le bouton : c'est à ce moment que l'ajout a lieu. Générez donc la procédure de réponse à l'événement OnClick de ce bouton.
Il y a différents cas à traiter. Un premier cas simple est lorsqu'une nouvelle ligne est à ajouter. L'autre cas, qui consiste à ajouter à la fin de la dernière ligne, dépend de l'existence de cette dernière ligne. Il faudra donc encore distinguer deux cas, ce qui en fait en tout trois. Pour distinguer ces cas, un bloc "if" est tout indiqué (c'est même notre seul choix possible !). Le cas où l'on ajoute une ligne est réalisé lorsque la case est cochée, c'est-à-dire lorsque la propriété Checked de cbNouvLigne est vraie. Voici alors le début du bloc "if" :

```
if cbNouvLigne.Checked then
  ...
```

Le deuxième cas (pas de dernière ligne, il faut donc en créer une) est réalisé lorsque le nombre de lignes du mémo vaut 0. Le nombre de lignes du mémo est indiqué par la propriété Count de la propriété objet Lines de meAjout. Voici la suite de notre boucle "if" :

```
if cbNouvLigne.Checked then
  ...
else if meAjout.Lines.Count = 0 then
  ...
```

Notez bien qu'on a imbriqué un second bloc "if" dans le premier pour traiter 3 cas et non pas seulement 2. La dernière condition est simplement réalisée quand aucune des deux autres ne l'est, c'est donc un simple else. Voici le squelette complet de notre structure "if" :

```
if cbNouvLigne.Checked then
  ...
else if meAjout.Lines.Count = 0 then
  ...
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
else
  ...
```

Etudions maintenant le premier cas. Ce cas ajoute simplement le texte de la zone d'édition dans une nouvelle ligne. Le texte de la zone d'édition est la propriété Text de edTxtAjout. Pour ajouter une ligne à meAjout, il faut appeler la méthode Add de sa propriété objet Lines. On doit transmettre une chaîne à ajouter. La chaîne transmise sera tout simplement le texte de la zone d'édition, de sorte que l'instruction sera :

```
meAjout.Lines.Add(edTxtAjout.Text)
```

L'action à effectuer dans le second cas est en fait identique à celle à effectuer dans le premier cas (ajouter le texte de la zone d'édition dans une nouvelle ligne). Plutôt que de réécrire l'instruction, regroupons plutôt les deux premiers cas du bloc "if". On peut faire cela avec un OU logique : l'instruction sera exécutée lorsque l'une des deux conditions sera réalisée. Voici ce que cela donne actuellement :

```
if cbNouvLigne.Checked or (meAjout.Lines.Count = 0) then
  meAjout.Lines.Add(edTxtAjout.Text)
else
  ...
```

Vous voyez qu'on a mis une des conditions entre parenthèses. la raison en est que sans les parenthèses, la ligne aurait été mal interprétée : le or aurait été considéré entre cbNouvLigne.Checked et meAjout.Lines.Count, ce qui est interdit. Les parenthèses permettent d'isoler des morceaux qui doivent être évaluées avant d'être intégrés au reste du calcul.
Le traitement du dernier cas est plus délicat. On doit récupérer la dernière ligne (on sait qu'elle existe dans ce cas), ajouter le texte de edTxtAjout, puis réaffecter le tout à la dernière ligne, ou alors tout faire en une seule opération. Je vous propose de voir d'abord la méthode facile et ensuite la moins facile.
Pour obtenir une ligne, on utilise une propriété tableau par défaut de la propriété objet Lines de meAjout. Vous n'avez pas besoin de connaître le nom de cette propriété, puisqu'il suffit de faire suivre "Lines" de crochets sans le mentionner. Le numéro de la ligne s'obtient par le nombre de lignes. La dernière ligne est le nombre de ligne diminué de 1. La dernière ligne du mémo est donc :

```
meAjout.Lines[meAjout.Lines.Count - 1]
```

La première étape est de concaténer le texte de edTxtAjout à cette ligne, et de stocker le tout dans une variable temporaire "temp" de type string. La seconde étape consiste à affecter cette chaîne temporaire à la dernière ligne. Voici donc les deux instructions :

```
temp := meAjout.Lines[meAjout.Lines.Count - 1] + edTxtAjout.Text;
meAjout.Lines[meAjout.Lines.Count - 1] := temp;
```

Ceci constitue la solution la plus simple, qui exige une variable temporaire. Nous la garderons car elle est plus claire comme cela, mais comme je vous l'ai déjà fait remarquer, l'emploi d'une variable ne se justifie pas lorsqu'elle n'est utilisée qu'une fois en tant que valeur de paramètre. Voici donc la version "moins facile", sans variable temporaire, et tenant en une seule instruction :

```
meAjout.Lines[meAjout.Lines.Count - 1] := { suite en dessous }
   meAjout.Lines[meAjout.Lines.Count - 1] + edTxtAjout.Text;
```

Ci-dessus, la chaîne affectée à temp est directement affectée à la dernière ligne. Voici donc le texte complet de la procédure que nous avions à écrire :

```
procedure TfmPrinc.btAjoutClick(Sender: TObject);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
var
  temp: string;
begin
  if cbNouvLigne.Checked or (meAjout.Lines.Count = 0) then
    meAjout.Lines.Add(edTxtAjout.Text)
  else
    begin
      temp := meAjout.Lines[meAjout.Lines.Count - 1] + edTxtAjout.Text;
      meAjout.Lines[meAjout.Lines.Count - 1] := temp;
    end;
end;
```

Je vous encourage enfin à tester votre application. Il va sans dire que vous pouvez entrer ou effacer du texte à votre guise dans le mémo. Je ne propose pas le téléchargement de ce projet assez simple. Si vous veulez malgré tout le code source, **contactez-moi**.

## X-B-8 - Composant "RadioButton"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Le composant "RadioButton" permet de proposer un choix à l'utilisateur parmi plusieurs possibilités. Chaque possibilité est constituée d'un composant "RadioButton". A l'intérieur d'un composant conteneur ou d'une fiche, l'utilisateur ne peut cocher qu'un seul composant "RadioButton", quel que soit le nombre proposé. Pour créer des groupes de composants séparés, il faut les regrouper dans des contrôles conteneurs comme les "Panel" ou les "GroupBox". Pour savoir quel RadioButton est coché, il faut passer en revue leur propriété Checked.
En ce qui concerne les propriétés et événemnts, un "RadioButton" est assez proche d'un "CheckBox". Voici, comme d'habitude, un résumé des éléments à connaître.

**Propriétés :**

| | |
|---|---|
| Caption | Permet de spécifier le texte qui apparait à coté de la case d'option. Il est à noter que le composant ne se redimensionne pas pour afficher automatiquement tout le texte. Ce sera donc à vous de prévoir cela si c'est nécessaire. |
| Checked | Permet de connaître ou de modifier l'état de la |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | case : cochée ou pas. |
|---|---|

**Evénements :**

| OnClick | Déclenché lorsque l'état de la case d'option change, quelle que soit l'origine du changement. La propriété Checked est déjà mise à jour lorsque la procédure de réponse à cet événement s'exécute. |
|---|---|

Nous aurons l'occasion d'utiliser ce composant dans de futurs exemples.

## X-B-9 - Composant "ListBox"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Ce composant, que vous connaissez pour l'avoir essayé au cours d'un exemple dans ce guide, permet d'afficher une liste d'éléments, parmi lesquels l'utilisateur peut choisir un ou plusieurs éléments (plusieurs choix sont possibles à ce niveau). Chaque élément est présent sur une ligne, un peu à la façon d'un mémo, mais il n'est pas question ici d'éditer les éléments, juste d'en sélectionner. Le composant ListBox n'est pas limité à proposer des éléments à l'utilisateur : il peut aussi servir à afficher une liste des résultats d'une recherche par exemple.
Au niveau du fonctionnement, la propriété objet Items permet la gestion des éléments. Cette propriété dispose d'une propriété tableau par défaut pour accèder aux éléments. Ces éléments sont de type chaîne de caractère. Items dispose en outre d'une méthode "Clear" pour vider la liste, d'une méthode "Add" pour ajouter un élément, et d'une propriété "Count" pour indiquer le nombre d'élément. Les éléments sont numérotés de 0 à Count - 1. Outre la propriété Items, le composant ListBox dispose d'un certain nombre de propriétés permettant de configurer le composant.
**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| ExtendedSelect | Lorsque MultiSelect vaut True, ExtendedSelect permet une sélection d'éléments non à la suite les uns des autres. Lorsque ExtendedSelect vaut False, seule une plage contigüe d'éléments peut être sélectionnée : d'un élément de départ à un élément de fin. Dans le cas contraire, n'importe quel élément peut être sélectionné ou non. |
| IntegralHeight | Permet ou interdit l'affichage d'éléments coupés, c'est-à-dire d'éléments partiellement visibles. Lorsque IntegralHeight vaut False, ces éléments sont affichés. Dans le cas contraire, la hauteur de la zone de liste est ajustée pour afficher un nombre entier d'éléments. |
| ItemHeight | Permet de fixer la hauteur d'un élément. Cette valeur est calculée automatiquement lorsque vous changez la police utilisée par la zone d'édition. Vous pouvez |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Fixe le style de la zone de liste. Pour l'instant, laissez la valeur de Style à lbStandard. Plus tard, si vous le souhaitez, vous pourrez en utilisant les autres valeurs dessiner vous-mêmes les éléments de la zone et créer ainsi des listes personnalisées visuellement. |

**Evénements :**

| | |
|---|---|
| OnClick | Se produit lorsqu'un clic se produit dans la liste. |

## X-B-10 - Composant "ComboBox"

**Fiche technique :**

| | |
|---|---|
| Icône | |
| Visibilité | Visible |
| Conteneur | Non |

Le composant "ComboBox" (zone de liste déroulante) permet également la gestion de liste d'éléments, mais à d'autres fins et avec d'autres possibilités. En premier lieu, un composant "ComboBox" ne permet la sélection que d'un élément au plus. Le composant affiche une zone montrant selon les cas l'élément sélectionné ou une zone d'édition (analogue à un "Edit"). Il est possible d'afficher une liste des éléments de la liste en cliquant sur un bouton prévu à cet effet. Il est alors possible de sélectionner un élément dans la liste.
Au niveau du code, ue composant ComboBox fonctionne comme un composant ListBox pour ce qui est de la propriété Items. Un composant ComboBox pouvant contenir une zone d'édition, il dispose de certaines éléments de ces dernières, comme les propriétés Length et Text, et l'événement OnChange. La propriété Style permet de choisir parmi 5 styles différents dont 3 seront décrits ici, les 2 autres autorisant la création d'éléments personnalisés visuellement, ce qui déborde largement du cadre de ce chapitre.
**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| MaxLength | Lorsque le composant comporte une zone d'édition, MaxLength fixe le nombre maximal de caractères entrés dans cette zone. |
| Sorted | Permet comme pour une ListBox d'activer le tri alphabétique des éléments. |
| Style | Détermine le style de la zone de liste déroulante. Voici la description de 3 des 5 valeurs proposées :<br><br>• csDropDown : crée une liste déroulante avec une zone d'édition. Seule cette dernière est visible et permet d'entrer un élément qui n'est pas dans la liste. Le fait de taper du texte dans la zone d'édition déselectionne tout élément sélectionné.<br>• csDropDownList : crée une liste déroulante sans zone d'édition. Le composant montre |



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Contient le texte de la zone d'édition. Attention, n'utilisez pas cette propriété si la propriété Style est fixée à csDropDownList car le texte est alors imprévisible (je dois avouer mon ignorance à ce sujet). |

**Evénements :**

| | |
|---|---|
| OnChange | Lorsque le composant comporte une zone d'édition, celle-ci déclenche des événements OnChange comme une zone d'édition normale. Ces événements vous permettent d'exécuter du code répondant à tout changement. |
| OnClick | Déclenché lors d'un clic sur toute partie du composant, que ce soit la liste déroulante ou la zone d'édition. |

## X-B-11 - Composant "GroupBox"

**Fiche technique :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Oui |

Les composant "GroupBox" ont pour but d'effectuer des regroupements visibles de composants. En effet, ils se présentent sous la forme d'un cadre 3D et d'un texte en haut à gauche. Un composant GroupBox est un conteneur, donc susceptible de contenir d'autres composants (conteneurs ou pas, soit dit en passant). Lorsque l'on veut proposer à l'utilisateur le choix parmi plusieurs options, un composant GroupBox contenant des composants RadioButton est un bon choix. Vous trouverez facilement des exemples en vous promenant dans l'interface de vos logiciels favoris. Si vous comptez vous limiter à des RadioButton placés sur un GroupBox, intéressez-vous au composant RadioGroup non décrit ici mais qui permet de créer rapidement ce genre d'interface.
Au niveau du code, un composant GroupBox n'a pas souvent de rôle particulier à jouer, étant donné que c'est surtout son caractère conteneur qui importe. Aucune propriété, méthode ou événement n'est à signaler ici, cependant, il ne faut pas sous-estimer ce composant qui tient un grand rôle dans l'organisation tant visuelle que logique des interfaces les plus fournies.

## X-B-12 - Composant "Panel"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Oui |

Les composants "Panel", ou panneaux, sont fréquement employés sous Delphi. Ils ont deux rôles connus, que l'on peut parfois combiner. L'intérêt de ces panneaux, c'est que ce sont des conteneurs, qu'ils possèdent une bordure 3D configurable à volonté, et qu'ils sont alignables. Cette dernière caractéristique, si elle n'évoque probablement pas grand chose pour vous, sera bientôt une aide précieuse dans la conception de vos interfaces. Les panneaux sont donc utilisés soit pour leurs bordures 3D, soit pour créer des alignements.
Au niveau du code, les panneaux sont assez peu manipulés, sauf lorsqu'on a besoin de règler leurs dimensions pour maintenir l'interface d'une application. A la conception, par contre, il est possible de manipuler quelques propriétés agissant sur l'alignement et l'apparence des panneaux.

**Propriétés :**


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| Align | Décide de l'alignement du panneau :<br><br>• alNone : aucun alignement.<br>• alLeft : alignement sur la gauche du conteneur. Seule la bordure droite du panneau est libre.<br>• alRight : alignement sur la droite du conteneur. Seule la bordure gauche du panneau est libre.<br>• alTop : alignement en haut du conteneur. Seule la bordure inférieure du panneau est libre.<br>• alBottom : alignement en haut du conteneur. Seule la bordure supérieure du panneau est libre.<br>• alClient : alignement sur toute la partie non recouverte par d'autres composants alignés. Aucune bordure n'est |
|---|---|


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Spécifie le texte qui apparaît à l'intérieur du panneau. Lorsque le panneau est utilisé pour créer des alignements, il faut penser à effacer le contenu de Caption. |

Les possibilités d'alignement des panneaux (propriété Align) seront utilisées dans de futurs exemples.

## X-B-13 - Composant "Bevel"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Visible |
| Conteneur | Non |

Le composant Bevel a un but simple : il est décoratif. En effet, ce composant ne sert qu'à créer des effets visuels à l'aide de ses quelques propriétés permettant de créer des bordures 3D. Vous pouvez ainsi créer un cadre, ou seulement l'un des cotés du cadre, et changer l'effet graphique du composant. Ce composant, moins performnt que Panel dans le sens ou il n'est pas un conteneur et offre un peu moins de possibilités de bordures, permet cependant de se passer d'un Panel de temps en temps.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| Align | Permet, comme pour les panneaux et d'autres composants, d'aligner le Bevel sur un des bords de son conteneur, voire sur tout le conteneur pour lui adjoindre une bordure si ce dernier n'en possède pas. |
| Shape | Détermine l'effet visuel créé par le Bevel (dépendant également de la propriété Style :<br><br>• bsBox : affiche le contenu du Bevel comme relevé ou enfoncé.<br>• bsFrame : affiche un cadre relevé ou enfoncé autour du contenu.<br>• bsTopLine : affiche uniquement la ligne supérieure du cadre.<br>• bsBottomLine : affiche uniquement la ligne inférieure du cadre.<br>• bsLeftLine : affiche uniquement la ligne gauche du cadre.<br>• bsRightLine : affiche uniquement la ligne droite du |


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

| | |
|---|---|
| | Fixe le style de bordure du cadre : bsLowered (enfoncé) ou bsRaised (relevé). |

## X-B-14 - Composant "ImageList"

**Fiche technique :**

| Icône | |
|---|---|
| Visibilité | Invisible |
| Conteneur | Non |

Le composant "ImageList" est non visuel. Il permet de stocker une liste d'images de mêmes dimensions. Le composant "ImageList" est ensuite utilisable par d'autres composants pour leur permettre d'accèder directement à la liste d'images. Le principe est d'assigner le composant à une propriété de type référence ImageList. Le composant qui utilise un "ImageList" permet ensuite d'accèder aux images en utilisant directement leur index. Les images sont indexées de 0 à Count - 1 où Count est une propriété du composant ImageList.
Pour créer cette liste d'images, placez un composant ImageList sur une fiche, puis double-cliquez dessus (il n'y a pas de propriété à éditer). vous vous retrouvez dans un éditeur spécialisé qui vous permet d'ajouter des images. Veillez, lorsque vous ajoutez une image (format bmp ou ico), à ce qu'elle soit de la bonne dimension. Fixez également une couleur transparente si une partie de l'image doit être transparente (utile pour créer des barres d'outils).
L'utilisation des composants ImageList est démontrée dans **l'Annexe 8-A : Création de menus**.
Ceci termine ce chapitre qui n'est qu'une première partie. La deuxième partie, présentée plus tard, sera consacrée à l'étude de composants plus délicats ou complexes à manipuler. Ces composants demandent pour être utilisés des connaissances que vous ne possèdez pas encore.

## X-C - Mini-projet

Je tiens absolument à terminer ce chapitre par un mini-projet : le premier mini-projet dont vous pourrez réaliser entièrement l'interface et le code source, en répondant à des exigences définies. La nature de ce projet dérangera ceux d'entre vous qui se sont jurés d'oublier les mathématiques, car il est en plein dans ce domaine.
**Accèder au mini-projet** (attention à la tentation : les solutions sont données juste à la fin de l'énoncé).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

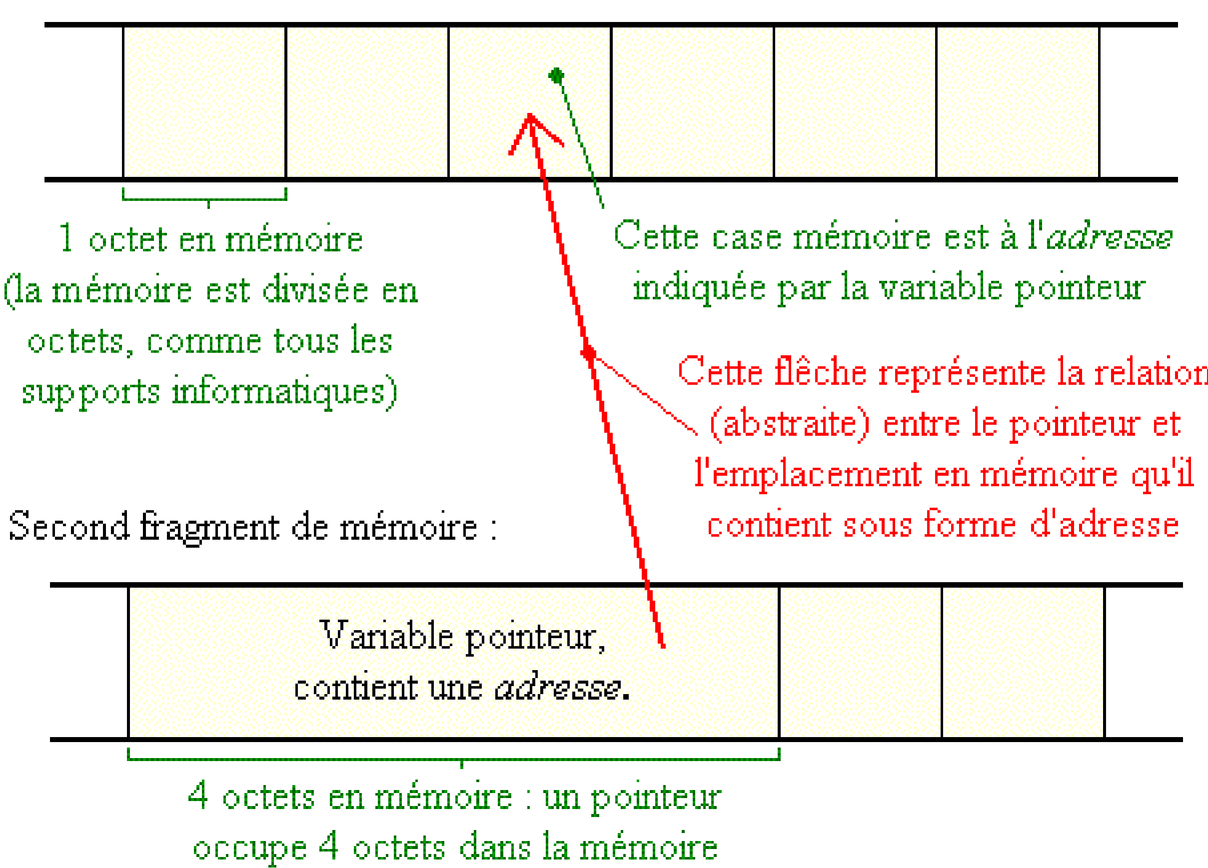
# XI - Pointeurs

Nous passons aux choses sérieuses avec ce chapitre dédié à un sujet souvent décrit comme complexe mais indispensable pour tout programmeur digne de ce nom. Les pointeurs sont en effet constamment utilisés dans les projets les plus importants car ils apportent des possibilités énormes et très diverses. Sans ce concept de pointeurs, de nombreux problèmes ne pourraient pas être transcrits et résolus sur ordinateur.
Ce chapitre commencera par vous expliquer ce que sont les pointeurs, puis diverses utilisations des pointeurs seront passées en revue. Nous parlerons entre autres des pointeurs de record et de transtypage dans ce chapitre.
Si vous êtes habitué au rythme de progression du guide, vous vous sentirez probablement un peu secoué(e) dans ce chapitre, car la notion de pointeur est assez délicate et demande de gros efforts d'apprentissage et d'assimilation au début. Dés que vous aurez "compris le truc", ce que je m'engage à faciliter, tout ira mieux.

## XI-A - Présentation de la notion de pointeur

La notion de base de pointeur est assez simple : il s'agit d'une variable qui stocke l'emplacement en mémoire d'un autre élément (qui est le plus souvent une variable). Cette première variable est dite de type pointeur, et est tout simplement nommée "pointeur". Le type Pascal Objet utilisé pour le type pointeur est tout simplement : pointer. Lorsqu'on voit cela pour la première fois, on peut douter de l'intérêt d'un tel assemblage, mais rassurez-vous si c'est votre cas, vous changerez très vite d'avis. L'autre élément dont nous parlions est de type indéterminé, ce qui nous apportera quelques petits soucis, vite résolus au **§XI-A-2**.
Le contenu de cette variable "pointeur" importe en fait bien peu. Ce qui importe, c'est l'élément "pointé" par la variable "pointeur". Une variable de type pointeur, ou plus simplement un pointeur, contient une adresse. Une adresse est tout simplement un nombre qui repère un emplacement en mémoire (la mémoire des ordinateurs est découpée en octets. Ces octets sont numérotés, et une adresse est tout simplement un numéro d'octet). Mais ne vous souciez pas trop de ces détails techniques, sauf si vous comptez approfondir le sujet (ce qui ne se fera pas ici). Voici un schémas illustrant la notion de pointeur et expliquant deux ou trois autres choses sur la mémoire.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Si vous êtes curieux de savoir ce qu'est exactement une adresse, lisez le contenu de l'encadré ci-dessous, sinon, passez directement à la suite.

***Tout sur les adresses mémoire :***
*Une adresse mémoire consiste en fait en un nombre. La convention adoptée partout est de noter ce nombre en hexadécimal. Etant donné que la mémoire des ordinateurs est aujourd'hui limitée à 4 gigaoctets, on a besoin d'un nombre entre 0 et 2^32-1, soit entre 0 et FFFFFFFF en hexadécimal.*
*Si vous ne connaissez pas l'hexadécimal, sachez que c'est un système de numérotation à 16 chiffres au lieu de 10 : les chiffres sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F au lieu des 10 chiffres traditionnels de 0 à 9 ; un nombre hexadécimal tel que 1F vaut 1 x 16 + 15 (valeur de F) = 31 en notation décimale classique.*
*En Pascal Objet, les nombres hexadécimaux sont notés avec le préfixe $. Le nombre 1F sera écrit : $1F.*
*Un tel nombre est stockable sur 4 octets (256 x 4 = 2^32), ce qui explique la taille d'une variable pointeur, qui contient ce nombre. Ce nombre hexadécimal est nommé adresse dans le cadre de l'utilisation avec les pointeurs. Une adresse permet donc de repèrer un octet quelconque dans la mémoire par son numéro.*
*Petite précision enfin : lorsque l'élément pointé fait plus d'un seul octet, ce qui est évidemment très souvent le cas, c'est l'octet de début (le premier de la suite d'octets qui stockent l'élément) qui est pointé.*

## XI-A-1 - nil

Le titre de ce paragraphe vous a certainement un peu étonné... non, il ne s'agit pas du fleuve, mais bien encore des pointeurs !


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Vous savez maintenant ce qu'est un pointeur : un élément qui stocke l'emplacement d'un autre élément. Parfois, pourtant, un pointeur ne pointera rien du tout (aucun emplacement en mémoire, donc rien). Ce cas particulier de pointeur, dont vous découvrirez rapidement l'extrème utilité, est appelé pointeur nul, ou plus simplement *nil. nil* est en fait la valeur d'un pointeur qui ne pointe rien. Voici un extrait de code (il est inscrit, comme d'habitude, dans une procédure répondant à l'événement OnClick d'un bouton) :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
var
  Machin: pointer;
begin
  Machin := nil;
end;
```

Ce code, qui ne fait pas grand chose, illustre plusieurs choses. En tout premier, on a déclaré une variable de type pointeur nommée "Machin" (vous voudrez bien excuser mon manque d'inspiration). Cette variable est désormais une variable comme les autres. L'unique instruction de la procédure est une affectation. On affecte *nil* à la variable "Machin" (Vous remarquerez en passant que nil est un mot réservé du langage Pascal Objet). Ceci prouve que nil est une valeur valable pour un pointeur, puisqu'on l'affecte à une variable de type pointeur. Après cette affectation, "Machin" est un pointeur nul, car sa valeur est nil : "Machin" ne pointe nulle part.
Ce sera désormais une préoccupation à avoir en tête lorsque vous manipulerez un pointeur : avant de vous soucier de l'élément pointé par le pointeur, il vous faudra savoir si le pointeur pointe effectivement quelque chose. Le moyen pour savoir cela sera de comparer le pointeur à nil. Si la comparaison donne "True", c'est que le pointeur est nul, donc ne donne rien, et si elle donne "False", c'est que le pointeur pointe quelque chose.

## XI-A-2 - Eléments pointés par un pointeur

Nous nous sommes surtout attardé pour l'instant sur les pointeurs et leur contenu. Il est temps de nous occuper un peu des éléments pointés par les pointeurs, qui sont tout de même importants.
Le type pointer utilisé jusqu'à présent a un inconvénient majeur : il déclare en fait un pointeur vers *n'importe quoi* (une donnée de n'importe quel *type*, ce type étant tout à fait inconnu, et c'est là le problème) : même si on peut accèder à l'information pointée par le pointeur, il est impossible de savoir de quel type est cette information, ce qui la rend inutilisable concrètement. Etant donné que lorsqu'on utilise un pointeur, c'est le plus souvent pour se servir de l'élément pointé et non du pointeur en lui-même, nous sommes en face d'un problème grave : si un élément est effectivement pointé par un pointeur, et qu'on ne sait pas si c'est une chaîne de caractères ou un entier, on va aux devants de gros embêtements. Il va nous falloir un moyen de savoir de quel type de données est l'information pointée.
La solution est trouvée en utilisant des types de pointeurs spécifiques aux informations pointées. On utilisera dans la réalité des "pointeurs vers des entiers" ou des "pointeurs vers des chaînes de caractères" plutôt que des "pointeurs vers n'importe quoi". Le principe pour se débarrasser de l'ambiguité sur l'élément pointé est en effet de dire tout simplement : ce type de pointeur pointera toujours vers le même type de donnée (entier, chaîne de caractères, ...).
Pour utiliser ces différents *types* de pointeurs spécifiques, nous utiliserons de vrais *types* de données Pascal Objet. Le type pointeur vers un autre type de donnée pourra facilement être défini et utilisé dans tout programme. Ces types de pointeurs ne souffreront pas de l'ambiguité sur le *type* des données pointées car le *type* de la donnée pointée est déterminé à l'avance par le type du pointeur.
Le type "pointeur vers *type*" s'écrit ^type en Pascal Objet. Il suffit de précèder le nom du type de donnée pointée par un accent circonflexe pour déclarer un pointeur vers ce type de donnée. Attention, le type en question ne peut PAS être un type composé du genre "array[1..100] of byte" : il faut que le type soit un identificateur. Pour résoudre ce problème, il faut passer par des déclarations de types intermédiaires. Cela sera vu au **§ XI-B-1** Par exemple, pour déclarer une variable de type "pointeur vers un integer", il faudra utiliser le type ^integer (integer est bien un identificateur). Pour vous en convaincre, substituez dans le code précédent le mot "pointer" par "^integer" :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
var
  Machin: ^integer;
begin
  Machin := nil;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Dans cet exemple, la variable "Machin" est toujours un pointeur, même si elle n'est pas déclarée avec le type "pointer". Elle est déclarée comme un pointeur vers un entier. Le seul type possible pour l'élément pointé par "Machin" sera donc par conséquent "integer".
Rassurez-vous, dans la pratique, on utilisera en fait assez peu cette forme de déclaration. On déclarera plutôt un nouveau nom de type (section **type**), et on déclarera ensuite les variables ou les paramètres en utilisant ce nouveau type. Voici ce que cela donnera pour l'exemple ci-dessus :

```
type
  PInteger = ^integer;
var
  Machin: PInteger;
begin
  Machin := nil;
end;
```

Cet exemple déclare un type nommé "PInteger" qui est le type "pointeur vers un entier". Toute variable ou paramètre de ce type sera un pointeur vers un entier. Ce type est utilisé pour déclarer la variable "Machin", ce qui la déclare comme pointeur vers un entier, comme dans l'extrait de code précédent.

*Remarque : notez la première lettre (P) du type PInteger : c'est une convention et un bon repère de nommer les types pointeurs en commençant par un P, contrairement aux autres types personnalisés qui sont commencés par la lettre T.*

## XI-A-3 - Gestion de l'élément pointé par un pointeur

Jusqu'ici, nous nous sommes surtout intéressé aux pointeurs, au type de l'élément pointé, mais pas encore à l'élément pointé lui-même. Cet élément, qui sera souvent d'un type courant tel qu'un entier, une chaîne de caractères, un énuméré ou même d'un type plus complexe comme un record (c.f. **§ XI-A-4** Pointeurs de record), a une particularité par rapport aux données classiquement stockées dans des variables : il est géré *dynamiquement* et *manuellement*, c'est-à-dire qu'au fil de l'éxécution du programme, il nous faudra nous occuper de tâches habituellement invisibles. Ces tâches sont assez déroutantes pour les débutants mais vous vous y habituerez vite, c'est juste une habitude à prendre.
L'élément pointé par un pointeur devra donc être géré manuellement. La gestion de l'élément pointé se divise en deux tâches simples : son *initialisation* et sa *destruction*.
Avant de pouvoir utiliser l'élément pointé, il faudra le *créer*. Cette étape d'initialisation est indispensable, car elle réserve en fait de la place en mémoire pour l'élément pointé par le pointeur, et fait pointer le pointeur vers cet espace fraichement réservé. Utiliser un pointeur vers un espace non réservé (non initialisé), c'est-à-dire un pointeur non *créé*, conduira inévitablement à un plantage de l'application, voire parfois de Windows (sans vouloir vous effrayer bien entendu, mais c'est la triste et dure vérité).
Une fois l'utilisation d'un pointeur terminée, il faudra penser à le *détruire*, c'est-à-dire libèrer l'espace réservé à l'élément pointé par le pointeur. Cette seconde étape, parfois oubliée, est indispensable lorsqu'on a initialisé le pointeur. Oubliez ce genre d'étape et votre application viendra s'ajouter à la liste des pollueurs : après leur passage, de la mémoire a disparu, et cela entraîne une dégradation des performances, voire un plantage de Windows au bout d'un certain temps.
Le schémas ci-dessous tente de résumer cette délicate notion de gestion de l'élément pointé. Prenez le temps de bien le lire :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

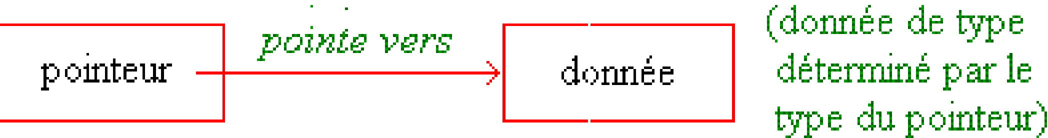
# Les 5 étapes de la gestion d'un pointeur :

**I** - pointeur déclaré vers un type déterminé :

(pas de donnée pointée car le pointeur n'a pas encore été "initialisé")

**II** - Initialisation du pointeur

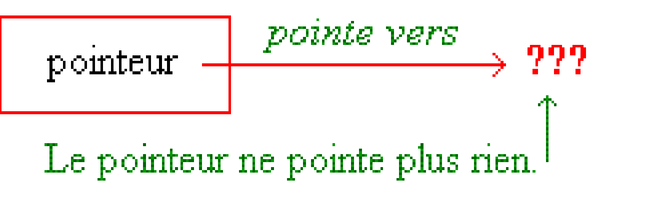
**III** - pointeur déclaré et initialisé vers une donnée valide :

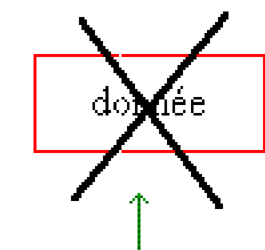



Pendant cette étape, l'élement pointé par le pointeur peut être utilisé.

**IV** - destruction du pointeur

**V** - pointeur détruit, plus de donnée pointée :




Le pointeur ne pointe plus rien.

La donnée pointée est détruite et inutilisable.

Retour à l'étape I

Dans ce schémas, les étapes 1, 3 et 5 sont des états du pointeur : les étapes 1 et 5 sont en fait identiques car dés lors que le pointeur est "détruit", il peut à nouveau être initialisé pour être réutilisé. Les étapes 2 et 4 vous sont dévolues. L'étape 2 est l'initialisation, qui précède l'utilisation. L'étape 4 est la destruction, qui suit l'utilisation. Toute utilisation de l'élément pointé est tout simplement impossible avant l'étape 2 et après l'étape 4 car cet élément pointé n'existe pas.

*Remarque : ce schéma ne s'applique qu'aux pointeurs vers des types déterminés créés par les types ^type et non pas au type pointeur général créé par le type pointer.*

Venons-en maintenant au code Pascal Objet proprement dit. L'initialisation d'un pointeur se fait au moyen de la procédure "New". Cette procédure admet un unique paramètre qui est le pointeur à initialiser. L'appel de cette procédure avec un pointeur en paramètre correspond à l'étape 2 du schémas ci-dessus. Comme initialisation et destruction vont de paire, voyons tout de suite la méthode de destruction : elle s'effectue par l'appel de la procédure "Dispose". Cette procédure, comme "New", prend un unique paramètre qui est le pointeur à "détruire". L'appel de "Dispose" avec un pointeur en paramètre correspond à l'étape 4 du schémas ci-dessus. "New" et "Dispose" ne peuvent initialiser et détruire que des pointeurs vers des types déterminés, et non des pointeurs de type pointer.
L'exemple ci-dessous utilise les nouvelles notions :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
type
  PInteger = ^integer;
var
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  Machin: PInteger;
begin
  new(Machin);
  { ici, on peut utiliser l'élément pointé par Machin }
  dispose(Machin);
end;
```

La procédure ci-dessus déclare "Machin" comme pointeur vers un entier. La première instruction initialise "Machin". Avant cette instruction, l'élément pointé par "Machin" n'existe même pas. Après l'appel à New, l'élément pointé par "Machin" est utilisable (nous allons voir comment juste après). La deuxième instruction détruit le pointeur : l'élément pointé est détruit et le pointeur ne pointe donc plus vers cet élément qui a été libéré de la mémoire.
Nous allons maintenant voir comment utiliser l'élément pointé par un pointeur.

### XI-A-4 - Utilisation de l'élément pointé par un pointeur

Une fois un pointeur initialisé, il est enfin possible d'utiliser l'élément pointé. Ceci se fait en ajoutant le suffixe ^ (accent circonflexe) au pointeur (^ est appelé *opérateur d'indirection* car il permet de passer du pointeur à l'élément pointé). Supposons que nous ayons un pointeur vers un entier, nommé "Machin". Dans ce cas, "Machin^" désigne l'élément pointé, et est donc dans ce cas précis un entier. Il peut être utilisé comme toute variable de type entier, à savoir dans une affectation par exemple, ou en tant que paramètre. Voici un exemple :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
type
  PInteger = ^integer;
var
  Machin: PInteger;
begin
  new(Machin);
  Machin^ := 1; { affectation d'une valeur entière à un entier }
  ShowMessage(IntToStr(Machin^)); { utilisation comme n'importe quel entier }
  dispose(Machin);
end;
```

Dans cet exemple, le pointeur est tout d'abord initialisé. Vient ensuite une affectation. Comme "Machin" est de type pointeur vers un entier, "Machin^", qui est l'élément pointé par le pointeur, est un entier, à qui on peut affecter une valeur entière. L'instruction suivante transmet la valeur de l'élément pointé par "Machin", soit toujours un entier, à IntToStr. Cette procédure, qui accepte les entiers, ne fait évidemment pas de difficulté devant "Machin^", qui vaut 1 dans notre cas. Le message affiché contient donc la chaîne "1". Enfin, le pointeur est détruit.
Voilà, il n'y a rien de plus à savoir pour utiliser l'élément pointé : utilisez-le lorsqu'il est initialisé, au moyen de l'opérateur d'indirection ^ et le tour est joué. Pour les types d'éléments pointés les plus simples, c'est tout ce qu'il faut savoir. Dans le paragraphe suivant, nous allons nous attarder sur les types plus complexes comme les tableaux et les enregistrements (record) utilisés avec les pointeurs.

## XI-B - Etude de quelques types de pointeurs

Ce paragraphe est destiné à vous fournir une aide concernant des types plus complexes que les types simples tels que les entiers ou les chaînes de caractères : aucune règle particulière n'est à appliquer à ces éléments de types plus complexes, mais leur utilisation avec les pointeurs demande un peu plus d'entrainement et de dextérité. Je vous propose donc quelques explications et exercices, le sujet étant intarrissable.

### XI-B-1 - Pointeurs de tableaux

Intéressons-nous au cas où l'on doit utiliser un pointeur vers un tableau. Nous évoquerons également les tableaux de pointeurs, bien entendu à ne pas confondre avec les pointeurs de tableaux.
Voici un tableau :
array[1..10] of integer
Si vous suivez mal ce que j'ai dit plus tôt, un pointeur vers ce genre de tableau serait du type :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

^array[1..10] of integer
Ce qui n'est pas le cas car "array[1..10] of integer" n'est pas un identificateur ! Oubliez vite la ligne ci-dessus, elle ne sera pas acceptée par Delphi et provoquera une erreur. Voici un bloc **type** correct qui déclare un type pointeur vers un tableau :

```
type
  TTabTest = array[1..10] of integer;
  PTabTest = ^TTabTest;
```

Tout élément déclaré de type PTabTest serait un pointeur vers un élément de type TTabTest. Or TabTest étant un tableau, un élément de type PTabTest est bien un pointeur vers un tableau.
Voici maintenant un exemple démontrant l'utilisation d'un tel pointeur de tableau :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
type
  TTabTest = array[1..100] of integer;
  PTabTest = ^TTabTest;
var
  References: PTabTest;
  indx: integer;
begin
  New(References);
  References^[1] := 0;
  for indx := 1 to 100 do
    References^[indx] := Trunc(Random(10000));
  ShowMessage('Référence n°24 : '+IntToStr(References^[24]));
  Dispose(References);
end;
```

Cet exemple utilise un type tableau de 100 entiers nommé "TTabTest". Un type pointeur vers ce tableau est ensuite déclaré et nommé "PTabTest". Le test utilise deux variables : un entier et un pointeur de tableau de type PTabTest. La première instruction initialise le pointeur. La deuxième instruction permet de voir comment on accède à un élément du tableau pointé par "References" : on commence par accèder à l'élément pointé, soît le tableau, grâce à l'opérateur d'indirection ^. "References^" étant l'élément pointé par "References", c'est un tableau d'entiers (type TTabTest). On peut donc accèder ensuite à sa première case en utilisant la syntaxe habituelle des tableaux, à savoir en ajoutant le numéro de case entre crochets. Ainsi, "References^[1]" est la première case du tableau pointé par "References" et est donc de type entier.
L'instruction en elle-même est une affectation. Le tableau étant un tableau d'entiers, chaque case du tableau est un entier et peut donc recevoir une valeur entière. On affecte donc la valeur 0 à "References^[1]" qui est une case du tableau. Cette instruction est en fait une initialisation de la case du tableau. L'instruction suivante passe à la vitesse supérieure : on utilise une boucle **for** pour initialiser directement TOUTES les cases du tableau pointé par "References". A cet effet, une variable "indx" parcours les valeurs de 1 à 100 et est utilisé comme numéro de case pour le tableau, ce qui permet donc de balayer une à une toutes les cases du tableau. Une affectation est faite à chaque fois, et ce avec une valeur aléatoire entière entre 0 et 10000.
L'instruction suivante (appel de ShowMessage) est un exemple d'utilisation de tableau de pointeur : on transmet la valeur de la case 24 du tableau pointé, donc un pointeur, à la fonction "IntToStr" qui la convertit en chaîne. Le reste est très conventionnel et permet d'afficher un message convenable. Enfin, la dernière instruction permet de "détruire" le pointeur (drôle de manière de dire les choses puisque ce n'est pas le pointeur mais l'élément pointé qui est détruit, conformément au schémas présenté plus haut).

## XI-B-2 - Pointeurs d'enregistrements (record)

Les pointeurs d'enregistrements sont tout simplement des types de pointeurs vers des enregistrements (record). L'utilité de ces pointeurs est considérable et nul doute que vous aurez de nombreuses occasions de vous en servir. La déclaration d'un type pointeur d'enregistrement se fait comme pour tous les types pointeurs vers un type particulier. Voici un exemple :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
type
  TSexePersonne = (spFeminin, spMasculin);
  TPersonne = record
    Nom,
    Prenom,
    CodePostal,
    Adresse,
    Ville: string;
    Sexe: TSexePersonne;
  end;
  { pointeur d'enregistrement }
  PPersonne = ^TPersonne;
```

De tels pointeurs s'initialisent et se détruisent comme d'habitude avec New et Dispose. Leur intérêt réside dans la gestion de la mémoire, sujet dont je vais parler juste un petit peu. La mémoire que vous utilisez dans vos programmes n'est pas illimitée : il faut éviter au maximum d'en utiliser de grosses quantités sans passer par les pointeurs. Cette mémoire se décompose à peu près en deux parties : la mémoire locale, et la mémoire globale. La première est celle que vous utilisez lorsque vous déclarez une variable locale par exemple. Cette mémoire est de taille très limitée. La mémoire globale, en revanche, est virtuellement limitée à 4 gigaoctets et Windows s'occupe d'en fournir lorsque c'est nécessaire. C'est là que les pointeurs interviennent : lorsque vous déclarez par exemple 100 enregistrements de type TPersonne, ces enregistrements sont stockés en mémoire locale, ce qui occupe pas mal de place. En utilisant des pointeurs, seuls ces derniers sont stockés en mémoire locale, et les éléments pointés le sont dans la mémoire globale.
Il faudra donc éviter cela :

```
array[1..1000] of  TPersonne
```

au profit de cela :

```
array[1..1000] of  PPersonne
```

la différence sst minime dans la déclaration, mais grande dans le fonctionnement et l'utilisation. Cette dernière forme est à préfèrer dans beaucoup de cas, même si l'utilisation de pointeurs rebute pas mal de programmeurs débutants.>
Venons-en à l'utilisation concrète de ces pointeurs d'enregistrements. En partant du pointeur, il faut accèder à l'élément pointé en placant l'opérateur d'indirection ^ . L'élément pointé étant un enregistrement, on peut accèder à un membre en écrivant un point suivi du nom du membre.
Voici un rapide exemple démontrant cela :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  PersTest: PPersonne;
begin
  New(PersTest);
  PersTest^.Nom := 'BEAULIEU';
  PersTest^.Prenom := 'Frédéric';
  PersTest^.Sexe := spMasculin;
  ShowMessage(PersTest^.Nom + ' ' + PersTest^.Prenom);
  Dispose(PersTest);
end;
```

Dans le code ci-dessus, une variable pointeur vers enregistrement est déclarée, initialisée, utilisée puis détruite. L'utilisation consiste à affecter des valeurs aux membres de l'enregistrement pointé par "PersTest". Vous pouvez au passage voir comment cela se fait. Une autre utilisation est faite en affichant un message comportant le nom et le prénom de la personne.
Il est également possible d'utiliser un bloc **with** avec de tels pointeurs, puisque **with** a besoin d'un enregistrement pour fonctionner. Voici la procédure réécrite avec un bloc **with** :

```
procedure TForm1.Button1Click(Sender: TObject);
var
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  PersTest: PPersonne;
begin
  New(PersTest);
  with PersTest^ do
    begin
      Nom := 'BEAULIEU';
      Prenom := 'Frédéric';
      Sexe := spMasculin;
      ShowMessage(Nom + ' ' + Prenom);
    end;
  Dispose(PersTest);
end;
```

Ce genre de bloc permet d'utiliser les pointeurs d'enregistrements à peu près comme les enregistrements eux-mêmes, en ajoutant initialisation et destruction. Dans l'exemple ci-dessus, PersTest est de type TPersonne, et convient donc bien là où on doit avoir un enregistrement.
Les pointeurs d'enregistrements ouvrent la porte à des structures très évoluées comme les listes chaînées ou les arbres. Ces deux notions sont des plus utiles pour modèliser certains problèmes. Nous n'étudierons cependant pas ces notions ici, car elles sont rendues non nécessaires par l'utilisation d'autres outils comme les objets TList sur lesquels, par contre, nous reviendrons prochainement.

## XI-C - Transtypage

Les pointeurs sont le domaine idéal pour commencer à vous parler de transtypage. Ne vous laissez pas effrayer par ce mot, il signifie tout simplement "transformation de type". Par transformation, on entend non pas une transformation de structure, mais un passage d'un type à un autre, une conversion explicite ou implicite. Commençons par les implicites.
Depuis longtemps, vous utilisez des transtypages implicites sans vous en rendre compte. Le transtypage est une conversion de type : ce genre de conversion est très utilisé dans Delphi de façon invisible, mais vous pourrez aussi vous servir de cette possibilité.
Prenons l'exemple suivant :

```
procedure TfmMain.btTestClick(Sender: TObject);
var
  X: Word;
begin
  X := 123;
  ShowMessage(IntToStr(X));
end;
```

Lorsque vous donnez X comme paramètre à IntToStr, vous donnez un élément de type Word à une procédure qui accepte un paramètre de type Integer. Cette différence de type est dans l'absolu interdite. Cela ne vous a jamais empèché de vivre jusqu'à maintenant, et que cela ne change pas, car Pascal Objet est un langage évolué et Delphi propose un compilateur puissant : lorsque vous donnez un paramètre du mauvais type, le compilateur regarde s'il est possible de convertir ce que vous lui présentez dans le bon type. Si c'est possible, le compilateur le fait parfois tout seul, notamment dans le cas des nombres ou de certaines chaînes de caractères. Aucune erreur n'est signalée, car Delphi fait les ajustements nécessaires dans votre dos.
Prenons un autre exemple :

```
procedure TfmMain.btTestClick(Sender: TObject);
var
  S1: string;
  S2: string[30];
begin
  S1 := 'Salut';
  S2 := S1;
  ShowMessage(S2);
end;
```

Ici, l'exemple est plus frappant : dans l'affectation "S2 := S1;", on affecte à une chaîne de longueur 30 une chaîne de longueur standard, c'est-à-dire illimitée sous Delphi 5 ou plus. Que se passerait-il si la longueur de S1 étant 45 par


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

exemple ? Lorsqu'on voit une simple affectation, on ne se doute pas de ce qui se fera par derrière : une troncature. S1 sera en effet tronquée si elle est trop longue pour tenir dans S2. Lorsqu'on transmet S2 à ShowMessage, il y a encore divergence des types car le type "string" est attendu alors qu'on transmet une variable de type "string[30]". Là encore, une conversion (techniquement, un transtypage) sera effectué.
Tout ce verbiage nous conduit à parler de transtypages, ceux-là explicites, que l'on peut donc écrire dans le code Pascal Objet. La notion de transtypage est liée à la notion de compatibilité de types. Certains types sont compatibles entre eux, c'est-à-dire qu'un transtypage est possible de l'un à l'autre, mais d'autres sont incompatibles et il n'y aurait aucune signification à effectuer de transtypages entre de tels types (par exemple d'une chaîne de caractères à un booléen).
Tous les types pointeurs sont compatibles entre eux, ce qui fait que n'importe quel pointeur peut être transtypé en n'importe quel autre type de pointeur. Ceci vient du fait que seul le type de l'élément pointé différe entre les types de pointeurs, et non la structure de stockages des pointeurs en mémoire.
Pour transtyper un élément, il faut écrire le nom du type à obtenir, suivi de l'élément entre parenthèses (un peu comme si l'on appelait une fonction, ce qui n'est absolument pas le cas). Les transtypages sont surtout utilisés avec les composants qui permettent d'adjoindre aux éléments contenus un ou des pointeurs. Ces pointeurs sont toujours de type "pointer", ce qui les rend impossibles à utiliser directement pour pointer quelque chose. Il faudra transtyper ces pointeurs en un type pointeur déclaré pour pouvoir accèder aux éléments pointés par le pointeur. Pour des exemples de transtypages, referrez-vous au (futur) chapitre sur les types abstraits de données, qui regorgera d'exemples et expliquera concrètement le transtypage des pointeurs. Rassurez-vous entretemps car cette notion ne devrait pas vous manquer jusque là.

## XI-D - Conclusion

C'est déjà la fin de ce chapitre consacré aux pointeurs. Nombre de notions importantes liées aux pointeurs n'y ont pas été abordées, pour ne pas tomber dans l'excés ou le bourrage de crâne. Des notions complèmentaires et nettement plus intéressantes sont abordées au **chapitre XV sur les types abstraits de données**. Parmi ces notions, seront abordés le transtypage de données et le chaînage.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XII - Objets

Nous abordons dans ce chapitre l'une des notions les plus importantes de la programmation moderne. En effet, la Programmation Orientée Objets, mieux connue sous l'acronyme de P.O.O, est très largement répandue et utilisée actuellement. Cette programmation , ou plutôt ce style de programmation, s'appuie sur le concept d'objets. Pascal Objet doit d'une part son nom au langage Pascal d'origine, et d'autre part aux grandes possibilités offertes par ce langage dans le domaine des objets depuis la première version de Delphi (et à moindre mesure dans les dernières versions de Turbo Pascal).
Ce chapitre présente la notion d'objet sans rentrer dans trop de détails techniques reservés à un futur chapitre et explique comment utiliser les objets. La notion de classe est également vue en détail, ainsi que la notion de hiérarchie de classes. Le rapport entre composants et objets sera également expliqué en détail.
Vous trouverez peut-être ce chapitre plus difficile que ce qui précède : c'est normal. La notion d'objet n'est pas simple à comprendre, mais une fois que l'on a compris le « truc », tout s'illumine. J'espère que ce sera le cas pour vous et je m'engage à tout faire pour faciliter cela.

## XII-A - Définitions des notions d'objet et de classe

La notion d'objet est, à mon sens, une des notions les plus passionnantes de la programmation. Bien qu'apparament très complexe et étendue, la notion d'objet répond à des besoins très concrets et rend possible l'existence de systèmes évolués tels que les environnements graphiques comme Microsoft Windows. Sans cette notion, programmer de tels systèmes relèverait sinon de l'utopie, au moins de l'enfer.
Le langage Pascal Objet doit bien sûr la deuxième partie de son nom aux grandes possibilités de ce langage dans l'exploitation des objets. La notion d'objet est basée sur un besoin simple : celui de regrouper dans une même *structure* des données et du code (des instructions en langage Pascal Objet dans notre cas) permettant de manipuler ces données. Les objets répondent à ce besoin, en apportant une foule de possibilités très puissantes qui en font un passage presque obligé dans la majorité des gros projets développés sous Delphi. Nous ne verrons ici qu'une petite partie de ces possibilités, l'essentiel étant réservé au chapitre sur la création d'objets.
Avant d'en venir aux définitions, il faut savoir que le concept d'objet est indépendant des langages qui permettent d'utiliser les objets. Nombre de langages plus ou moins modernes supportent les objets, mais chacun supporte un certain nombre de fonctionnalités liées aux objets. Les possibilités offertes par Pascal Objet sont très correctes, mais toutefois moins étendues que celles offertes par le langage C++ par exemple.

### XII-A-1 - Objet

Concrètement, un objet est une donnée possédant une structure complexe. On utilisera les objets en utilisant des variables Pascal Objet, comme cela se fait déjà avec les autres données classiques telles les chaînes de caractères ou les pointeurs. Ces variables contiennent, puisque c'est leur principe de base, des données et du code Pascal Objet permettant de les traiter. C'est une des nouveautés marquantes des objets : on peut y stocker des données mais aussi des instructions Pascal Objet (nous verrons cela en détail dans le chapitre sur la création d'objets). Les données consistent en des variables de n'importe type (y compris des objets). Le code Pascal Objet est réparti dans des procédures et fonctions nommées *méthodes* (c'est leur dénomination dans le monde des objets). Si ce terme de méthode vous est déjà familier, ce n'est pas dû au hasard : les composants, auxquels se rattachait jusqu'ici le concept de méthodes, sont en effet des objets.
Delphi propose un large éventail d'objets tout faits, que nous allons apprendre à utiliser dans ce chapitre. Cette collection livrée avec Delphi s'appelle la VCL (Visual Component Library : Bibliothèque de composants visuels). Cette VCL fait de Delphi un instrument unique car aucun autre outil de programmation visuelle ne propose une telle bibliothèque prête à l'emploi et aussi facile à utiliser. La création d'objets est également possible mais constitue un objectif plus ambitieux et plus complexe que nous aborderons dans un prochain chapitre.
Les méthodes et variables contenues dans les objets sont complétés par d'autres éléments. Parmi ces éléments particuliers, figurent les propriétés (que vous connaissez bien maintenant, n'est-ce pas ?) et deux méthodes aux dénominations un peu barbares puisqu'on les appelle constructeur et destructeur.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XII-A-2 - Classe

Déjà cité deux ou trois fois dans ce chapitre, ce terme n'a pas encore fait l'objet d'une explication. Nous venons de voir que les objets sont des variables. Les classes sont les types permettant de déclarer ces variables. Là où auparavant une variable était d'un type donné, un objet sera dit d'une classe donnée. Venons-en tout de suite à un exemple :

```
var
  S: string;
  Button1: TButton;
```

Dans l'exemple ci-dessus, deux variables sont déclarées. Leur noms sont « S » et « Button1 ». Les types de ces deux variables sont respectivement « string » et « TButton ». Vous connaissez bien le type « string » pour l'avoir utilisé de nombreuses fois depuis le début de ce guide. Le type « TButton » ne vous est pas non plus inconnu puisqu'il figure dans le code source de vos applications depuis un certain temps déjà. Si vous ne voyez pas où, créez une nouvelle application, placez un bouton sur la seule fiche du projet, et allez regarder le source de l'unité associée à la fiche : vous devriez pouvoir trouver un extrait de code similaire à celui présenté ci-dessous :

```
...
type
  TForm1 = class(TForm)
    Button1: TButton;
    ...
```

le type TButton correspond en fait au composant Button décrit utilisé dés les premiers chapitres de ce guide. Le type « TButton » est une classe, c'est-à-dire que la variable « Button1 » de type « TButton » est un objet. Comme vous pouvez le constater, déclarer un objet se fait de la même manière que pour une variable, à ceci près qu'on le déclare en utilisant une classe et non un type classique.
Une classe, un peu comme le fait un type pour une variable, détermine entièrement la structure des objets de cette classe. Plus précisément, c'est la classe qui définit les méthodes et les données (mais pas leurs valeurs) contenues dans les futurs objets de cette classe. Pour reprendre un hyper-classique (peut-être pas pour vous, désolé), une classe peut être comparée à un moule à gâteaux. Les objets, quant à eux, je vous le donne en mille, sont les gâteaux qu'on peut réaliser à partir de ce moule. En admettant que le moule définisse entièrement la forme du gâteau, c'est la même chose pour les classes : elles déterminent entièrement la structure des objets de cette classe (de ce type, donc). Par contre, lorsque vous avez un moule à tarte par exemple, qui peut donc être assimilé à une classe précise, vous pouvez faire une tarte aux pommes ou aux poires, ce n'est pas contrôlé par le moule. Il en va de même pour la classe en ce qui concerne les données : elle définit les noms et les types des données mais pas leurs valeurs. Chaque objet a ses données dont les noms et types sont déterminées par la classe, mais dont les valeurs sont indépendantes pour chaque objet. Ainsi, la classe « TMouleATarte » définirait une variable « Fruit » de type énuméré « TSorteDeFruit » par exemple. Chaque objet de classe « TMouleATarte » pourrait alors spécifier une valeur pour « Fruit ». Cette valeur serait indépendante des autres objets et de la classe.
Et ceci termine les définitions. Dans le paragraphe suivant, vous allez apprendre à utiliser les objets et les classes.

## XII-B - Utilisation des objets

Les objets ne s'utilisent pas exactement comme les autres variables. Comme les pointeurs, ils nécessitent des opérations spécifiques et leur utilisation se fait en respectant certaines règles qui ne doivent rien au hasard. Ce paragraphe est dédié à l'utilisation et à la manipulation des objets.

## XII-B-1 - Construction et destruction

Comme les pointeurs, les objets nécessitent deux étapes particulières avant et après leur utilisation. Un objet se construit, s'utilise puis se détruit. La construction se fait par une instruction particulière que nous verrons ci-dessous. La destruction se fait également par une simple instruction qui fait appel au destructeur.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Afin d'illustrer ces propos quelque peu abstraits, nous allons étudier une classe proposée par Delphi : la classe TStringList. Son but est de manipuler une liste de chaînes de caractères. Vous allez vous familiariser avec la manipulation des objets en apprenant à vous servir d'un objet de cette classe.
Commençons donc par déclarer un objet de classe TStringList. Nous utiliserons la désormais classique procédure associée à l'événement OnClick d'un bouton.

```
var
  Lst: TStringList;
```

L'exemple-ci-dessus déclare un objet Lst de type TStringList. La construction d'un objet se fait par une instruction dont la syntaxe est la suivante :
*objet := classe.constructeur[(parametres)]*
*objet* désigne l'objet que vous voulez construire. *classe* est la classe de l'objet à construire et donc qui a servi à déclarer la variable qui représente l'objet.
*constructeur* est le nom du constructeur défini par la classe de l'objet. Le plus souvent, ce constructeur s'appelle « create ». Cette méthode, comme toutes les procédures, peut éventuellement avoir des paramètres. Ceux-ci sont le cas échéant et comme d'habitude transmis entre parenthèses et sont indiqués ci-dessus entre crochets pour indiquer qu'ils sont parfois absents. Vous remarquerez que l'appel du constructeur se fait comme l'appel d'une méthode pour les composants. En fait, le constructeur est une méthode et les composants sont des objets, ce qui explique cette similarité. Vous remarquerez également que l'instruction est une affectation. Cette écriture permet certaines constructions intéressantes que nous étudierons plus tard. La raison de cette construction est qu'appeler une méthode (même un constructeur) d'un objet non déjà construit est absurde : on utilise la classe de l'objet en lieu et place de l'objet (ceci sera expliqué en détail dans le chapitre consacré à la création d'objets).
Comme la construction et la destruction vont systématiquement de paire, la destruction se fait par une instruction simple qui a la forme suivante :
*objet.destructeur[(parametres)]*
*objet* désigne l'objet que l'on souhaite détruire. Il ne sera plus possible d'utiliser l'objet après l'instruction, à moins de le reconstruire. *destructeur* est le nom du destructeur défini par la classe de l'objet. Il est assez rare qu'un destructeur ait besoin de paramètres, si c'est le cas, il suffit de les transmettre entre parenthèses.
Voici maintenant la construction et la destruction en pratique :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lst: TStringList;
begin
  Lst := TStringList.Create;
  Lst.Destroy;
end;
```

La procédure ci-dessus construit l'objet Lst. Examinez bien l'instruction : « create » est le constructeur de la classe TStringList. Il n'a pas de paramètres. Au-delà de cette construction, l'objet est utilisable. Nous allons voir l'utilisation dans un prochain paragraphe. La destruction est effectuée pour cette fois immédiatement après la construction. Vous constaterez que la destruction, contrairement à la construction qui prend une forme spéciale, est un simple appel à la méthode spéciale qu'est le destructeur.

## XII-B-2 - Manipulation des objets

Il va y avoir un air de déjà vu ici si vous avez suivi le chapitre sur la manipulation des composants. En effet, tous les composants étant des objets, ils se manipulent comme les objets (et non le contraire, comme on serait tenté de le croire, car il y a des objets qui ne sont pas des composants, comme les objets de classe TStringList). Ce qui va être dit ici est donc une généralisation de ce qui a été dit pour les composants et non pas une simple répétition.
Nous avons vu qu'un objet contient des variables, des méthodes, des propriétés et deux méthodes spéciales que sont le constructeur et le destructeur. Parmi tous ces éléments, seuls certains sont accessibles. Cette notion d'accessibilité sera expliquée au paragraphe sur les sections publiques et privées d'un objet.
Parmi les éléments accessibles figurent rarement les variables : elles sont souvent le domaine réservé de l'objet et il n'est nul besoin d'y avoir accès. Par contre, un certain nombre de méthodes et de propriétés sont accessibles. Le


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

constructeur et le destructeur sont bien évidemment toujours accessibles. Pour faire appel à une méthode d'un objet, il suffit d'employer la syntaxe habituelle :
*objet.methode[(parametres)]*
*objet* désigne l'objet dont on veut appeler une méthode. *methode* est la méthode qu'on veut appeler. Si cette méthode a des paramètres, il suffit de les transmettre entre parenthèses. Etant donné que les méthodes regroupent des procédures et des fonctions, dans le cas de ces dernières, le résultat est utilisable comme une valeur du type du résultat de la fonction. Par exemple, dans une affectation, cela donne :
*variable := objet.fonction_methode[(parametres)]*
La plupart des objets possèdent également des propriétés. Leur utilisation se fait comme pour un composant. Selon ce que la propriété est en lecture seule ou en lecture/écriture, elle s'utilise comme une constante ou une variable du type de la propriété. Certaines propriétés particulières comme les propriétés tableau ou objet sont également au programme. Les premières, vous les connaissez déjà. Les deuxièmes sont simplement des propriétés de type objet, c'est-à-dire d'une classe déterminée.
Poursuivons l'exemple débuté au paragraphe précédent. Nous allons ajouter une chaîne à la liste. L'ajout d'une chaîne se fait en appelant la méthode "Add" de l'objet liste de chaînes. Cette méthode est une procédure qui accepte en unique paramètre la chaîne à ajouter à la liste. Voici ce que cela donne en pratique :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lst: TStringList;
begin
  Lst := TStringList.Create;
  Lst.Add('Bonjour !');
  Lst.Destroy;
end;
```

Vous remarquerez que l'objet est construit, puis utilisé, et enfin détruit : c'est ainsi qu'il faudra toujours faire, même si ces trois étapes auront tendance à s'espacer avec le temps et l'expérience. Vous noterez également que nous ne nous soucions pas d'effacer la chaîne de la liste, car c'est une tâche effectuée automatiquement lors de la destruction de l'objet (c'est le destructeur qui fait ce travail pour nous). L'instruction qui ajoute une chaîne à la liste est très facile à décortiquer : on appelle la méthode Add de l'objet Lst avec le paramètre 'bonjour !'.
La classe TStringList possède une propriété en lecture seule "Count" (de type integer) indiquant le nombre de chaînes actuellement dans la liste. Nous allons afficher deux messages indiquant cette valeur, avant et après l'ajout de la chaîne. Le premier message devrait indiquer 0 et le second 1.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lst: TStringList;
begin
  Lst := TStringList.Create;
  ShowMessage(IntToStr(Lst.Count));
  Lst.Add('Bonjour !');
  ShowMessage(IntToStr(Lst.Count));
  Lst.Destroy;
end;
```

La propriété Count est utilisé comme une constante puisqu'elle est en lecture seule. On utilise la fonction IntToStr pour transformer sa valeur en chaîne de caractère affichée par ShowMessage. Le premier appel se situe juste après la construction de l'objet : à ce moment, aucune chaîne n'est encore présente dans la liste. Le nombre de chaînes est donc 0 et c'est bien ce qu'affiche le message. Après l'ajout de la chaîne, la même opération est effectuée. Comme une chaîne a été ajoutée, l'objet a « mis à jour » sa propriété Count et la valeur renvoyée est non plus 0 mais 1.
C'est très intéressant d'ajouter des chaînes à une liste, mais il faut également pouvoir y accèder. Ceci se fait par une propriété tableau, qui a l'immense avantage d'être par défaut, ce qui vous dispense de retenir son nom : Strings. Cette propriété tableau a ses indices numérotés à partir de 0. La première chaîne de la liste, lorsqu'elle existe, est donc :

```
Lst.String[0]
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

ce que l'on peut écrire plus simplement (et cette deuxième écriture, équivalente à la première, lui sera désormais systématiquement préférée, car un bon développeur est comme un matheux : partisan du moindre effort) :

```
Lst[0]<br/>
```

Vous admettrez que c'est plus court et plus facile à retenir, bien que moins explicite. Chacune de ces deux expressions est de type String. Modifions donc notre petit programme pour qu'il affiche la chaîne que nous venons d'ajouter dans la liste :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lst: TStringList;
begin
  Lst := TStringList.Create;
  Lst.Add('Bonjour !');
  ShowMessage(Lst[0]);
  Lst.Destroy;
end;
```

Lst[0], qui est de type String, est affiché par ShowMessage, ce qui permet à votre ordinateur d'être poli pour la première fois de la journée.
Dans les listes d'éléments telles que les objets de classe TStringList en manipulent, les éléments sont souvent acccessible via une propriété tableau indexée à partir de 0. Une propriété, souvent nommée Count, permet de connaître le nombre d'éléments dans la liste. Les éléments sont donc, sous réserve qu'il y en ait dans la liste, indexés de 0 à Count - 1. Si je prends le temps de bien indiquer cela, c'est parce qu'il est souvent nécessaire de passer en revue tous les éléments via une boucle for. Les bornes inférieures et supérieures à utiliser sont dans ce cas 0 et Count - 1. Voici une nouvelle version de la procédure qui utilise cela pour afficher toutes les chaînes ajoutées à la liste.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Lst: TStringList;
  indx: integer;
begin
  Lst := TStringList.Create;
  Lst.Add('Bonjour !');
  Lst.Add('Maître');
  if Lst.Count > 0 then
    for indx := 0 to Lst.Count - 1 do
      ShowMessage(Lst[indx]);
  Lst.Destroy;
end;
```

L'exemple ci-dessus ajoute deux chaînes à la liste. Le suspense sur la valeur de Count n'est pas vraiment très intense, mais essayez d'oublier que Count a une valeur connue dans cet exemple particulier : pensez au cas général où Count est une valeur inconnue. La boucle for affiche toutes les chaînes présentes dans la liste, quel que soit leur nombre. indx parcourant les valeurs entre 0 et Count - 1, Lst[indx] parcours donc toutes les chaînes de la liste. Le bloc **for** est inclus dans un bloc **if** qui permet de n'effectuer l'affichage que si la propriété Count vaut au moins 1, c'est-à-dire que l'affichage n'est lancé que s'il y a quelque chose à afficher.

## XII-B-2-a - Exercice résolu

Passons à la vitesse supérieure. Nous allons utiliser une fonctionnalité fort appréciable de la classe TStringList : la méthode Sort. Cette méthode applique un tri alphabétique sur les chaînes présentes dans la liste maintenue par l'objet dont on appelle la méthode Sort. Concrètement, les chaînes sont simplement réorganisées de sorte que la première dans l'ordre alphabétique ait l'indice 0 dans la propriété tableau par défaut de l'objet. Pour illustrer cela, nous allons confectionner une procédure (toujours déclenchée par un clic sur un bouton, pour faire dans l'originalité) qui demande de saisir des chaînes. La procédure demandera des chaînes jusqu'à ce que l'on clique sur "Annuler"


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

et non "OK". La procédure affichera alors le nombre de chaînes entrées dans la liste, puis les chaînes dans l'ordre alphabétique.
Si vous voulez réaliser cela comme un exercice, c'est le moment d'arrèter de lire et de vous mettre à chercher : ce qui suit réalise pas à pas ce qui vient d'être proposé ci-dessus.
La première chose à faire est de bien discerner les étapes à programmer :

1. initialisation de l'objet qui gérera la liste
2. une boucle demandant un nombre à priori inconnu de chaînes
3. l'affichage du nombre de chaînes
4. le tri alphabétique de la liste
5. l'affichage d'un nombre connu de chaînes
6. destruction de l'objet

La première étape nécessite une boucle : chaque itération lira une chaîne et la boucle s'arrètera lorsque la condition "la chaine entrée est la chaîne vide" est réalisée. Le problème est qu'une boucle **for** est inadaptée puisque le nombre de chaînes que l'utilisateur désire rentrer est inconnu et puisque nous ne souhaitons pas fixer ce nombre. Restent deux boucles possibles : une boucle **while** ou une boucle **repeat**. Pour savoir laquelle utiliser, la question est toujours la même : le contenu de la boucle sera-t-il à exécuter au moins une fois ? La réponse dépend de ce que nous allons effectuer à chaque itération, c'est pour cela qu'il est toujours nécessaire de réfléchir à un programme d'abord sur papier avant de se lancer dans l'écriture du code.
Nous allons utiliser la fonction InputQuery pour lire les chaînes à rentrer dans la liste. Cette fonction renvoie un résultat booléen qui peut être utilisé indifférement dans une condition d'arrèt ou de poursuite (selon le type de boucle choisie). Etant donné que la fonction renvoie true si l'utilisateur valide sa saisie, on peut utiliser la boucle suivante :

```
while InputQuery({...}) do
  {...}
```

Lors de l'entrée dans la boucle, InputQuery sera exécutée. Son résultat sera utilisé comme condition d'arrèt : si l'utilisateur annule, on n'effectue pas d'itération, s'il valide, on effectue une itération et on refait une demande de chaîne, et ainsi de suite. Nous aurions pu écrire les choses de façon plus compréhensible mais plus longue en utilisant une variable booléenne :

```
Cont := True;
while Cont do
  begin
    Cont := InputQuery({...});
    ...
  end;
```

ou même comme cela :

```
repeat
  Cont := InputQuery({...});
  ...
until not Cont;
```

mais ces deux méthodes auraient introduit des difficultés supplémentaires car il aurait fallu tester la valeur de Cont avant d'effectuer un quelconque traitement, ce qui nous aurait obligé à utiliser une boucle **if**. La méthode choisie est plus propre et est plus proche d'un raisonnement logique du genre "tant que l'utilisateur valide sa saisie, on traite sa saisie".
Passons au contenu de la boucle, c'est-à-dire à ce qui est éxécuté à chaque itération : La chaîne entrée par l'utilisateur est tout simplement ajoutée à la liste. Il faudra également penser à vider la chaîne lue car rappelez-vous que cette chaîne est affichée dans la boite de dialogue de saisie. Il faudra aussi la vider avant la première itération car la première lecture s'effectuera avant le premier « vidage ». Voici donc la première partie de la procédure, qui inclue


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

également la construction et la destruction de l'objet de classe TStringList puisque ce dernier est utilisé dans la boucle **while** :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StrLst: TStringList;
  STemp: String;
begin
  StrLst := TStringList.Create;
  STemp := '';
  while InputQuery('Entrez une chaîne', 'Chaine à ajouter à la liste :', STemp) do
    begin
      StrLst.Add(STemp);
      STemp := '';
    end;
  StrLst.destroy;
end;
```

Ce code permet de lire autant de chaînes que l'on veut et de les stocker dans une liste de chaînes. La deuxième étape qui consiste à afficher le nombre de chaînes entrées par l'utilisateur est la plus simple de ce que nous avons à faire :

```
ShowMessage('Vous avez entré ' + IntToStr(StrLst.Count) + ' chaînes');
```

Il nous faut ensuite trier la liste. Pour cela, il suffit d'appeler la méthode Sort de l'objet StrLst :

```
StrLst.Sort;
```

Enfin, il nous faut afficher les chaînes. Ici, on ne sait pas combien il y en a, mais ce nombre est contenu dans la propriété Count de l'objet. On utilise donc une boucle **for** avec une variable allant de 0 à StrLst.Count - 1. L'affichage passe par un simple appel à ShowMessage. Pour donner du piquant à cet affichage, on donne à l'utilisateur un affichage du style "chaîne n° x sur y : chaîne" (indx + 1 est utilisé pour donner le numéro actuel, en partant de 1 au lieu de 0, et StrLst.Count est utilisé pour donner le nombre de chaînes) :

```
for indx := 0 to StrLst.Count - 1 do
  ShowMessage('Chaîne n° ' + IntToStr(indx + 1) + ' sur ' +
    IntToStr(StrLst.Count) + ' : ' + StrLst[indx]);
```

Voici enfin le code complet de la procédure que nous avons créée. Vous constaterez qu'elle est très courte, mais qu'elle fait intervenir nombre de notions importantes non seulement concernant les objets, mais aussi concernant les boucles.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StrLst: TStringList;
  STemp: String;
  indx: integer;
begin
  StrLst := TStringList.Create;
  STemp := '';
  while InputQuery('Entrez une chaîne', 'Chaine à ajouter à la liste :', STemp) do
    begin
      StrLst.Add(STemp);
      STemp := '';
    end;
  ShowMessage('Vous avez entré ' + IntToStr(StrLst.Count) + ' chaînes');
  StrLst.Sort;
  for indx := 0 to StrLst.Count - 1 do
    ShowMessage('Chaîne n° ' + IntToStr(indx + 1) + ' sur ' +
      IntToStr(StrLst.Count) + ' : ' + StrLst[indx]);
```



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  StrLst.destroy;
end;
```

## XII-C - Notions avancées sur les classes

Ce paragraphe présente deux aspects importants de la programmation orientée objet. Pour les débutants, ces notions sont inutiles, mais tout développeur qui veut s'initier sérieusement non seulement à l'utilisation mais aussi à la création d'objets se doit de les connaître.

### XII-C-1 - Hiérarchie des classes

Nous avons déjà parlé des classes en expliquant qu'elle sont les types à partir desquels sont déclarés les objets. Ces classes ne sont pas simplement un ensemble désorganisé dans lequel on pioche : il existe une hiérarchie. Cette hiérarchie est basée, comme les objets, sur un besoin simple : celui de ne pas réinventer constamment la roue.

#### XII-C-1-a - Concept général

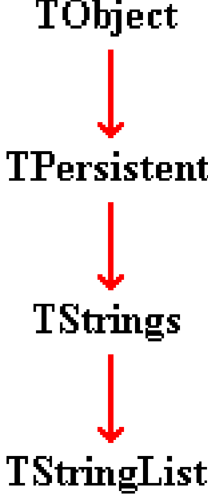
Les objets sont des structures pour la plupart très complexes dans le sens où ils possèdent un nombre important de méthodes, de variables et de propriétés. Mettez-vous un instant à la place d'un programmeur chevronné et imaginez par exemple tout ce qu'il faut pour faire fonctionner un simple bouton : il faut entre autres le dessiner, réagir au clavier, à la souris, s'adapter en fonction des propriétés. C'est une tâche, croyez-moi sur parole, qui nécessite un volume impressionnant de code Pascal Objet. Prenons un autre composant, par exemple une zone d'édition : elle réagit également au clavier et à la souris. Ne serait-il pas intéressant de pouvoir « regrouper » la gestion du clavier et de la souris à un seul endroit pour éviter de la refaire pour chaque composant (pensez qu'il existe des milliers de composants).
De tels besoins de regroupement, il en existe énormément. Pour cela, il existe la notion de hiérarchie parmi les classes. Cette hiérarchisation permet de regrouper dans une classe « parent » un certain nombre de propriétés, de méthodes et de variables. Les classes qui auront besoin d'avoir accès à ces fonctionnalités devront simplement « descendre » de cette classe, c'est-à-dire être une classe « descendante » de cette classe « parent ».
Le principe de la hiérarchie des classes est en effet basé sur la relation parent-enfant ou plus exactement parent-descendant. Chaque classe possède, contrairement à ce que l'on peut voir d'insolite dans les relations parent-descendant humaines, une seule et unique classe parente directe. Une classe peut avoir un nombre illimité de descendants. Lorsqu'une classe descend d'une autre classe, la première possède absolument tout ce qui est défini par la seconde : c'est **l'héritage**. La classe « descendante » est plus puissante que la classe « parente » dans le sens où de nouvelles méthodes, variables et propriétés sont généralement ajoutées ou modifiées.
Une telle hiérarchie impose l'existence d'un unique ancètre ultime qui n'a pas de parent : c'est la classe « TObject » (qui possède un nom quelque peu embrouillant). De cette classe descendent TOUTES les classes existantes sous Delphi, et ceci plus ou moins directement (toutes les classes ne sont pas des descendantes directes de TObject mais sont souvent des descendantes de descendantes de... de « TObject ». Cette dernière définit les mécanismes de base du fonctionnement d'un objet. Tous les objets sous Delphi sont d'une classe descendante de TObject, et possèdent donc tout ce que définit TObject, à savoir le minimum.
La classe « TObject » ne vous est pas inconnue car elle est mentionnée dans toutes les procédures associées à des événements. Voici un extrait de code qui devrait vous aider à y voir plus clair :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
begin

end;
```

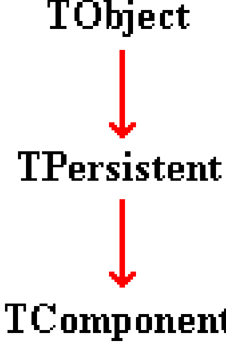
Dans l'extrait de code ci-dessus, la seule partie qui restait énigmatique, à savoir « Sender: TObject », est simplement la déclaration d'un paramètre de type TObject, c'est-à-dire que Sender est un objet de classe TObject.
Prenons une classe que vous connaissez : « TStringList ». Cette classe descend de « TObject », mais pas directement. Elle a pour parent la classe « TStrings ». La classe « TStrings » peut avoir un nombre illimité de classes descendantes, et a entre autre « TStringList » comme descendante directe. « TStrings » est moins puissante que « TStringList » car cette dernière, en plus de l'héritage complet de ce que contient « TStrings », contient bien


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

d'autres choses. La classe « TStrings » a pour parent la classe « TPersistent ». Cette classe, qui est encore moins perfectionnée que « TStrings », a enfin pour parent la classe « TObject », ce qui termine l' « arbre généalogique » de TStringList. Voici un schémas qui résume cette descendance :

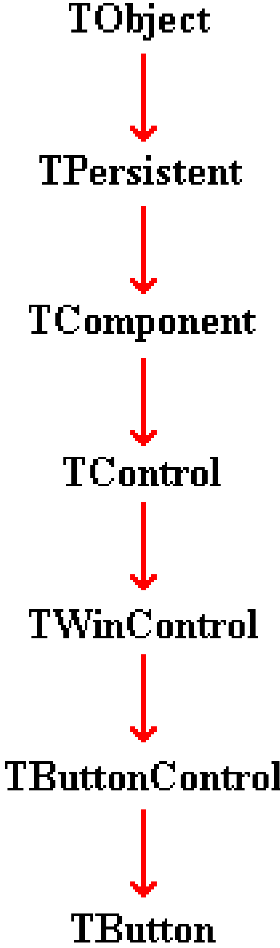



Nous avons déjà plusieurs fois dit au cours de ce chapitre que les composants sont des objets. Il est maintenant temps de parler de la classe TComponent. Cette classe est elle aussi descendante de TObject. Voici son « arbre généalogique, pour curiosité :



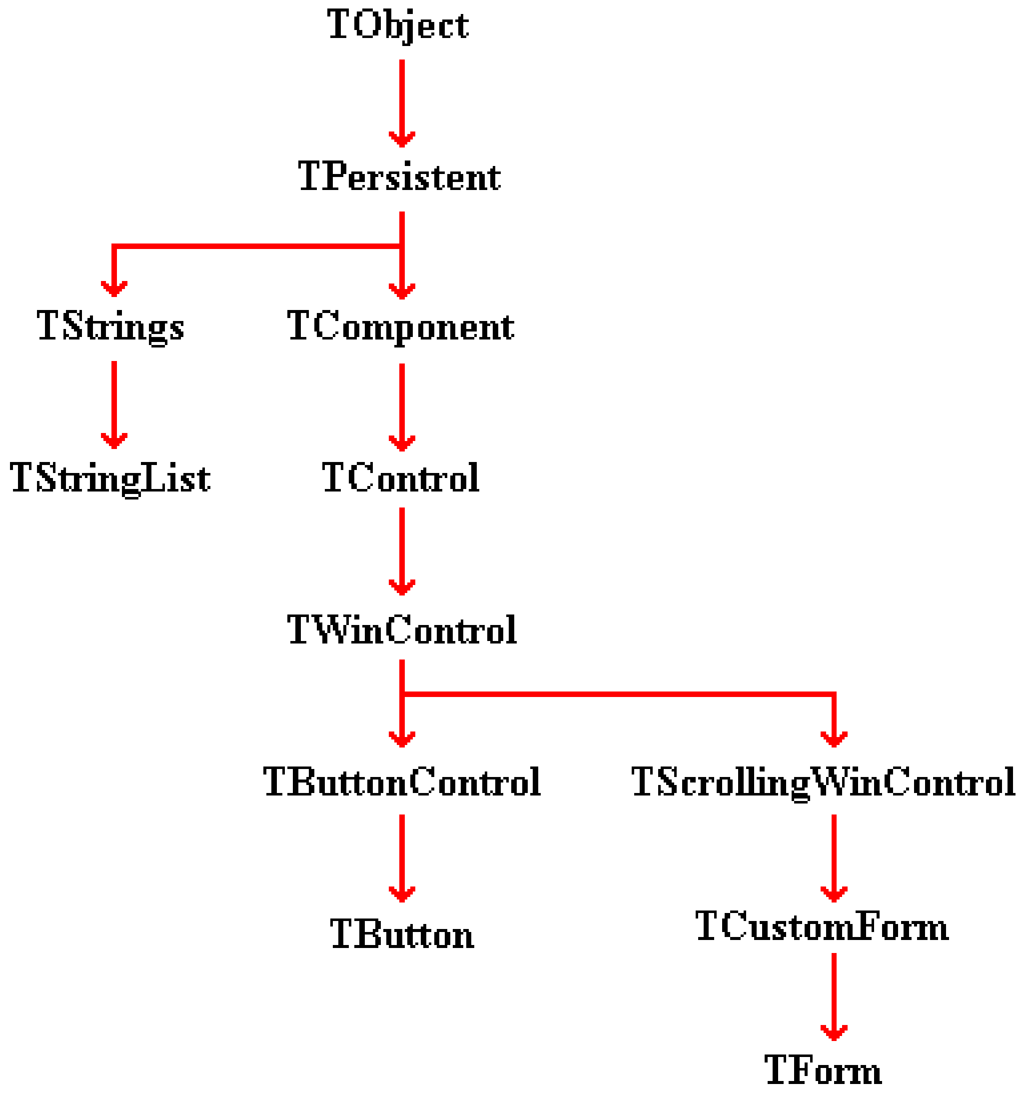

L'intérêt de cette classe, c'est qu'elle est à l'origine de la définition des composants. Un composant est en fait par définition un objet dont la classe descend de TComponent. TComponent définit les caractéristiques de base nécessaire à tout composant. Tous les composants que vous connaissez, et tous les autres, sont d'une classe descendante de TComponent.
Prenons la classe TButton qui permet de créer les composants boutons bien connus. Cette classe est bien un descendant de TComponent, mais la descendance est indirecte puisqu'il y a des classes intermédiaires qui ajoutent chacune un grand nombre de fonctionnalités. Voici l' « arbre généalogique » de TButton :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Comme vous pouvez le constater, TButton possède une longue ascendance pour atteindre TObject. C'est en quelque sorte le reflet de la complexité du composant : on ne se rend pas compte lorsqu'on place un bouton sur une fiche que c'est une petite usine à gaz qu'on met en fonctionnement. Puisque nous en sommes à parler des fiches, la classe TForm qui permet de définir des fiches est également une descendante de TComponent. Son arbre généalogique a été inclus ci-dessous, et les composants TStringList, TButton ont été inclus dans le schéma pour vous montrer qu'on obtient visuellement une sorte de hiérarchie entre les classes lorsqu'on en prend un nombre suffisant :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



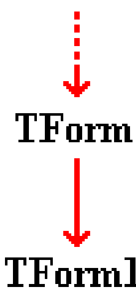

## XII-C-1-b - Classes descendantes de TForm

Si j'insiste autant sur cette notion de hiérarchie entre les classes, c'est non seulement car sa compréhension est indispensable pour tout bon programmeur, mais surtout pour en venir à un certain bloc de code que vous connaissez pour l'avoir déjà vu un grand nombre de fois (il apparaît lorsque vous créez un projet vierge ou une nouvelle fiche, avec de petites variantes) :

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

La ligne qui nous intéresse plus particulièrement ici est : « TForm1 = **class**(TForm) ». Cette ligne, nous l'avons déjà expliqué, définit un nouveau type "TForm1". Ce qui est nouveau et que vous pouvez désormais comprendre, c'est que cette ligne déclare "TForm1" comme une classe (mot réservé **class**) descendante de la classe "TForm". Une relation de type « parent - descendant » existe entre TForm1 et TForm. Voici la relation entre les deux, exprimée comme un morceau d' « arbre généalogique » (l'ascendance de TForm a été omise pour gagner un peu de place) :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




La classe "TForm1" descendante de "TForm" peut alors définir de nouveaux éléments : méthodes, variables, propriétés, ... Les éléments ajoutés sont listés d'une manière spécifique que nous ne décrirons pas en détail ici, entre les mots réservés **class** et **end**.
Faites l'expérience d'ajouter un bouton et de générer la procédure associée à son événement OnClick. L'extrait de code devient :

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Vous voyez que deux lignes ont fait leur apparition, l'une décrivant un élément Button1 de type TButton et l'autre une procédure. Ces deux éléments sont deux nouveaux éléments de la classe TForm1 : ils ne sont pas présents dans la classe TForm. La classe TForm1 a été améliorée par rapport à TForm en ce sens qu'elle possède deux nouveaux éléments. Le premier est une variable interne (on nommera plutôt *champ* une variable interne à une classe) et le second une méthode de la classe TForm1. Tout objet de cette classe possède ces deux éléments, en particulier l'objet Form1 déclaré de classe TForm1.
Vous voyez peut-être ici toute la puissance de la hiérarchie des classes de Delphi : pour créer une fiche comportant un bouton, il a suffit d'exploiter la classe TForm qui permet d'utiliser une fiche vierge de base, de créer une classe descendante de TForm, et d'y ajouter ce qu'on voulait en plus sur la fiche, ici un composant Button1 de classe TButton. Sans cette hiérarchie des classes, il aurait fallu beaucoup plus de code car il aurait fallu passer par des appels à Windows et par des procédures relativement complexes pour un débutant.
La partie située entre **class** et **end** est la déclaration de la classe TForm1. Dans ce morceau de code, on place les déclarations des éléments de la classe. Les variables se déclarent comme dans un bloc **var**, les méthodes se déclarent comme dans une unité, à savoir qu'on inclut la ligne de déclaration que l'on placerait habituellement dans l'interface de l'unité. Ce bloc de code qui contient la déclaration de la classe "TForm1" ne déclare bien entendu que ce qui est ajouté à TForm1 en plus de ce que contient TForm. Ce bloc est divisé en plusieurs parties, que nous allons étudier avant de pouvoir nous-mêmes ajouter des éléments aux classes.

## XII-C-2 - Ajout d'éléments dans une classe

Un des buts majeurs de ce chapitre est de vous apprendre à ajouter vos propres méthodes et variables aux classes. Pour l'instant, il est hors de question de créer nos propres classes : nous allons nous contenter d'ajouter des méthodes et des variables aux classes définissant les fiches (comme TForm1 dans le paragraphe précédent), ce qui est déjà un bon début.
Avant de nous lancer, il faut encore connaître une notion sur les classes : celle de sections. Le paragraphe suivant est consacré à une explication de cela. Le paragraphe suivant explique concrètement comment ajouter des éléments à une classe.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XII-C-2-a - Sections privées et publiques d'une classe

Jusqu'ici, nous avons appris qu'une classe contenait des méthodes, des champs (nous allons passez progressivement de la dénomination de variable à celle de champ, qui est préférable dans le cadre des classes), des propriétés, et encore d'autres choses. Tout comme les classes ne sont pas un ensemble désorganisé, il existe à l'intérieur d'une classe un certain classement, dont le principe n'a rien à voir avec la notion de hiérarchie des classes, et qui est également plus simple à comprendre.
Une classe comporte en pratique un maximum de cinq sections. Chaque élément défini par une classe est classé dans l'une de ces sections. Parmi ces cinq sections, nous allons en étudier seulement trois ici. Les sections sont visibles au niveau du code source car délimitées par des mots réservés à cet effet. Le début d'une section est généralement donné par un mot réservé, sa fin est donnée par le début d'une autre section ou la fin de la définition de la classe (par le mot réservé **end**). La première section que l'on rencontre dans une classe est située avant toutes les autres : elle est délimitée au début par **class**(...) et est terminée comme indiqué ci-dessus par le début d'une autre section ou la fin de la déclaration de la classe. Les deux autres sections sont débutées par les mots réservés **private** ou **public**. Voici l'extrait de code utilisé dans le paragraphe précédent, commenté pour vous montrer le début et la fin des trois sections :

```
type
  TfmPrinc = class(TForm)
    { section débutée après class(...) }
    { section terminée par le début de la section "private" }
  private
    { section private débutée par le mot réservé "private" }
    { Private declarations }
    { section private terminée par le début de la section "public" }
  public
    { section public débutée par le mot réservé "public" }
    { Public declarations }
    { section public terminée par la fin de la classe (end) }
  end;
```

Le but de ces sections est de spécifier la visibilité des éléments qu'elles contiennent vis-à-vis de l'extérieur de la classe. La première section est réservée exclusivement à Delphi. Vous n'avez pas le droit d'y écrire quoi que ce soit vous-même. La section suivante débutée par le mot réservé **private** est nommée « section privée » de la classe : tout ce qui y est écrit est innaccessible depuis l'extérieur, il n'y a que depuis d'autres éléments de la classe que vous pouvez y accèder. Cette section est idéale pour mettre des champs dont la valeur ne doit pas être modifiée depuis l'extérieur (je vais expliquer tout cela plus en détail dans le paragraphe suivant). La troisième et dernière section est nommée « section publique » : tout ce qui y est écrit est accessible depuis l'extérieur. Lorsque par exemple vous pouvez utiliser une méthode d'un objet ou d'un composant, c'est que cette méthode est dans la section publique de la classe de ce composant. En ce qui concerne la section réservée à Delphi, tous les éléments qui y sont déclarés sont visibles comme s'ils étaient déclarés dans la section publique.
Pour bien comprendre l'intérêt de telles sections, considèrons un objet simulant une salle d'opération. La section publique comprendrait par exemple une méthode pour lancer telle opération. Par contre, la méthode destinée à manier le scalpel serait assurément privée, car une mauvaise utilisation aurait des conséquences fâcheuses. De même, un champ "Patient" serait public, pour que vous puissiez décider qui va subir l'opération, mais un champ "température" fixant la température de la salle d'opération serait privé, car c'est un paramètre sensible qui doit être manipulé avec précaution.

## XII-C-2-b - Ajout de méthodes et de variables

Nous en savons assez pour commencer à ajouter des éléments aux classes. Pour le moment, nous allons ajouter ces éléments dans la classe TForm1 descendante de TForm. L'intérêt d'ajouter des éléments à ces classes ne vous apparaît peut-être pas encore, mais cela viendra avec l'expérience.
Tout d'abord, le nom de cette classe (TForm1) est décidé dans l'inspecteur d'objets : changez le nom de la fiche (propriété "name") en "fmPrinc" et observez le changement : la classe s'appelle désormais "TfmPrinc". Ce nom est fabriqué en préfixant un "T" au nom que vous choisissez pour la fiche. L'objet qui est alors déclaré de classe TfmPrinc


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

est alors bien "fmPrinc", nom que vous avez choisi pour la fiche et qui est en fait, comme vous le saviez déjà, le nom de la variable qui stocke la fiche :

```
type
  TfmPrinc = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmPrinc: TfmPrinc;
```

Nous allons maintenant ajouter un champ nommé fTest de type Integer (le préfixe f ou F est souvent employé pour préfixer les identificateurs qui servent à nommer les champs des classes) à la classe TfmPrinc. Nous allons en outre faire cet ajout dans la section publique de la classe. Pour cela, complètez sous Delphi votre code en prenant comme modèle l'extrait suivant où la modification a été effectuée :

```
type
  TfmPrinc = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
    fTest: Integer;
  end;

var
  fmPrinc: TfmPrinc;
```

La classe TfmPrinc déclarant un champ fTest de type Integer, tous les objets de cette classe et des éventuelles classes descendantes de TfmPrinc contiendront une variable fTest. C'est le même principe avec les éléments rajoutés par Delphi tels les composants ou les méthodes associées aux événements. Ces éléments sont en fait déclarés dans la classe et sont donc accessibles dans l'objet qui est déclaré de cette classe. Posez donc un bouton nommé "btAction" sur la fiche et générez la procédure (i.e. la méthode) associée à son événement OnClick. La déclaration de la classe TfmPrinc est alors :

```
TfmPrinc = class(TForm)
  btAction: TButton;
  procedure btActionClick(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
  fTest: Integer;
end;
```

Vous remarquez que la méthode btActionClick et le champ btAction sont déclarés avant les sections privées et publiques, dans l'espace réservé à Delphi. Ces deux éléments, ainsi que le champ fTest, font partie de la classe TfmPrinc. De ce fait, la méthode btActionClick a accès non seulement à btAction puisque ce champ est public (il est déclaré dans la section réservée à Delphi, qui est analogue à la section publique en terme de visibilité), mais également au champ fTest puisque la méthode est à l'intérieur de la classe. Nous allons modifier la valeur du champ fTest et afficher cette valeur depuis la procédure pour démontrer cela. Utilisez l'extrait de code ci-dessous pour complèter votre procédure btActionClick :

```
procedure TfmPrinc.btActionClick(Sender: TObject);
begin
  fTest := 10;
  ShowMessage(IntToStr(fTest));
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;
```

Vous pouvez constater qu'on utilise fTest comme une propriété, mais ce n'en est pas une, et fTest ne peut pas non plus être en lecture seule.
Voilà, vous avez modifié une classe pour la première fois. Maintenant, il faut savoir que le champ que nous avons utilisé était très simple, mais que rien n'empêche d'utiliser des types comme les tableaux, les enregistrements, les pointeurs et les objets.
Nous allons maintenant ajouter une méthode à la classe TfmPrinc. Pour goûter à tout, nous allons la déclarer dans la section publique de la classe et déplacer fTest dans la section privée. La méthode, nommée SetTest, servira en quelque sorte d'intermédiaire entre l'extérieur de la classe (qui aura accès à la méthode mais pas au champ) et l'intérieur pour nous permettre de refuser certaines valeurs. Voici la nouvelle déclaration de la classe :

```
TfmPrinc = class(TForm)
  btAction: TButton;
  procedure btActionClick(Sender: TObject);
private
  { Private declarations }
  fTest: Integer;
public
  { Public declarations }
  procedure SetTest(Valeur: Integer);
end;
```

Maintenant que la méthode est déclarée, il va falloir écrire son code. Pour cela, il ne faut pas oublier que SetTest est une méthode interne à TfmPrinc et qu'il faudra, pour écrire son code source, lui donner son nom qualifié. Pour ceux qui ne se rappelent pas de ce que ça signifie, il faudra précéder le nom de la méthode par le nom de la classe et un point. Il va de soi que l'ensemble sera écrit dans la partie implémentation de l'unité. Voici donc le squelette du code de la méthode :

```
procedure TfmPrinc.SetTest(Valeur: Integer);
begin

end;
```

Pour les heureux posseseurs de Delphi 5, il n'est pas indispensable d'insèrer vous-même ce code : placez-vous sur la ligne de la déclaration de la méthode et utilisez le raccourci clavier Ctrl + Maj + C. Ceci aura pour effet d'écrire automatiquement le code présenté ci-dessus et de placer le curseur entre le **begin** et le **end**.
Nous allons complèter cette méthode. Etant donné qu'elle est interne à la classe TfmPrinc, elle a accès au champ fTest. Nous allons fixer fTest à la valeur transmise, uniquement si cette valeur est supérieure à la valeur actuelle. Voici le code à utiliser :

```
procedure TfmPrinc.SetTest(Valeur: Integer);
begin
  if Valeur >= fTest then
    fTest := Valeur;
end;
```

Afin de tester la méthode, nous allons l'appeler plusieurs fois avec diverses valeurs et afficher ensuite la valeur résultante de fTest. Voici un nouveau contenu pour la procédure btActionClick :

```
procedure TfmPrinc.btActionClick(Sender: TObject);
begin
  SetTest(10);
  SetTest(30);
  SetTest(20);
  ShowMessage(IntToStr(fTest));
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Le message résultant affiche bien évidemment 30. Ce n'est encore une fois pas le résultat qui compte. mais la méthode employée.
Comme vous pouvez le constater, modifier une classe existante n'est pas très compliqué. Lorsque l'on ajoute un élément à une classe, il faut cependant essayer de respecter une règle : on place en priorité les éléments dans la section privée, et on ne les place dans la section publique que lorsque c'est nécessaire.

## XII-C-3 - Paramètres de type objet

### XII-C-3-a - Explications générales

Maintement que vous avez de bonnes bases sur les objets et les classes, il est un sujet qu'il nous est enfin possible d'aborder. Ce sujet requièrerait d'ailleurs de plus grandes connaissances sur les objets, mais nous allons tenter de faire sans.
Comme le titre de ce paragraphe le suggère, il s'agit ici des paramètres - de procédures et de fonctions - dont les types sont des classes (ce qui implique donc que les paramètres eux-mêmes sont des objets). Voici immédiatement un premier exemple qui devrait vous être quelque peu familier :

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Cet extrait de code, comme vous vous en doutez (j'espère !), est le squelette d'une procédure associée à un événement OnClick. Cette procédure, qui fait, soit dit en passant partie de la définition de la classe "TForm1", a un trait particulier qui nous intéresse davantage ici : elle accepte un unique paramètre de type "TObject". Or "TObject" est un objet, ce qui fait que le paramètre appelé "Sender" est de type objet.
Cet exemple est toutefois peu intéressant car la classe "TObject" n'est pas très développée. Imaginons maintenant que nous ayons besoin de changer le texte d'un bouton sur une fiche, mais sans utiliser le nom de ce bouton. Ceci permettrait par exemple de changer le texte de n'importe quel bouton avec cette procédure. Nommons-là "ChangeTexte". Cette procédure doit agir sur un bouton, c'est-à-dire sur un objet de classe "TButton". Le premier paramètre de la procédure sera donc de cette classe. Nous allons également transmettre à la procédure le texte à écrire sur le bouton, ce qui se fera par un paramètre de type chaîne. Voici le squelette de cette procédure :

```
procedure ChangeTexte(Bt: TButton; NouvTxt: String);
begin

end;
```

Depuis cette procédure, "Bt" doit être considéré comme un véritable objet de classe "TButton", comme le paramètre "NouvTxt" qui est considéré comme une vraie chaîne de caractères. On a tout à fait le droit d'accèder à la propriété "Caption" de "Bt" puisque c'est un bouton. Nous allons donc écrire l'unique instruction de la procédure ainsi :

```
procedure ChangeTexte(Bt: TButton; NouvTxt: String);
begin
  Bt.Caption := NouvTxt;
end;
```

Il faudra cependant faire très attention avec les paramètres de type objet, car contrairement aux types simples qui ont toujours une valeur, et similairement aux pointeurs qui peuvent valoir **nil**, les paramètres de type objet peuvent également ne pas être corrects, s'ils sont non construits par exemple. Un bon moyen pour savoir si un objet est valide est d'utiliser la fonction "Assigned". Cette fonction particulière qui accepte un peu n'importe quoi comme paramètre (pointeurs et objets sont acceptés) renvoie un booléen renseignant sur l'état du paramètre : faux indique pour un pointeur qu'il vaut **nil**, et pour un objet qu'il est invalide, vrai indique que le pointeur n'est pas **nil**, ou qu'un objet est correctement construit.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Nous éviterons donc des erreurs en modifiant la procédure "ChangeTexte" de cette manière :

```
procedure ChangeTexte(Bt: TButton; NouvTxt: String);
begin
  if Assigned(Bt) then
    Bt.Caption := NouvTxt;
end;
```

Avec cet exemple, vous voyez que l'on manipule les paramètres de type objet comme les autres paramètres, hormis le fait qu'on prend un peu plus de précaution en les manipulant, ce qui est tout à fait compréhensible vu la complexité des classes. Nous allons maintenant voir que cette apparente simplicité cache en fait bien plus.

## XII-C-3-b - Envoi de paramètres de classes différentes

Dans l'exemple de la procédure "Button1Click" ci-dessus, on serait en droit de croire que "Sender" est alors de classe "TObject", ce qui serait tout à fait légitime vu ce qui précède dans tout ce chapitre. En fait, ce n'est pas toujours le cas. En réalité, "Sender" peut effectivement être de classe "TObject", mais également de toute classe descendante de la classe "TObject". Ainsi, "Sender" peut très bien être de classe "TComponent" ou "TButton" par exemple, mais aussi "TObject". Ceci signifie qu'on peut transmettre en tant que paramètre de type classe n'importe quel objet dont la classe est ou descend de la classe du paramètre dans la déclaration de la procédure/fonction.
Pas d'affolement, je m'explique : considérons une procédure "Toto" qui a un paramètre "Obj" de classe "TMachin". On peut alors appeler "Toto" en donnant comme paramètre soit un objet de classe "TMachin", soit un objet dont la classe descend de "TMachin". Par exemple, si la classe "TSuperMachin" descend de "TMachin", alors tout objet de classe "TSuperMachin" peut être transmis en tant que paramètre de "Toto". Voici une déclaration possible pour "Toto" :

```
procedure Toto(Obj: TMachin);
```

Mais cette liberté dans le choix des classes des paramètres objets a une limite : à l'intérieur de la procédure, un paramètre de classe "TMachin" n'est plus de classe "TSuperMachin". Ceci signifie que même si vous transmettez un paramètre de classe "TSuperMachin" à la procédure "Toto", cette dernière considérera que votre paramètre est de type "TMachin", et perdra donc tout ce qui a été ajouté à "TMachin" pour en faire "TSuperMachin".
Imaginons par exemple qu'une méthode "SuperMethode" soit ajoutée à "TSuperMachin" (elle ne figure pas dans "TMachin", nous supposerons également pour faire simple que cette méthode n'a pas de paramètres). Supposons aussi que nous avons un objet "SuperObjTest" de classe "TSuperMachin". Depuis l'extérieur de la procédure "Toto", vous pouvez appeler "SuperMethode" (à condition toutefois que cette méthode soit dans la section publique de la classe "TSuperMachin") de la manière habituelle, à savoir :

```
SuperObjTest.SuperMethode;
```

Depuis l'intérieur de la procédure "Toto", "Obj" est considéré de classe "TMachin", c'est-à-dire que si vous écrivez ce qui est présenté ci-dessous, vous aurez une erreur :

```
procedure Toto(Obj: TMachin);
begin
  Obj.SuperMethode; { <-- Ceci est interdit !!! }
end;
```

Pourquoi une erreur ? Simplement parce que la méthode "SuperMethode" n'est pas définie au niveau de la classe "TMachin", mais seulement par l'une de ses classes descendantes. Un objet de classe "TMachin" (Obj) ne possède pas de méthode "SuperMethode", et donc une erreur est affichée en vous indiquant que la méthode "SuperMachin" est inconnue.
Ceci termine la partie cours de ce premier chapitre consacré aux objets. Le prochain chapitre va vous permettre de respirer un peu car le sujet évoqué, les fichiers, bien qu'important, est à mon sens un sujet plus abordable que les objets.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XIII - Utilisation des fichiers

Lorsqu'on commence à écrire des applications sérieuses, il apparaît rapidement un besoin : stocker des informations de façon permanente, que ce soit de façon interne sans intervention de l'utilisateur, ou de façon externe pour permettre par exemple à l'utilisateur d'effectuer une sauvegarde des données éditées dans l'application. Le seul moyen envisageable, mis à part l'utilisation d'une base de données (ce qui revient finalement au même), est l'utilisation de fichiers.
Les fichiers sont universellement utilisés pour stocker des informations, et leur utilisation est permise de plusieurs manières par le langage Pascal Objet et Delphi. Ce chapitre va tout d'abord vous inculquer quelques principes pour l'utilisation des fichiers, puis les divers moyens d'accès aux fichiers seront étudiés.

## XIII-A - Introduction : Différents types de fichiers

Cette partie décrit les différents types de fichiers que vous serez amenés à utiliser. La liste présentée ici n'a rien à voir avec une liste de types de fichiers au sens où l'explorateur Windows l'entend, mais indique plutôt les différents types de structures possibles pour le contenu des fichiers.

### XIII-A-1 - Fichiers texte

Les fichiers texte sont certainement les fichiers les plus simples à comprendre et à utiliser en Pascal. Ces fichiers comportent du texte brut, comme vous en lisez régulièrment dans le bloc-notes de Windows par exemple. Le texte contenu dans ce genre de fichiers est réparti en lignes. Pour information, chaque ligne se termine, comme dans tous les fichiers texte, par un marqueur de fin de ligne qui est constitué d'un ou de deux caractères spéciaux : le retour chariot et éventuellement le saut de ligne (Sous les environnements Windows, les deux caractères sont généralements présents, contrairement aux environnements de type UNIX qui favorisent le retour chariot seul).
Ces fichiers sont adaptés à des tâches très utiles :

- stocker du texte simple entré par l'utilisateur dans l'application par exemple
- écrire des fichiers de configuration (entre autre des fichiers INI)
- générer des fichiers lisibles par d'autres applications, comme des fichiers HTML (pages web), RTF (texte mis en page) ou CSV (texte réparti en colonnes).

Ce genre de fichier sera étudié en premier, car sa méthode d'utilisation est certainement l'une des plus simples parmi les méthodes d'accès aux fichiers.

### XIII-A-2 - Fichiers séquentiels

Les fichiers séquentiels partent sur un principe différent des fichiers texte. Leur principe est de stocker séquentiellement (à la suite) un certain nombre d' « enregistrements » d'un type donné, et ce de façon automatisée en ce qui concerne la lecture et l'écriture. Tous les types de données ne peuvent pas être stockés dans ce type de fichier : classes, pointeurs ou tableaux dynamiques, entre autres, ne seront pas au programme.
Ce genre de fichier est surtout utilisé pour stocker une liste d'éléments de type enregistrement (record) : à condition de n'avoir que des sous-éléments admis par le type de fichier séquentiel, il sera possible de lire et d'écrire avec une facilité déconcertante des enregistrements depuis les fichiers séquentiels.
Leur facilité d'utilisation avec les enregistrement est leur point fort, par contre, leur principale faiblesse est de ne procurrer aucune liberté concernant ce qui est concrètement écrit dans les fichiers, ou de ne pas pouvoir utiliser de pointeurs ou de tableaux dynamiques, souvent très utiles.

### XIII-A-3 - Fichiers binaires

Ces fichiers sont aussi nommés "fichier non typé" dans l'aide de Delphi, on prefèrera ici la dénomination générale de "fichier binaire". Ce dernier genre de fichier est en fait un genre universel, car tous les types de fichiers, sans


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

exception, peuvent être créés avec ce genre de fichier : vous êtes libres à 100% du contenu du fichier, ce qui vous permet d'élaborer une structure personnelle qui peut être très évoluée. Mais cela a un prix : vous n'avez aucune des facilités offertes en lecture/écriture par les fichiers texte ou les fichiers séquentiels.
Ce genre de fichier est très souvent utilisé, car il n'est pas rare qu'on ait besoin d'élaborer sa propre structure de fichier, pour stocker des données non supportées par les fichiers séquentiels par exemple. Imaginez un enregistrement comportant un tableau dynamique de pointeurs vers des données. Le type de fichier séquentiel ne peut pas convenir ici : vous devrez effectuer manuellement les écritures et les lectures, mais c'est à ce prix que vous êtes libres du contenu des fichiers que vous manipulerez.

## XIII-B - Manipulation des fichiers texte

Nous commençons comme promis par les fichiers texte. C'est le type le plus adapté pour débuter car il est très simple de vérifier les manipulations. Pour cela, il suffit d'ouvrir le fichier créé pendant la manipulation dans un éditeur de texte pour afficher son contenu. Il est à noter qu'il existe une période d'ombre entre l'ouverture et la fermeture d'un fichier, pendant laquelle le fichier ne peut pas être affiché, car il est monopolisé par votre application. Avant de détailler plus avant les moyens de tester vos résultats, voyons comment on accède à ces fichiers.

### XIII-B-1 - Ouverture et fermeture d'un fichier texte

***Note aux programmeurs Turbo Pascal :***
*Les fichiers texte, déjà utilisables dans Turbo Pascal, sont exploités de manière très similaire sous Delphi. La seule différence notable est que le type "Text" est remplacé par "TextFile". La procédure d'assignation "Assign" est remplacée par "AssignFile", et la procédure "Close" est remplacée par "CloseFile". "Reset", "Rewrite" et "Append" sont conservées.*
*Vous verrez par contre dans les autres paragraphes que Delphi apporte de nouvelles méthodes plus modernes pour l'accès aux fichiers comme l'utilisation des flux.*

Le langage Pascal Objet a sa manière bien à lui de manipuler les fichiers en général. Cette méthode, qui date de Turbo Pascal, est toujours employée sous Delphi, car elle a fait ses preuves. Un fichier texte sous Delphi se manipule par l'intermédiaire d'une variable de type "TextFile". La manipulation d'un fichier passe par 3 étapes obligatoires : 2 avant l'utilisation, et une après. Voici la déclaration de la variable fichier :

```
var
  FichTest: TextFile;
```

La première étape, avant utilisation, est l'étape d'assignation. Elle consiste à associer à la variable-fichier le nom du fichier que nous voulons manipuler. Cette assignation se fait par l'appel d'une procédure très simple nommée "AssignFile" qui admet deux paramètres. Le premier est la variable-fichier, le second est le nom du fichier que vous désirez manipuler. Noter que si vous voulez lire le contenu du fichier, il est bien évident que ce fichier doit exister.
Voici l'instruction qui assigne le fichier « c:\test.txt » à la variable-fichier "FichTest" déclarée ci-dessus (le nom du fichier est transmis dans une chaîne de caractères) :

```
AssignFile(FichTest, 'c:\test.txt');
```

La deuxième étape consiste à ouvrir le fichier. Cette ouverture peut se faire suivant trois modes différents suivant ce qu'on a besoin de faire pendant que le fichier est ouvert. La méthode qu'on emploie pour ouvrir le fichier détermine ce mode d'ouverture. Les trois modes possibles sont la lecture seule (écriture impossible), l'écriture seule (lecture impossible), soit simplement l'ajout du texte à la fin du fichier (ajout seul). Notez qu'il n'est pas possible de pouvoir lire et écrire à la fois dans un fichier texte.

- Si on souhaite lire, il faut utiliser la procédure Reset. Si le fichier n'existe pas, attendez-vous à une erreur de la part de votre application. L'écriture n'est pas possible avec ce mode d'ouverture.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Si on souhaite écrire, il faut utiliser la procédure Rewrite. Si le fichier existe, il sera écrasé. La lecture n'est pas possible avec ce mode d'ouverture.
- Si on souhaite juste ajouter du texte dans un fichier texte, il faut l'ouvrir en utilisant la procédure Append. Cette procédure ouvre le fichier et permet d'y écrire seulement. Ce que vous y écrirez sera ajouté à la fin du fichier.

Ces trois procédures s'utilisent de la même manière : on l'appelle en donnant comme unique paramètre la variable-fichier à ouvrir. Voici l'instruction qui commande l'ouverture du fichier c:\test.txt en lecture seule :

```
Reset(FichTest);
```

Pour ouvrir le même fichier en écriture seule, il suffirait de substituer "Rewrite" à "Reset". Venons-en tout de suite à la troisième étape obligatoire : la fermeture du fichier. Cette fois-ci, c'est plus simple, car la procédure "CloseFile" (qui accepte un seul paramètre de type variable-fichier) ferme un fichier ouvert indifférement avec "Reset", "Rewrite" ou "Append". Voici l'instruction qui ferme le fichier (qui doit avoir été ouvert, sous peine d'erreur dans le cas contraire).

```
CloseFile(FichTest);
```

Voici donc une procédure qui ouvre un fichier en lecture seule et le ferme aussitôt. Pour changer un peu des procédures associées aux événements, cette procédure est indépendante d'une quelconque fiche et accepte un paramètre qui est le fichier à ouvrir.

```
procedure OuvrirFichier(Fich: string);
var
  FichTest: TextFile;
begin
  AssignFile(FichTest, Fich);
  Reset(FichTest);
  { lecture possible dans le fichier ici }
  CloseFile(FichTest);
end;
```

Pour tester cette procédure, créez un fichier texte ou trouvez-en un qui existe, puis appelez la procédure en donnant l'emplacement du fichier en tant que chaîne de caractères. Par exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OuvrirFichier('c:\bootlog.txt');
end;
```

Il est possible d'améliorer nettement cette procédure d'ouverture, notamment en contrôlant l'existence du fichier avant son ouverture, car un fichier inexistant provoquerait une erreur génante qu'il est possible d'éviter facilement. On teste l'existence d'un fichier grâce à la fonction FileExists. Cette fonction admet en unique paramètre le nom du fichier. Le résultat de la fonction est un booléen qui indique l'existence du fichier (True si le fichier existe, False dans le cas contraire). Voici la nouvelle procédure, améliorée par l'utilisation de FileExists.

```
procedure OuvrirFichier(Fich: string);
var
  FichTest: TextFile;
begin
  if not FileExists(Fich) then exit;
  AssignFile(FichTest, Fich);
  Reset(FichTest);
  { lecture possible dans le fichier ici }
  CloseFile(FichTest);
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Comme vous le voyez, l'ajout est fort simple, puisque l'appel de "exit" dans le cas où le fichier n'existerait pas évite la tentative d'ouverture en sortant directement de la procédure. Venons-en à la lecture.

## XIII-B-2 - Lecture depuis un fichier texte

Avant de se lancer tête baissée dans la lecture du contenu d'un fichier texte, il faut apprendre quelques règles sur les mécanismes de lecture en Pascal. La lecture se fait au choix par la procédure Read ou Readln ("READ LiNe", "lire ligne" en anglais). La procédure Readln est la plus simple à utiliser car elle permet de lire une ligne entière d'un fichier (je rappelle qu'un fichier texte est découpé en lignes). La procédure Read, quant à elle, permet de lire de différentes manières : caractère par caractère, ou petit morceau par petit morceau. Cette dernière technique est à éviter car elle prend plus de temps que la lecture ligne par ligne, et tout ce que permet la procédure Read peut être réalisé sur le texte lu au lieu d'être réalisé sur le texte à lire. La lecture ligne par ligne sera donc la seule étudiée ici.
Lorsqu'on ouvre un fichier texte, la « position » dans ce fichier est fixée à son début. La « position » d'un fichier détermine à quel endroit aura lieu la prochaine opération sur ce fichier. Pour lire une ligne d'un fichier, on utilise donc Readln. Readln est une procédure très spéciale dans le sens où son nombre de paramètres est a priori indéterminé. Le premier paramètre est la variable-fichier dans laquelle on désire lire. Les paramètres suivants sont des variables permettant de stocker ce qui a été lu. Dans la pratique, on se limite très souvent à une seule variable de type chaîne. Voici l'instruction qui lit une ligne du fichier associé à la variable-fichier TestFile et qui la stocke dans la variable tmpS de type **string** :

```
Readln(FichTest, tmpS);
```

Il va de soi que pour que cette instruction fonctionne, il faut que le fichier soit ouvert. Readln lit une ligne depuis la « position » en cours dans le fichier, jusqu'aux caractères de fin de ligne (ceux-ci sont éliminés), puis fixe la nouvelle « position » au début de la ligne suivante (sous réserve qu'elle existe, nous allons parler de l'autre cas un peu plus bas). L'appel suivant à Readln lira la ligne suivante du fichier, et ainsi de suite jusqu'à ce qu'il n'y ait plus de ligne après la position dans le fichier, c'est-à-dire lorsqu'on a atteint la fin du fichier. Pour tester cette condition, on doit faire appel à une fonction nommée Eof ("End Of File", "Fin de fichier" en anglais). Cette fonction accepte en unique paramètre une variable-fichier et renvoie un booléen qui indique si la position de fichier est la fin de ce dernier, c'est-à-dire que lorsque Eof renvoie True, il n'y a plus de ligne à lire dans le fichier.
La plupart du temps, lorsqu'on lit l'ensemble des lignes d'un fichier texte, il faut utiliser une boucle **while**. Cette boucle permet de lire les lignes une par une et de tester à chaque fois si la fin du fichier est atteinte. Voici un exemple d'une telle boucle **while** :

```
while not Eof(FichTest) do
  Readln(FichTest, tmpS);
```

La boucle ci-dessus lit un fichier ligne par ligne jusqu'à la fin du fichier. Lorsque celle-ci est atteinte, Eof devient vrai et la condition de continuation de la boucle n'est plus respectée, et la lecture s'arrête donc d'elle-même. Voici notre procédure OuvrirFichier encore améliorée. Elle lit maintenant l'intégralité du fichier transmis :

```
procedure OuvrirFichier(Fich: string);
var
  FichTest: TextFile;
  tmpS: string;
begin
  if not FileExists(Fich) then exit;
  AssignFile(FichTest, Fich);
  Reset(FichTest);
  while not Eof(FichTest) do
    Readln(FichTest, tmpS);
  CloseFile(FichTest);
end;
```

Il serait tout de même intéressant de faire quelque chose de tout ce texte que l'on lit dans le fichier. Le plus simple est d'écrire tout ce texte dans un mémo. Placez donc un bouton btTest et un mémo meFich sur une fiche fmPrinc.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'ajout de lignes dans un mémo ayant déjà été étudié, nous ne réexpliqueront pas comme on fait. Pour donner un peu de piquant à cet exemple, nous allons nous intéresser d'un peu plus près au fichier étudié : BOOTLOG.TXT. Ce fichier, qui devrait exister à la racine de votre disque dur, résume ce qui s'est passé durant le dernier démarrage anormal de Windows. Ce fichier est un peu illisible par un humain, nous allons donc filtrer les lignes qui indiquent un problème. Ces lignes comportent pour la plupart la chaîne 'Fail' ('échouer' en anglais). Nous allons examiner chaque ligne et ne l'ajouter au mémo que lorsque la ligne contient 'Fail'.
Pour cela, il nous faudra la fonction Pos. Cette chaîne permet de rechercher une sous-chaîne dans une chaîne. Si la sous-chaîne est trouvée, la fonction renvoie la position du premier caractère de la sous-chaîne dans la chaîne (par exemple si on cherche 'our' dans 'bonjour tout le monde', le résultat sera 5, soit le numéro du premier caractère de 'our' : o. Cette procédure recherche une sous-chaîne exacte. Si on recherche 'Our' dans 'bonjour tout le monde', la fonction Pos renverra 0, indiquant par là que la sous-chaîne n'existe pas dans la chaîne. Pour rechercher sans s'occuper des minuscules/majuscules, l'idée est de faire la recherche dans des chaînes que l'on aura préalablement mise en majuscules.
Pour mettre une chaîne en majuscules, il suffit de la transmettre à la fonction UpperCase, qui renvoie la chaîne mise en majuscules. Voici donc la recherche de 'Fail' dans la chaîne lue, à savoir tmpS :

```
Pos('FAIL', UpperCase(tmpS))
```

Pour savoir si la ligne doit être ajoutée, il suffira de comparer le résultat de Pos à 0 et d'ajouter lorsque Pos sera strictement supérieur à 0. Voici la procédure LectureFichier qui fait la lecture complète, le filtrage et l'ajout dans le mémo :

```
procedure OuvrirFichier(Fich: string);
var
  FichTest: TextFile;
  tmpS: string;
begin
  if not FileExists(Fich) then exit;
  AssignFile(FichTest, Fich);
  Reset(FichTest);
  fmPrinc.meFich.Lines.Clear;
  while not Eof(FichTest) do
    begin
      Readln(FichTest, tmpS);
      if Pos('FAIL', Uppercase(tmpS)) < 0 then
        fmPrinc.meFich.Lines.Add(tmpS);
    end;
  CloseFile(FichTest);
end;
```

Voilà qui conclue ce petit paragraphe consacré à la lecture dans un fichier texte. Plutôt que de vous donner sèchement la procédure Readln, j'ai préféré vous démontrer une utilisation concrète, ce qui je pense est un bien meilleur choix.

## XIII-B-3 - Ecriture dans un fichier texte

L'écriture dans un fichier texte s'effectue également ligne par ligne grâce à la procédure Writeln, ou morceau par morceau avec la procédure Write. Contrairement à la lecture où l'utilisation de Read est à déconseiller, l'utilisation de Write est ici tout à fait possible, même si les temps d'écriture sont à prendre en considération : écrire un gros fichier ligne par ligne est plus rapide qu'en écrivant chaque ligne en plusieurs fois.
La procédure Writeln s'utilise comme Readln, mis à part qu'elle effectue une écriture à la place d'une lecture. L'écriture se fait à la position actuelle, qui est définie au début du fichier lors de son ouverture par "Rewrite", et à la fin par "Append". Writeln ajoute la chaîne transmise sur la ligne actuelle et insère les caractères de changement de ligne, passant la position dans le fichier à la ligne suivante. Ainsi, pour écrire tout un fichier, il suffira d'écrire ligne par ligne en appelant autant de fois Writeln qu'il sera nécessaire.
Voici une procédure qui ouvre un fichier "c:\test.txt" en tant que texte, et qui écrit deux lignes dans ce fichier. Après l'execution de la procédure, vous pourrez contrôler le résultat en ouvrant ce fichier dans le bloc-notes par exemple. Si vous venez à changer le nom du fichier, méfiez-vous car l'écriture ne teste pas l'existence du fichier et vous pouvez


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

donc écraser par mégarde un ficbier important. La même remarque reste valable pour chacun des fichiers que vous écrirez.

```
procedure TestEcriture(Fich: string);
var
  F: TextFile;
begin
  AssignFile(F, Fich);
  Rewrite(F);
  Writeln(F, 'ceci est votre premier fichier texte');
  Writeln(F, 'comme vous le constatez, rien de difficile ici');
  CloseFile(F);
end;
```

Dans l'exemple ci-dessus, le fichier texte est ouvert en écriture seule par Rewrite, puis deux écritures successives d'une ligne à chaque fois sont effectuées. Le fichier est ensuite normalement refermé. L'écriture, comme vous le constatez, est beaucoup plus simple que la lecture puisqu'on a pas à se soucier de la fin du fichier.
Voici pour conclure ce très court paragraphe un exercice typique, mais assez instructif. Contrairement à beaucoup d'exercices donnés jusqu'ici, vous êtes laissés très libres et vous vous sentirez peut-être un peu perdus. Pour vous rassurer, sachez que vous avez toutes les connaissances nécessaires à la réalisation de cet exercice, et qu'il est indispensable pour vous de pouvoir résoudre cet exercice sans aide. Comme d'habitude, vous pourrez cependant accèder à des indications et à un corrigé détaillé en fin de résolution.

Exercice 1 : (voir les **indications et le corrigé**)
Le but de cet exercice est de vous faire manipuler en même temps la lecture et l'écriture, et non plus les deux chacun de leur coté. L'exercice idéal pour cela est de réaliser une copie d'un fichier texte. Le but de l'exercice est donc l'écriture d'une fonction de copie dont voici la déclaration :

```
function CopyFichTexte(Src, Dest: string): boolean;
```

Cette fonction, qui ne sera utilisable qu'avec les fichiers texte, copiera le contenu du fichier texte dont le chemin est Src dans un nouveau fichier texte dont le chemin sera Dest. La fonction renverra faux si le fichier source n'existe pas ou si le fichier destination existe déjà (ce qui vous évitera d'écraser par accident un fichier important). Dans les autres cas, la fonction renverra vrai pour indiquer que la copie est effectuée. Notez que vous êtes libres de la méthode à adopter pour effectuer vos tests, mais que vous devez en faire un minimum.
Dans un deuxième temps, écrire une fonction similaire mais nommée CopyFichTexteDemi qui n'écrit qu'une ligne sur deux (en commencant par la première) du fichier source dans le fichier destination.

*Indication : utilisez une variable entière qui stockera le numéro de ligne qui vient d'être lu. L'écriture dépendra de la valeur de cette variable (notez qu'un nombre est impair lorsque le reste de sa division entière par deux est 1)*

Bon courage !

## XIII-B-4 - Utilisation des fichiers texte par les composants

Certains composants, dont le but avoué est la manipulation du texte, permettent également d'utiliser des fichiers texte. Ces composants proposent chacun deux méthodes : une pour lire un fichier texte et qui fait que le composant utilise ce texte, et une pour écrire le texte manipulé par le composant dans un fichier texte. La méthode de lecture s'appelle généralement LoadFromFile et accepte en paramètre le nom d'un fichier existant (une erreur est signalée si le fichier n'existe pas). Quand à la méthode d'écriture, elle est appelée SaveToFile et accepte également le nom d'un fichier qui est écrasé au besoin.
Ce genre de fonctionnalité très pratique pour sauver et charger facilement un texte est gérée par la classe "TStrings". "TStrings" est l'ancêtre immédiat de la classe "TStringList" que vous connaissez déjà, et est utilisé par de nombreux composants pour gérer de façon interne une liste de chaînes (n'utilisez pas vous-même "TStrings" mais "TStringList" pour gérer une liste de chaînes, sauf si vous savez vraiment ce que vous faites). Parmi les composants qui utilisent


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

des objets de classe "TStrings", on peut citer ceux de classe "TMemo" : la propriété "Lines" des composants Mémo, que nous disions jusqu'à présent de type « objet » sans préciser davantage est en fait de type "TStrings".
La classe "TStrings", seule à être étudiée ici, définit donc les deux méthodes LoadFromFile et SaveToFile. Pour preuve, placez un bouton "btCharg" et un mémo "meAffich" sur une fiche "fmPrinc" et utilisez le code ci-dessous :

```
procedure TfmPrinc.btChargClick(Sender: TObject);
begin
  meAffich.Lines.LoadFromFile('c:\bootlog.txt');
end;
```

Cette simple instruction appelle la méthode LoadFromFile de la propriété objet Lines (de type TStrings) du mémo meAffich. Le paramètre est une chaîne de caractères donnant l'emplacement du fichier à charger dans le mémo.
Si vous voulez maintenant permettre de sauver le contenu du mémo, vous pouvez créer un bouton "btSauve" et utiliser le code suivant (toujours dans la procédure associée à l'événement OnClick : c'est tellement habituel que je ne le préciserai plus) :

```
procedure TfmPrinc.btSauveClick(Sender: TObject);
begin
  meAffich.Lines.SaveToFile('c:\test.txt');
end;
```

Cette instruction permet de sauver le contenu du mémo dans le fichier 'c:\test.txt'. Dans l'avenir, je vous expliquerai comment afficher une des boites de dialogue (ouvrir, enregistrer, enregistrer sous) de Windows et comment configurer ces boites de dialogue. Vous pourrez alors permettre à l'utilisateur d'ouvrir un fichier de son choix ou d'enregistrer avec le nom qu'il choisira.
Ceci termine la section consacrée aux fichiers texte. Bien que très utiles, ces fichiers ne sont pas très souvent utilisés car ils sont tout simplement limités au texte seul. Dans la section suivante, nous allons étudier les fichiers séquentiels, qui sont adaptés à certains cas où l'on souhaite stocker une collection d'éléments de même type.

## XIII-C - Manipulation des fichiers séquentiels

### XIII-C-1 - Présentation

Les fichiers séquentiels sont une autre catégorie de fichiers qu'il est parfois très intéressant d'utiliser, quoique de moins en moins dans les versions successives de Delphi. Contrairement aux fichiers texte dont le but avoué est le stockage de texte seul, les fichiers séquentiels sont destinés à stocker un nombre quelconque de données du même type, à condition que le type soit acceptable.
Prenons par exemple le type de donnée suivant, qui ne devrait pas vous poser de problème particulier :

```
type
  TPersonne = record
    Nom,
    Prenom: string;
    Age: word;
    Homme: boolean; { vrai: homme, faux: femme }
  end;
```

L'objectif de ce type enregistrement est bien entendu de stocker les paramètre relatifs à une personne dans un répertoire par exemple. Imaginons maintenant que vous utilisez ce type pour créer un petit carnet d'adresses, vous aimeriez probablement proposer à vos utilisateurs l'enregistrement de leur carnet, sans quoi votre application ne serait pas vraiment intéressants. L'enregistrement de telles données, bien que techniquement possible en passant par un fichier texte, serait pénible avec ces derniers. La solution réside dans l'utilisation d'un fichier séquentiel, puisque ce dernier genre a été conçu spécifiquement pour répondre à ce genre de besoin du programmeur.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Un fichier séquentiel se définit en utilisant un type Pascal Objet qui repose sur un type plus fondamental qui désigne le type des éléments stockés dans ce fichier. Ce type s'écrit ainsi :

```
file of type_de_donnee
```

Il est possible de déclarer des variables de ce type ou d'écrire un type dédié. type_de_donnee est le type des données qui seront stockées dans le fichier. Dans notre exemple, nous allons ajouter un type "fichier de personnes" :

```
TPersFich = file of TPersonne;
```

Il suffira ensuite de déclarer une variable de type TPersFich pour pouvoir manipuler un fichier séquentiel. Nous verrons également comment lire et écrire des éléments de type TPersonne dans un tel fichier.
Mais ce qui nous importe à présent, c'est le message d'erreur que vous allez recevoir si vous entrez le code présenté ci-dessus dans Delphi : il semble que Delphi digère difficilement ce code, et ce pour une raison qui ne saute pas forcément aux yeux : le type TPersonne contient des données non acceptables dans un fichier séquentiel. Le coupable est le type "string" utilisé depuis Delphi 4. Ce type, qui permet désormais de manipuler des chaînes de plus de 255 caractères (contrairement aux anciennes versions de Delphi qui étaient limitées à 255 caractères pour le type string), n'est pas compatible avec les fichiers séquentiels car la taille d'une variable de type string est *dynamique* (susceptible de changer), ce qui n'est pas toléré dans le cadre des fichiers séquentiels. Pour utiliser des chaînes de caractères avec des fichiers séquentiels, il faudra soit fixer une longueur inférieure ou égale à 255 caractères pour la chaîne (en utilisant le type **string**[n]) ou utiliser l'ancien type "string" utilisé dans Delphi 3, qui a été rebaptisé "shortstring" ("chaîne courte" en anglais) (notez que cela équivaut à utiliser le type **string**[255]. Voici donc notre type TPersonne revu et corrigé :

```
type
  TPersonne = record
    Nom,
    Prenom: string[100];
    Age: word;
    Homme: boolean; { vrai: homme, faux: femme }
  end;
  TPersFich = file of TPersonne;
```

Encore une précision technique : la taille du fichier sera proportionnelle aux tailles des variables qui y seront placées. Les chaînes de 255 caractères, bien qu'autorisant plus de caractères, gâcheront beaucoup d'espace disque. Songez qu'en mettant 100 au lieu de 255 dans le type ci-dessus, on économise 155 x 2 = 310 octets par élément stocké. Sur un fichier de 100 éléments, ce qui est assez peu, on atteint déjà 31000 octets, ce qui est peut-être négligeable sur un disque dur, mais qui doit tout de même être pris en considération.
Avant de passer aux opérations de lecture/écriture sur les fichiers séquentiels, nous allons rapidement voir comment ouvrir et fermer ces fichiers.

## XIII-C-2 - Ouverture et fermeture d'un fichier séquentiel

Tout comme les fichiers texte, les fichiers séquentiels doivent être ouverts et fermés respectivement avant et après leur utilisation. L'étape initialise qui consiste à associer un nom de fichier à une variable qui désigne le fichier est elle aussi au menu. La bonne nouvelle, cependant, est que mis à part la déclaration des types fichiers séquentiels différente de celle des fichiers texte, les trois opérations d'assignation, lecture et écriture sont identiques entre les deux types de fichiers.
Les utilisations et les effets de Reset et de Rewrite sont cependant légèrement modifiés avec les fichiers séquentiels. Au niveau de l'utilisation, Reset doit être utilisé lorsque le fichier existe, et Rewrite lorsqu'il n'existe pas. L'effet produit par ces deux commandes est identique pour les fichiers séquentiels : on peut lire et écrire à la fois dans un fichier séquentiel ouvert. Voici une procédure qui manipule un fichier de personnes : le fichier est ouvert avec Reset ou Rewrite selon les cas et refermé aussitôt :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
procedure ManipFichierSeq(FName: string);
var
  Fichier: TPersFich;
begin
  AssignFile(Fichier, FName);
  if FileExists(FName) then
    Reset(Fichier)
  else
    Rewrite(Fichier);
  { manipulation possible ici }
  CloseFile(Fichier);
end;
```

Deux choses sont à remarquer dans cette procédure. Tout d'abord, la variable "Fichier" est de type TPersFich, ce qui en fait un fichier séquentiel contenant des personnes : des éléments de type TPersonne. Ensuite, les méthodes d'ouverture et de fermeture ressemblent à s'y méprendre à ce qu'on fait avec les fichiers texte, et il faudra se méfier de cette ressemblance car "Fichier" est ouvert ici en lecture/écriture quelle que soit la méthode d'ouverture.

## XIII-C-3 - Lecture et écriture depuis un fichier séquentiel

Ce paragraphe sera en fait plus que l'étude de la lecture et de l'écriture, car le fait que les fichier séquentiels soient ouverts en lecture/écriture offre de nouvelles possibilités. Parmi ces possibilité, celle de se déplacer dans le fichier et de le tronquer sont parfois des plus utiles.
La lecture dans un fichier séquentiel passe par la procédure standard Read. L'écriture, elle se fait par Write. Hors de question ici d'utiliser Readln et Writeln car les fichiers séquentiels ne sont pas organisés en lignes mais en *enregistrements*. Voyons tout d'abord l'écriture :

```
procedure ManipFichierSeq(FName: string);
var
  Fichier: TPersFich;
begin
  AssignFile(Fichier, FName);
  if FileExists(FName) then
    Reset(Fichier)
  else
    Rewrite(Fichier);
  Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Jean';
  Pers.Age := 39;
  Pers.Homme := True;
  Write(Fichier, Pers);
  Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Marie';
  Pers.Age := 37;
  Pers.Homme := False;
  Write(Fichier, Pers);
  CloseFile(Fichier);
end;
```

Cette procédure écrit deux éléments de type TPersonne dans le fichier représenté par la variable "Fichier". Vous voyez qu'après avoir rempli un élément de type TPersonne, on le transmet directement en paramètre de la procédure Write. Testez cette procédure en l'appelant lors d'un clic sur un bouton (un simple appel de ManipFichierSeq avec un nom de fichier inexistant en paramètre suffit, mais vous devenez trop fort pour que je vous fasse désormais l'affront de vous donner le code source), et allez ensuite examiner la taille du fichier résultant : vous devriez avoir un fichier de 412 octets (ou une taille très proche). Pour mettre en évidence une faiblesse de notre test, mettez la deuxième opération d'écriture ainsi que les 4 affectations précédentes en commentaires.

```
procedure ManipFichierSeq(FName: string);
var
  Fichier: TPersFich;
begin
  AssignFile(Fichier, FName);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  if FileExists(FName) then
    Reset(Fichier)
  else
    Rewrite(Fichier);
  Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Jean';
  Pers.Age := 39;
  Pers.Homme := True;
  Write(Fichier, Pers);
  { Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Marie';
  Pers.Age := 37;
  Pers.Homme := False;
  Write(Fichier, Pers); }
  CloseFile(Fichier);
end;
```

Relancez la procédure. Retournez voir la taille du fichier : 412 octets alors que vous n'écrivez plus qu'une fois au lieu de deux ? Es-ce bien normal ? La réponse est bien entendu non. En fait, lorsque vous ouvrez un fichier séquentiel existant, il faut savoir que on contenu n'est pas effacé : vous vous contentez d'écraser les parties que vous écrivez dans le fichier. Si vous n'écrivez pas autant d'éléments qu'il en contenait auparavant, les derniers éléments restent inchangés dans le fichier.
Pour résoudre cet indélicat problème, il suffit de faire appel à la procédure Truncate. Cette procédure, qui accepte en paramètre une variable fichier séquentiel, tronque le fichier à partir de la position en cours dans le fichier. Cette position, qui correspond à une position comptée en enregistrements, est déterminée par la dernière opération effectuée sur le fichier :

- Après l'ouverture, la position est le début du fichier, soit la position 0.
- Chaque lecture et écriture passe la position actuelle après l'enregistrement lu ou écrit, ce qui a pour effet d'augmenter la position de 1.
- Il est aussi possible de fixer la position actuelle en faisant appel à Seek. Seek accepte un premier paramètre de type fichier et un second qui est la position à atteindre. Si cette position n'existe pas, une erreur se produira à coup sûr.

Voici notre procédure améliorée par un appel à Truncate :

```
procedure ManipFichierSeq(FName: string);
var
  Fichier: TPersFich;
begin
  AssignFile(Fichier, FName);
  if FileExists(FName) then
    Reset(Fichier)
  else
    Rewrite(Fichier);
  Seek(Fichier, 0);
  Truncate(Fichier);
  Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Jean';
  Pers.Age := 39;
  Pers.Homme := True;
  Write(Fichier, Pers);
  Pers.Nom := 'DUPONT';
  Pers.Prenom := 'Marie';
  Pers.Age := 37;
  Pers.Homme := False;
  Write(Fichier, Pers);
  CloseFile(Fichier);
end;
```

Ici, l'appel de Seek est inutile, il n'est fait que pour vous démontrer cette possibilité. Le simple appel de Truncate après l'ouverture permet de tronquer entièrement le fichier. les écritures se font ensuite en ajoutant des éléments au


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

fichier et non plus seulement en les remplacant. Lorsque vous n'aurez aucune intention de lire le contenu d'un fichier séquentiel, fixez la position dans ce fichier à 0 :

```
Seek(Fichier, 0);
```

et tronquez le fichier : vous aurez ainsi l'assurance d'écrire sans patauger dans d'anciennes données parasites. Une autre solution est de lancer Truncate après la dernière écriture, pour vider ce qui peut rester d'indésirable dans le fichier qui vient d'être écrit. Vous pourrez ensuite le fermer en étant sûr que seul ce que vous venez d'y mettre y sera écrit.
La lecture dans un fichier séquentiel se fait sur le même principe que pour les fichiers texte : on lit à partir d'une position de départ et ce jusqu'à être rendu à la fin du fichier. Relancez une dernière fois la procédure qui écrit les deux enregistrements, puis effacez les affectations et les écritures. Nous allons effectuer une lecture *séquentielle* du fichier que vous venez d'écrire, c'est-à-dire que nous allons lire les éléments un par un jusqu'à la fin du fichier. Pour cela, une boucle **while** est tout indiquée, avec la même condition que pour les fichiers texte. A chaque itération, on lira un élément en affichant le résultat de la lecture dans un message. Voici :

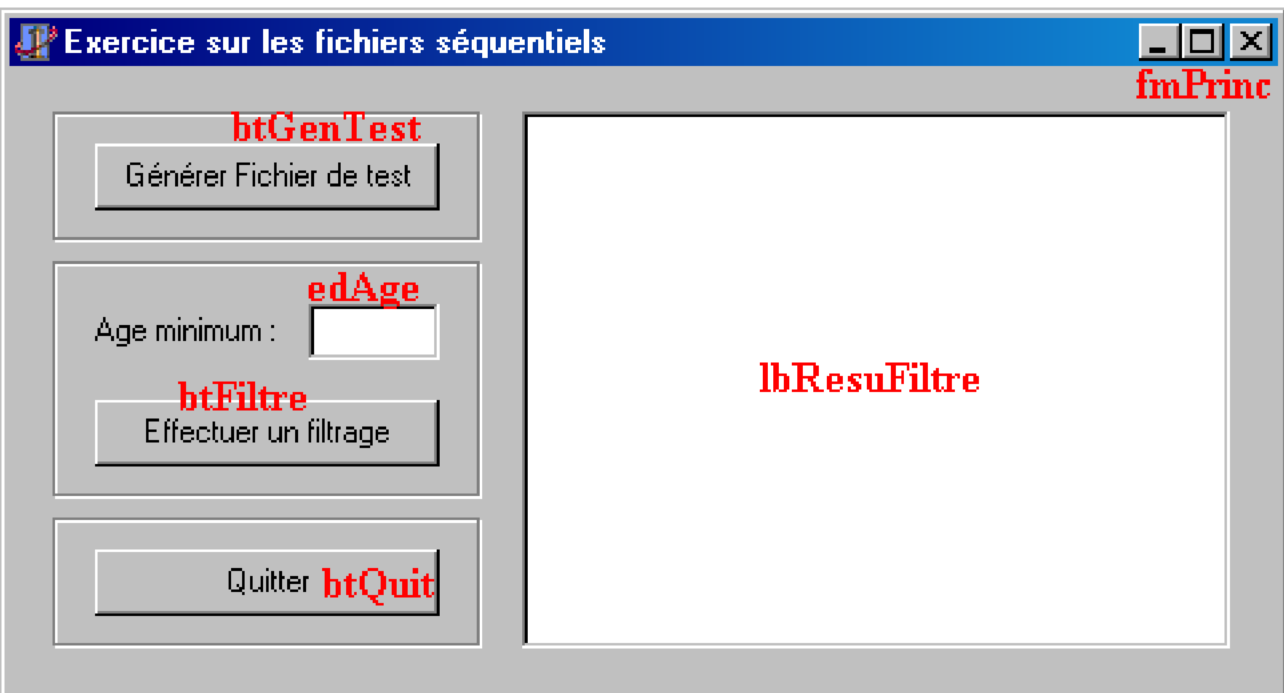
```
procedure ManipFichierSeq(FName: string);
var
  Fichier: TPersFich;
  Pers: TPersonne;
begin
  AssignFile(Fichier, FName);
  if FileExists(FName) then
    Reset(Fichier)
  else
    Rewrite(Fichier);
  Seek(Fichier, 0);
  while not Eof(Fichier) do
    begin
      Read(Fichier, Pers);
      ShowMessage(Pers.Nom + ' ' + Pers.Prenom + ' (' +
        IntToStr(Pers.Age) + ' ans)');
    end;
  CloseFile(Fichier);
end;
```

Exercice 2 : (voir la **solution, les commentaires et le téléchargement**)
Pour manipuler un peu plus les fichiers séquentiels par vous-même, tout en ne vous limitant pas à de simples lectures/écritures sans rien autour, je vais vous proposer un gros exercice d'application directe de ce qui vient d'être vu. Le principe est de programmer une petite application (vous pouvez considèrer que c'est un mini-projet guidé qui est proposé ici). Le principe est de générer un fichier de personnes (prénoms, noms et âges pris au hasard). L'application pourra ensuite effectuer un « filtrage » du fichier en affichant dans une zone de liste les personnes du fichier dont l'âge est supérieur à une valeur donnée par l'utilisateur. Cet exercice est assez long à réaliser, alors prévoyez un peu de temps libre (disons au moins une heure), et résolvez les questions suivantes dans l'ordre (vous pouvez consulter le **corrigé** qui donne la solution de chaque question) :

1. le programme aura avant tout besoin de générer aléatoirement des personnes (des éléments de type TPersonne). Le moyen que nous retiendrons sera l'utilisation de tableaux comprenant les noms et les prénoms. Un "random" permettra de choisir un indice au hasard dans chaque tableau, et donc un prénom et un nom indépendants. L'âge sera à générer directement avec un random, et le sexe devra être déterminé par exemple par un tableau de booléens fixés en fonction des prénoms. Créez donc la fonction GenerePersonneAlea sans paramètres et qui renvoit un résultat aléatoire de type TPersonne.
2. Ecrivez maintenant la procédure "GenereFichierSeq" qui génére le fichier séquentiel. Cette procédure acceptera en paramètre le nombre de personnes à mettre dans le fichier. La procédure se permettra exceptionnellement d'écraser un éventuel fichier existant. Pour nous mettre d'accord, le fichier sera nommé 'c:\test.txt' (ce nom sera fixé par une constante). Il va de soi que les personnes écrites dans le fichier doivent être générées aléatoirement (à l'aide de la fonction écrite à la question précédente).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

3 Ecrivez la fonction "AffichePersone" qui permet d'afficher les renseignements sur une personne : Nom, Prenom, Age et Sexe. La fonction créera une simple chaîne de caractères qui sera renvoyée en résultat.
4 Ecrivez la procédure "TraiteFichierSeq" qui permettra le traitement du fichier précédemment généré. Cette procédure acceptera deux paramètres. Le premier sera de classe "TListBox" (composant zone de liste) dans lequel seront ajoutées les personnes dont l'âge sera supérieur au deuxième paramètre (type entier). La zone de liste sera vidée en début de traitement.
5 Il est maintenant temps de créer une interface pour notre application. Inspirez-vous de la capture d'écran ci-dessous pour créer la vôtre. Les noms des composants ont été ajoutés en rouge vif.




N'oubliez pas que la fenêtre ne doit pas être redimensionnable, et programmez le bouton "Quitter".
6 Programmez le bouton "Générer fichier test" en appelant la procédure "GenereFichierSeq" avec 100 comme paramètre. Programmez également le bouton "Effectuer un filtrage" en appelant la procédure "TraiteFichierSeq". Le premier paramètre sera la seule zone de liste de la fiche, et le second sera obtenu depuis le texte de la zone d'édition "edAge". Le filtrage se terminera en afficbant un message qui indiquera le nombre de personnes affichées dans la liste.

Une fois toutes ces étapes réalisées, il ne vous reste plus qu'à tester le projet : il faudra d'abord générer le fichier de test, puis donner une valeur d'âge minimum, et tester. Vous pouvez ensuite changer de valeur d'âge ou regénérer le fichier test pour obtenir des résultats différents.
C'est la fin de cet exercice, et aussi la fin du paragraphe consacré aux fichiers séquentiels.

## XIII-D - Fichiers binaires

Les fichiers binaires, dont le nom vient du fait que leur structure est arbitrairement décidée par le programmeur, représentent la forme de fichier la plus couramment employée, du fait de la liberté totale qui vous est laissée quant au contenu. Contrairement aux fichiers séquentiels, les fichiers binaires ne sont pas organisés systmatiquement en lignes ou en enregistrements, mais peuvent contenir tout et n'importe quoi. Ainsi, vous pourrez élaborer une structure de donnée adaptée à une application et la mettre en oeuvre grâce aux fichiers binaires.
Cette liberté de mouvement a toutefois un prix que vous finirez par accepter : les lectures et les écritures sont plus sensibles. Il nous faudra ici faire de nombreux tests, car de nombreux traitements faits dans notre dos avec les fichiers textes ou séquentiels ne le sont plus avec les fichiers binaires. On peut dire que vous rentrez de plein fouet dans la vraie programmation en utilisant les fichiers binaires.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Dans ce paragraphe, une seule méthode d'utilisation des fichiers binaires va être vue : l'utilisation des instructions standards de Pascal pour la gestion des fichiers binaires. Une autre méthode qui utilise un objet de classe "TFileStream" et qui facilite la gestion d'un fichier binaire sera vue dans un prochain chapitre.

*Note aux initiés : Les méthodes présentées ici sont très perfectibles. En effet, il serait très judicieux ici d'employer des structures "try" pour vérifier que les fichiers ouverts sont systématiquement refermés. Il n'est pas ici question d'assomer le lecteur par une quirielle de détails techniques et de vérifications, mais de lui montrer une méthode de gestion d'un fichier binaire.*

## XIII-D-1 - Présentation

La méthode qui consiste à gérer les fichiers binaires de façon directe est à mon sens la méthode « brutale ». En effet, vous verrez plus tard que la classe "TFileStream" facilite tout ce que nous allons faire ici, en apportant quelques difficultés du fait de sa nature de classe. Cependant, cette classe masque beaucoup d'aspects importants des accès aux fichiers que le futur programmeur ne pourra ignorer, car si vous désirez un jour programmer en C, il vous faudra affronter des méthodes encore plus barbares, et donc commencer en Pascal est un bon compromis.
Nous allons débuter par un petit cours technique sur la capture et la gestion des erreurs, et nous poursuivrons ensuite avec la gestion des fichiers binaires. Au fil des paragraphes, quelques notions annexes vous seront présentées, car elles sont indispensables à la bonne utilisation des fichiers binaires. S'ensuivra une description détaillée de la lecture et de l'écriture de données de divers types dans un fichier binaire, le tout assorti de quelques exemples.
L'écriture d'un fichier binaire ne pose en général pas de problèmes. Par contre, la lecture impose de connaître la structure logique du fichier, c'est-à-dire que vous devrez savoir au fur et à mesure de la lecture ce que signifie les octets que vous lirez, car rien n'indiquera dans le fichier ce qu'ils peuvent signifier. Ainsi, l'utilisation des fichiers binaires impose auparavant de mettre noir sur blanc la structure du fichier qu'on va générer. Nous verrons que le plus simple moyen de créer un fichier complexe est d'utiliser des blocs, c'est-à-dire des suites d'octets dont le début indique ce que la suite signifie, et éventuellement la taille de la suite. Nous verrons que cette construction peut devenir passionnante, pour peu qu'on ait un minimum de patience et une certaine capacité d'abstraction.

## XIII-D-2 - Capture des erreurs d'entrée-sortie

Je vais me permettre un petit discours technique afin d'introduire ce qui suit. Pascal, ou plutôt Pascal Objet est un langage compilé, c'est-à-dire que le texte que vous écrivez passe par un logiciel spécial appelé compilateur qui fabrique un fichier éxécutable à partir de tout votre code Pascal Objet. Ce compilateur, comme la plupart des bons compilateurs, accepte une sorte de sous-langage qui lui est spécifiquement destiné. Ce langage comporte des commandes incluses dans le texte Pascal Objet, mais traitées à part. Ces commandes sont connues sous le terme de « directives de compilation ». Ces directives indiquent qu'on souhaite un comportement particulier de sa part lors de la compilation du code ou d'une partie du code.
Nous allons nous intéresser ici à une seule directive de compilation, car c'est celle qui nous permettra de détecter les erreurs provoquées par nos accès aux fichiers. Une directive de compilation est de la forme :

```
{$<directive>} .
```

C'est un commentaire, dont l'intérieur commence par un signe $, se poursuit par une directive. Dans notre cas, la directive est tout simplement I, suivi d'un + ou d'un -. Les deux directives que nous allons donc utiliser sont :

```
{$I+} et {$I-} .
```

Vous vous dites : "c'est bien beau tout ça, mais à quoi ça sert ?". La réponse a déjà été donnée en partie précédemment : lorsque nous manipulons un fichier, chaque commande d'ouverture (Reset et Rewrite), de déplacement (Seek), d'effacement (Truncate), de lecture (Read), d'écriture (Write) ou de fermeture (CloseFile) est susceptible de provoquer une erreur dite d' « entrée-sortie ». Dans le cas général, lorsqu'une telle erreur se produit, un message d'erreur s'affiche, indiquant à l'utilisateur la nature franchement barbare de l'erreur, dans un langage également des plus barbares. Imaginons maintenant que nous souhaitions intercepter ces erreurs pour y réagir nous-


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

mêmes, et ainsi au moins empècher l'affichage du message barbare. Pour cela, il faudra désactiver temporairement la gestion automatique des erreurs d'entrées-sorties, et la réactiver plus tard. Dans ce laps de « temps », la gestion des erreurs d'entrées-sorties nous incombe.
Rassurez-vous toutefois, la gestion des erreurs, à notre niveau, sera des plus simple. Il passe par l'appel de la fonction "IOResult", qui ne peut être utilisée que lorsque la gestion automatique des erreurs aura été temporairement désactivée par une directive {$I-} (il faudra penser à réactiver cette gestion automatique avec une directive {$I+}). Lorsque la gestion automatique est désactivée, et qu'une erreur se produit, un indicateur interne est modifié, empêchant toutes les instructions d'entrée-sortie suivantes de s'éxexuter. Pour connaître la valeur de cet indicateur à le remettre à 0 (qui signifie « pas d'erreur »), un simple appel à "IOResult" sera nécessaire après chaque instruction d'entrée-sortie.
Voici un exemple d'ouverture et de fermeture d'un fichier. Le procédé a été sécurisé au maximum avec des appels à "IOResult". Nous allons expliquer le déroulement de cette procédure ci-dessous :

```
function OuvrirFichier(NomFich: String): Boolean;
var
  F: File;
begin { retourne vrai si tout s'est bien passé }
  {$I-} { permet d'utiliser IOResult }
  AssignFile(F, NomFich);
  { ouverture }
  if FileExists(NomFich) then
    Reset(F, 1)
  else
    Rewrite(F, 1);
  Result := (IOResult = 0);
  if not Result then exit;
  { fermeture }
  CloseFile(F);
  Result := (IOResult = 0);
  if not Result then exit;
  {$I+} { réactive la gestion automatique des erreurs }
end;
```

La longueur de cet extrait de code, qui ne fait pas grand chose, peut effrayer, mais c'est à ce prix qu'on sécurise un programme qui autrement pourrait facilement générer des erreurs. Avec la version ci-dessus, aucune erreur ne sera signalée, quoi qu'il arrive. La procédure se termine simplement en cas d'erreur. Voyons comment tout cela fonctionne : tout d'abord un {$I-} permet d'utiliser IOResult. Ensuite, chaque instruction susceptible de provoquer une erreur est suivie d'un appel à IOResult. Cet appel suffit à réinitialiser un indicateur interne qui empêcherait autrement les autres instructions d'accès aux fichiers de s'éxécuter. Le résultat de IOResult est comparé à 0, de sorte que si cette comparaison est fausse, ce qui signifie qu'une erreur s'est produite, alors la procédure se termine en passant par un {$I+} qui réactive la gestion automatique des erreurs d'entrée-sortie (on dit aussi simplement « erreurs d'I/O » par abus de langage)
Venons-en maintenant à la véritable description de cet exemple : vous voyez en l'examinant que la méthode d'ouverture d'un fichier binaire (on dit également « fichier non typé » en référence à la partie "of *type*" qui est absente après le mot réservé "file" dans la déclaration de tels fichiers) ressemble beaucoup à celle des fichiers manipulés jusqu'à présent. La seule différence notable se situe au niveau de l'instruction d'ouverture : "Reset" ou "Rewrite", selon le cas, a besoin d'un paramètre supplémentaire, dont la signification est assez obscure : c'est la taille d'un enregistrement ! En fait, ces enregistrements-là n'ont rien à voir avec ceux des fichiers séquentiels. Dans la pratique, on doit toujours utiliser 1, ce qui signifie : "lire et écrire octet par octet".

## XIII-D-3 - Lecture et écriture dans un fichier binaire

La lecture et l'écriture dans un fichier binaire est un sport bien plus intéressant que dans les types précédents de fichiers. En effet, il n'y a ici rien de fait pour vous faciliter la vie. Il va donc falloir apprendre quelques notions avant de vous lancer.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Pour lire et écrire dans un fichier binaire, on utilise les deux procédures "Blockread" et "BlockWrite". Voici leur déclaration, ce qui vous permettra d'avoir déjà un tout petit aperçu de leur non-convivialité (ne vous inquiètez pas si vous ne comprenez pas tout, c'est normal à ce stade, je vais donner toutes les explications nécessaires juste après) :

```
procedure BlockRead(var F: File; var Buf; Count: Integer;
  var AmtTransferred: Integer);
procedure BlockWrite(var f: File; var Buf; Count: Integer;
  var AmtTransferred: Integer);
```

## XIII-D-3-a - Paramètres variables

Avant de passer à la description, quelques explications s'imposent. Le mot **var** employé dans une déclaration de procédure/fonction signifie que lorsque vous donnez ce paramètre, ce que vous donnez doit être une variable, et non une constante. Par exemple, avec la procédure suivante :

```
procedure Increm(var A: integer);
begin
  A := A + 1;
end;
```

La procédure "Increm" a besoin d'un paramètre variable, c'est-à-dire que si vous avez une variable "V" de type integer, vous pouvez écrire l'instruction suivante, qui est correcte :

```
Increm(V);
```

Par contre, l'instruction suivante serait incorrecte car le paramètre n'est pas une variable :

```
Increm(30);
```

En fait, le fait de déclarer un paramètre variable fait que la procédure a non plus simplement le droit de lire la valeur du paramètre, mais a aussi le droit de modifier sa valeur : ci-dessus, la valeur du paramètre est incrémentée de 1, et toute variable que vous donnerez en paramètre à "Increm" sera donc incrémentée de 1.

***Pour ceux qui veulent (presque) tout savoir :***
*Les paramètres non variables sont en fait donnés par valeur aux procédures, c'est-à-dire qu'une zone mémoire temporaire est créée et la valeur du paramètre à envoyer est copiée dans cette zone temporaire, et c'est l'adresse de la zone temporaire qui est envoyée à la procédure. Du fait de cette copie, toute modification laisse invariant le paramètre, ce qui correspond bien au fait qu'il soit non variable. En revanche, dans le cas des paramètres variables, l'adresse du paramètre est directement envoyée à la procédure, ce qui lui permet de modifier directement le paramètre puisque son adresse est connue (on dit que le paramètre est donné par adresse).*
*Dans la mesure du possible, et par précaution, les paramètres variables doivent être employés avec parcimonie, car ils sont souvent responsables d'effets de bords. Cependant, ils sont très intéressants lorsqu'une fonction doit renvoyer plusieurs résultats : on transforme la fonction en procédure et on rajoute les résultats en tant que paramètres variables, de sorte que la procédure écrit ses résultats dans les paramètres, ce qui est un peu tiré par les cheveux, mais couramment employé par les développeurs.*

Il faudra simplement faire attention, dans la pratique, lorsque vous utiliserez "Blockread" ou "Blockwrite", à ce que tous les paramètres que vous leur donnerez soient variables, mis à part le troisième qui pourra être une constante.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XIII-D-3-b - Paramètres non typés

Une petite chose devrait vous choquer dans les déclarations de "Blockread" et "Blockwrite" : le paramètre variable "Buf" n'a pas de type ! C'est un cas très particulier : Buf sera en fait interprété par la procédure comme un emplacement mémoire (un peu comme un pointeur, et le type véritable de ce que vous donnerez comme valeur pour "Buf" n'aura absolument aucune importance : vous pourrez envoyez absolument n'importe quoi, du moment que c'est une variable. Dans la pratique, cette caractéristique nous permettra d'envoyer une variable de n'importe quel type, qui contiendra les données à écrire. Ce sera très pratique pour écrire directement des entiers et des chaînes de caractères par exemple. Nous verrons cependant que la lecture sera un peu moins drôle.

## XIII-D-3-c - Description des deux procédures de lecture et d'écriture

Dans les deux procédures, le premier paramètre variable devra être la variable correspondante à un fichier binaire ouvert. Le second devra être une variable de n'importe quel type (nous y reviendrons, rassurez-vous). Nous désignerons désormais ce second paramètre sous le nom de "Buffer" (« Tampon » en français). Le troisième paramètre, constant, indique pour la lecture le nombre d'octets à prendre dans le fichier et à mettre dans le Buffer : la position dans le fichier est avancée d'autant d'octets lus et ce nombre d'octets réellement lu est écrit dans le quatrième paramètre variable "AmtTransferred". Pour l'écriture, ce troisième paramètre indique le nombre d'octets à prendre dans le Buffer et à écrire dans le fichier. La position dans le fichier est ensuite augmentée du nombre d'octets réellement écrits, et ce nombre est écrit dans le paramètre "AmtTransferred".
D'une manière générale, pour reprendre ce qui a été dit ci-dessus, "F" désigne un fichier binaire, "Buf" un Buffer, "Count" le nombre d'octet qu'on désire transferrer, et "AmtTransferred" est un paramètre résultat (dont la valeur initiale n'a pas d'importance) dans lequel le nombre d'octets réellement transferrés est écrit. Vous pourrez ainsi savoir si vos lectures ou vos écritures se sont bien passées.
La lecture et l'écriture de fichiers binaires étant un art dans lequel on excelle pas vite, nous allons décrire en détail la lecture et l'écriture des types fondamentaux, puis nous passerons aux types élaborés tels les tableaux et les enregistrements.

## XIII-D-4 - Lecture et écriture des différents types de données

L'ensemble de ce (trop) gros paragraphe doit être considéré comme une manipulation guidée : en effet, il est indispensable d'effectuer les manipulations proposées, sans quoi vous n'aurez pas compris grand chose à ce qui va être dit. Nous allons commencer par quelques petits compléments sur les types de données, puis voir comment lire et écrire des types de plus en plus complexes dans les fichiers binaires.

## XIII-D-4-a - Compléments sur les types de données

Chaque type de données que vous connaissez occupe une certaine quantité d'espace en mémoire (un nombre d'octets). Cette place, qui paraît parfois variable comme dans le cas des tableaux dynamique, est en fait toujours fixée à un nombre d'octets. Pour faire simple, nous ne parlerons pas de l'influence que peuvent avoir les systèmes d'exploitation et les types d'ordinateur sur la place qu'occupe ces types en mémoire, mais il faut savoir que les chiffres que je vais annoncer n'ont qu'une validité restreinte à Windows 95, 98, Millenium, NT 4.x ou 2000, et sur des architectures basées sur des processeurs Intel, Cyrix ou AMD : nous n'en parlerons donc pas...
Il est possible de connaître l'occupation mémoire (toujours en octets) d'un type simple en utilisant la fonction SizeOf. Cette fonction des plus étranges accepte en paramètre non pas seulement les variables, mais aussi les types. Elle renvoie le nombre d'octets occupés par un élément (constante, variable) de ce type ou la taille de la variable. Par exemple, éxécutez l'instruction suivante qui donne la taille en octet d'un booléen :

```
ShowMessage('Un booléen occupe '+IntToStr(SizeOf(Boolean))+
  ' octet(s) en mémoire.');
```

La réponse devrait être :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Maintenant, déclarez le type suivant (qui doit vous rappeler quelque chose) :

```
TPersonne = record
  Nom,
  Prenom: string[100];
  Age: word;
  Homme: boolean; { vrai: homme, faux: femme }
end;
```

Exécutez maintenant l'instruction :

```
ShowMessage('Un élément de type TPersonne occupe '+IntToStr(SizeOf(TPersonne))+
  ' octet(s) en mémoire.');
```

La réponse devrait avoisiner les 206 octets. Enfin, essayez l'instruction suivante :

```
ShowMessage('Un pointeur occupe '+IntToStr(SizeOf(Pointer))+
  ' octet(s) en mémoire.');
```

Cette fois, pas de doute sur la réponse : 4 octets (Quelques jolies surprises vous attendent avec les types "String" et "TObject" par exemple) : ne vous rabâche-t-on pas régulièrement que les systèmes actuels sont 32 bits ? Eh bien réfléchissez à cela : 32 bits, c'est 4 octets... comme par hasard.
Si j'insiste aussi lourdement sur cette instruction, c'est non seulement parce qu'elle permet d'entrevoir ce qu'il peut y avoir en dessous des types que vous utilisez, là où personne n'aime vraiment mettre les mains, mais surtout évidemment parce que les valeurs renvoyées par SizeOf vont énormément nous servir pour les lectures/écritures dans les fichiers binaires.
L'autre sujet que je souhaite aborder ici, et qui est en rapport étroit avec le précédent, concerne les chaînes de caractères. Pour ces dernières, les lectures et les écritures vont être quelque peu pénibles, car si l'écriture ne posera pas vraiment de soucis, en prenant des précautions, la lecture nous posera une difficulté que j'expliquerai et que nous contournerons facilement grâce à l'emploi de la fonction "Length". Cette fonction accepte une chaîne en paramètre et renvoie sa vraie longueur en caractères (et non pas la place qui lui est réservée en mémoire). Par exemple, dans l'exemple suivant, la réponse sera 6 pour la longueur et 51 pour la taille :

```
procedure TForm1.Button1Click(Sender: TObject);
type
  Str50 = String[50];
var
  S: Str50;
begin
  S := 'coucou';
  ShowMessage('La longueur de S est '+IntToStr(length(S)));
  ShowMessage('La taille de Str50 est '+IntToStr(SizeOf(Str50))+' octet(s).'); end;
```

Vous voyez qu'il serait inutile d'écrire 51 octets dans un fichier alors que 6 suffiraient en théorie (en pratique, il en faudra 7, j'expliquerai pourquoi). Entre 51 et 7, il y a une petite différence, qui devient énorme si on envisage d'écrire 10000 chaînes dans le fichier. "Length" et "SizeOf" nous servirons donc conjointement.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Nous voilà armés pour débuter les lectures/écriture. Pour donner un peu de piment aux sections suivantes, un problème tout à fait réel que vous affronterez tôt ou tard va être étudié : l'écriture et la lecture d'un fichier d'options. L'usage d'un tel procédé est évidemment utilisé pour sauvegarder les options d'un logiciel. Courage et en selle.

## XIII-D-4-b - Préliminaires

Avant de nous lancer tête baissée dans la lecture et l'écriture, il nous faut réfléchir un minimum à l'ouverture et à la fermeture de nos fichiers : ce sera toujours la même suite d'instructions, ce qui fait qu'on peut écrire deux fonctions qui seront utilisées pour réaliser l'ouverture et la fermeture d'un fichier. Ces fonctions éviterons par la suite d'avoir à répéter un grand nombre d'instruction : ce serait pénible pour votre serviteur (en l'occurence moi), et aurait tendance à vous perdre dans un flot de code source.
Ces deux fonctions vont utiliser un paramètre variable F de type File. En effet, les deux fonctions devront manipuler F pour l'ouvrir ou pour le fermer. Ainsi, il suffira de déclarer une variable fichier, et de la transmettre à l'une des deux fonctions pour que l'ouverture et la fermeture se fasse avec toutes les vérifications qui s'imposent. Le résultat de chaque fonction renseignera sur le succés de l'opération.
Sans plus vous faire languir, voici les deux fonctions en question, qui sont largement inspirées de la procédure "OuvrirFichier" présentée ci-dessus ("OuvrirFichier" est d'ailleurs transformée en fonction et ne fait plus que l'ouverture) :

```
function OuvrirFichier(var F: File; NomFich: String): boolean;
begin { retourne vrai si l'ouverture s'est bien passé }
  {$I-}
  AssignFile(F, NomFich);
  { ouverture }
  if FileExists(NomFich) then
    Reset(F, 1)
  else
    Rewrite(F, 1);
  Result := (IOResult = 0);
  if not Result then exit;
  { vidage du fichier }
  Seek(F, 0);
  {$I+}
end;

function FermeFichier(var F: File): boolean;
begin { retourne vrai si la fermeture s'est bien passé }
  {$I-}
  { fermeture }
  CloseFile(F);
  Result := (IOResult = 0);
  {$I+}
  if not Result then exit;
end;
```

Désormais, lorsque nous disposerons d'une variable "F" de type "File", OuvrirFichier(F, *nom de fichier*) tentera d'ouvrir le fichier indiqué et renverra vrai si l'ouverture s'est bien passée. De même, FermerFichier(F) tentera de fermer F et renverra vrai si la fermeture s'est bien passée.

## XIII-D-4-c - Types simples : entiers, réels et booléens

Les types simples comme les entiers, les réels et les booléens sont les plus simples à lire et à écrire. Admettons que nous ayons à écrire un fichier qui fasse une sauvegarde d'une liste d'options d'un logiciel. Pour simplifier, ces options seront dans un premier temps uniquement des booléens et des nombres. Les options consistent pour l'instant en deux cases à cocher : si la première est cochée, une valeur entière est donnée par l'utilisateur, sinon, cette valeur est ignorée. La deuxième case à cocher permet de saisir une valeur non entière cette fois. Cette valeur est également ignorée si la case n'est pas cochée. La signification concrête de ces paramètres nous importe peu, car notre travail consiste ici à permettre la lecture et l'écriture de ces options.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Afin de représenter facilement ces données, nous allons créer un type enregistrement, qui ne sera en aucun utilisé pour générer un fichier séquentiel : ce serait une très mauvaise idée qui, bien qu'encore réalisable ici, deviendra irréalisable très bientôt.

```
TOptions = record
  Opt1, Opt2: Boolean;
  Choix1: Integer;
  Choix2: Real;
end;
```

Commencez par déclarer la variable et la constante suivante à la fin de l'interface de l'unité principale du projet (je vais expliquer leur utilité ci-dessous) :

```
const
  OptsFichier = 'c:\test.dat';
var
  FOpts: File;
```

Nous allons écrire deux fonctions ("LireOptions" et "EcrireOptions") qui lironr et écrirons respectivement les options dans un fichier dont le nom est connu à l'avance. Commençons par "EcrireOptions" : voici le squelette de cette fonction. La fonction renvoie un booléen qui indique si les options ont pu être écrites. Le seul paramètre de la fonction permet de spécifier les valeurs à écrire dans le fichier. Le code ci-dessous réalise l'ouverture et la fermeture du fichier.

```
function LireOptions(var Opts: TOptions): Boolean;
begin
  { on ne mettra Result à vrai qu'à la fin, si tout va bien }
  Result := false;
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  { effacement initial du fichier }
  {$I-}
  Truncate(FOpts);
  if IOResult &lt;> 0 then exit;
  {$I+}
  { lecture ici }
  Result := FermeFichier(FOpts);
end;
```

Comme vous pouvez le constater, la fonction renvoie le résultat de FermeFichier, tout simplement parce que toutes les autres opérations feront quitter la fonction en renvoyant False. Ainsi, si la dernière opération réussit, c'est que tout a réussi, sinon, on sait que quelque chose n'a pas marché, sans savoir quoi.
La structure du fichier d'options sera la suivante : si l'option 1 vaut vrai, la valeur entière est écrite dans le fichier, et pas dans le cas contraire. De même, la valeur réelle n'est écrite dans le fichier que si l'option 2 vaut vrai. Cette structure pourrait se noter ainsi (cette notation n'a rien d'officiel, elle est inventée pour la circonstance :
Opt1: Boolean
{
Vrai: Choix1: Integer
Faux:
}
Opt2: Boolean
{
Vrai: Choix2: Real
Faux:
}
la lecture devra se faire entre deux directives {$I-} et {$I+} pour capturer au fur et à mesure les éventuelles erreurs.
Voici la première instruction, celle qui écrit "Opt1" :

```
BlockWrite(FOpts, Opts.Opt1, SizeOf(Opts.Opt1), nbTrans);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cette instruction mérite quelques explications, puisque c'est la première fois qu'on utilise "Blockwrite". Le premier paramètre est le fichier dans lequel on souhaite écrire. Ce fichier sera ouvert car sinon cette instruction n'est pas atteinte. Le second paramètre donne le "Buffer", c'est-à-dire la donnée à écrire. Le troisième paramètre donne le nombre d'octets à écrire. Etant donné que la donnée est de type booléen, on donne la taille d'un booléen, et donc la taille de Opts1 en octets. Le quatrième paramètre est une variable qui devra être déclarée de type Integer et qui reçoit le nombre d'octets réellement écrits dans le fichier.
Cette instruction devra être immédiatement suivie d'un test pour vérifier ce qui s'est passé. "IOResult" doit être appelée et comparée à 0, et nbTrans doit aussi être vérifié pour savoir si le nombre correct d'octets a été écrit. Ceci est fait par l'instruction suivante à laquelle il faudra vous habituer :

```
if (IOResult <> 0) or (nbTrans <>  SizeOf(Boolean)) then exit;
```

Concrètement, si la procédure continue de s'éxécuter après ces deux instructions, il y a de fortes chances pour que "Opt1" ait été écrit dans le fichier. La suite dépend de cette valeur de "Opt1" : nous allons effectuer l'écriture de "Choix1", mais dans une boucle **if** qui permettra de n'éxécuter cette écriture que si "Opt1" est vrai. Voici le bloc complet, où seule l'instruction qui contient "Blockwrite" est intéressante :

```
if Opts.Opt1 then
  begin
    { écriture de choix1 }
    BlockWrite(FOpts, Opts.Choix1, SizeOf(Opts.Choix1), nbTrans);
    if (IOResult <> 0) or (nbTrans <> SizeOf(Opts.Choix1)) then exit;
  end;
```

Comme vous le voyez, on a simplement substitué "Opts.Choix1" à "Opts.Opt1". Les deux instructions sont presque les mêmes et sont toujours du style : « écriture - vérification ».
La suite de l'écriture ressemble à s'y méprendre à ce qui est donné ci-dessus, voici donc la fonction complète :

```
function EcrireOptions(Opts: TOptions): Boolean;
var
  nbTrans: Integer;
begin
  { on ne mettra Result à vrai qu'à la fin, si tout va bien }
  Result := false;
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  { effacement initial du fichier }
  {$I-}
  Truncate(FOpts);
  if IOResult <> 0 then exit;
  { écriture de Opt1 }
  BlockWrite(FOpts, Opts.Opt1, SizeOf(Opts.Opt1), nbTrans);
  if (IOResult <> 0) or (nbTrans <> SizeOf(Opts.Opt1)) then exit;
  if Opts.Opt1 then
    begin
      { écriture de Choix1 }
      BlockWrite(FOpts, Opts.Choix1, SizeOf(Opts.Choix1), nbTrans);
      if (IOResult < 0) or (nbTrans <> SizeOf(Opts.Choix1)) then exit;
    end;
  { écriture de Opt2 }
  BlockWrite(FOpts, Opts.Opt2, SizeOf(Opts.Opt2), nbTrans);
  if (IOResult <> 0) or (nbTrans <> SizeOf(Opts.Opt2)) then exit;
  if Opts.Opt2 then
    begin
      { écriture de Choix2 }
      BlockWrite(FOpts, Opts.Choix2, SizeOf(Opts.Choix2), nbTrans);
      if (IOResult <> 0) or (nbTrans <> SizeOf(Opts.Choix2)) then exit;
    end;
  {$I+}
  Result := FermeFichier(FOpts);
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Maintenant que vous avez vu la version longue, nous allons voir comment raccourcir cette encombrante fonction. En effet, vous remarquez que nous répétons 4 fois deux instructions (une écriture et un test) avec à chaque fois assez peu de variations. Il est possible de faire une fonction qui fasse cette double tâche d'écriture et de vérification, et qui renverrait un booléen renseignant sur le succés de l'écriture. Voici cette fonction :

```
function EcrireElem(var F: File; var Elem; nbOctets: Integer): Boolean;
var
  nbTrans: Integer;
begin
  {$I-}
  BlockWrite(F, Elem, nbOctets, nbTrans);
  Result := (IOResult = 0) and (nbTrans = nbOctets);
  {$I+}
end;
```

"EcrireElem accepte trois paramètres : le premier, variable, est le fichier dans lequel on doit écrire. Le second, également variable, est la donnée à écrire. Le troisième, comme dans "BlockWrite", est le nombre d'octets à écrire. Cette fonction va considérablement nous simplifier la tâche, car les deux instructions d'écriture-vérification vont se transformer en quelque chose du genre :
**if not** EcrireElem(FOpts, *variable*, SizeOf(*variable*)) **then** exit;
Dans notre procédure, nous gagnerons ainsi 4 lignes et surtout nous gagnerons en lisibilité. Nous pouvons faire encore mieux en utilisant un bloc **with** qui permettra d'éviter les références à "Opts". Enfin, toutes les opérations sur les fichiers étant réalisées à l'extérieur de la fonction "EcrireOptions", on peut retirer les directives {$I-} et {$I+}. On pourra enfin retirer la variable "nbTrans". Voici la fonction nouvelle formule :

```
function EcrireOptions(Opts: TOptions): Boolean;
begin
  { on ne mettra Result à vrai qu'à la fin, si tout va bien }
  Result := false;
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  { effacement initial du fichier }
  {$I-}
  Truncate(FOpts);
  if IOResult <> 0 then exit;
  {$I+}
  with Opts do
    begin
      { écriture de Opt1 }
      if not EcrireElem(FOpts, Opt1, SizeOf(Opt1)) then exit;
      if Opt1 then
        { écriture de Choix1 }
        if not EcrireElem(FOpts, Choix1, SizeOf(Choix1)) then exit;
      { écriture de Opt2 }
      if not EcrireElem(FOpts, Opt2, SizeOf(Opt2)) then exit;
      if Opt1 then
        { écriture de Choix1 }
        if not EcrireElem(FOpts, Choix2, SizeOf(Choix2)) then exit;
    end;
  Result := FermeFichier(FOpts);
end;
```

Passons maintenant aux tests. Nous allons simplement déclarer une variable de type TOptions et la remplir de données factices. Enfin, nous lancerons l'écriture de ces données dans un fichier. Vous pourrez contrôler la bonne écriture du fichier en utilisant un éditeur héxadécimal tel qu'**UltraEdit** ou Hex WorkShop. Voici la procédure a utiliser pour effectuer le test :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OptTest: TOptions;
begin
  with OptTest do
    begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
      Opt1 := True;
      Choix1 := 16;
      Opt2 := False;
      { Choix2 non fixé }
    end;
  EcrireOptions(OptTest);
end;
```

Cette procédure donne une valeur arbitraire à certains champs de "OptTest" et lance l'écriture. Le fichier destination est fixé, je le rappelle, par une constante nommée "OptsFichier". A la fin de l'éxécution de cette procédure, le contenu héxadécimal du fichier devrait être :
01 10 00 00 00 00
Je ne m'engagerai pas sur le terrain glissant de la description détaillée de ces octets, mais sachez que si vous avez ce qui est présenté ci-dessus, votre fichier est correctement écrit. Rien ne vous empèche ensuite de modifier "OptTest" pour voir ce que cela donne dans le fichier généré (le nombre d'octets pourra alors être différent).
Maintenant que l'écriture est faite, il nous reste encore à programmer la lecture du fichier. Cette lecture se fait sur le modèle de l'écriture : lorsqu'on testait la valeur d'une variable avant d'écrire, il faudra ici tester la valeur de la même variable (qui aura été lue auparavant) pour savoir si une lecture du ou des éléments optionnels est à réaliser. La lecture dans un fichier séquentiel se fait avec "BlockRead", qui s'utilise a peu près comme "BlockWrite". Le début de la fonction "LireOptions" ayant été donné plus haut, nous allons directement nous attaquer à la lecture.
Comme pour l'écriture, il est avantageux pour les éléments simples d'écrire une petite fonction annexe "LireElem" qui effectuera les détails de la lecture. Depuis "LireOptions", il nous suffira de faire quelques appels à LireElem et le tour sera joué. Voici donc ce qu'on pourrait utiliser comme fonction "LireElem" :

```
function LireElem(var F: File; var Elem; nbOctets: Integer): Boolean;
var
  nbTrans: Integer;
begin
  {$I-}
  BlockRead(F, Elem, nbOctets, nbTrans);
  Result := (IOResult = 0) and (nbTrans = nbOctets);
  {$I+}
end;
```

Comme pour l'écriture, on désactive la gestion automatique des erreurs d'entrée-sortie, et on lance un "BlockRead". Un test sur "IOResult" et sur "nbTrans" est alors effectué pour savoir si la lecture s'est bien passée. Pour lire un élément dans le ficbier d'options, et à condition que ce dernier soit ouvert, il suffira alors d'utiliser "LireElem" comme on le faisait avec "EcrireElem". Voici la partie qui lit le premier booléen (comme pour la lecture, on utiliser un bloc **with** pour raccourcir les écritures)

```
with Opts do
  begin
    { lecture de Opt1 }
    if not LireElem(FOpts, Opt1, SizeOf(Opt1)) then exit;
    { suite de la lecture ici }
  end;
```

La suite de la lecture consiste éventuellement à lire la valeur de "Choix1", qui dépend de "Opt1". Comme cette valeur vient d'être lue, on peut s'en servir. Voici donc la suite, qui inclue la lecture de "Choix1" dans un bloc if testant la valeur de "Opt1" :

```
if Opt1 then
  { lecture de Choix1 }
  if not LireElem(FOpts, Choix1, SizeOf(Choix1)) then exit;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Vous voyez qu'en écrivant les choses dans le style ci-dessus, les écritures et les lectures se lancent d'une façon très similaire. Dans les sections suivantes, nous verrons que cela n'est pas toujours possible. Mais pour l'instant, voici la fonction "LireOptions" en entier :

```
function LireOptions(var Opts: TOptions): Boolean;
begin
  { on ne mettra Result à vrai qu'à la fin, si tout va bien }
  Result := false;
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  with Opts do
    begin
      { lecture de Opt1 }
      if not LireElem(FOpts, Opt1, SizeOf(Opt1)) then exit;
      if Opt1 then
        { lecture de Choix1 }
        if not LireElem(FOpts, Choix1, SizeOf(Choix1)) then exit;
      { lecture de Opt2 }
      if not LireElem(FOpts, Opt2, SizeOf(Opt2)) then exit;
      if Opt2 then
        { lecture de Choix2 }
        if not LireElem(FOpts, Choix2, SizeOf(Choix2)) then exit;
    end;
  Result := FermeFichier(FOpts);
end;
```

Nous allons tout de suite tester cette fonction. Le principe va être de déclarer une variable de type "TOptions", d'aller lire le fichier écrit auparavant pour obtenir les valeurs des champs de la variable, puis d'afficher les résultats de la lecture. Voici la procédure effectuant le test :

```
procedure TForm1.Button2Click(Sender: TObject);
var
  OptTest: TOptions;
begin
  LireOptions(OptTest);
  if OptTest.Opt1 then
    ShowMessage('Opt1 vrai, Choix1 vaut ' + IntToStr(OptTest.Choix1))
  else
    ShowMessage('Opt1 faux');
  if OptTest.Opt2 then
    ShowMessage('Opt2 vrai, Choix2 vaut ' + FloatToStr(OptTest.Choix2))
  else
    ShowMessage('Opt2 faux');
end;
```

Si vous voulez concocter une petite interface, libre à vous. Pour ceux d'entre vous que cela intéresse, le projet ci-dessous comporte non seulement tout ce qui est écrit ci-dessus sur les types simples, et implémente une petite interface permettant de fixer visuellement la valeur de OptTest lors de l'écriture et de lire visuellement ces mêmes valeurs lors de la lecture. Comme la création d'interfaces n'est pas à l'ordre du jour, je n'y reviendrai pas.
Téléchargement : **BINAIRE1.ZIP** (4 Ko)

## XIII-D-4-d - Types énumérés

Les types énumérés nécessitent un traitement particulier pour être stockés dans un fichier binaire. En effet, ces types sont une facilité offerte par Pascal Objet, mais ils ne pourront en aucun cas être traités comme tels dans les fichiers binaires. Bien que ces types soient en fait des entiers bien déguisés, il ne nous appartient pas de nous appuyer sur de telles hypothèses, même si elles nous simplifieraient la vie. L'astuce consistera donc à écrire la valeur ordinale d'un élément de type énuméré. Lors de la lecture, il faudra retransformer cette valeur ordinale en un élément du type énuméré correct, ce qui se fera assez simplement.
Considérons le type énuméré et la variable suivants :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
type
  TTypeSupport = (tsDisq35, tsDisqueDur, tsCDRom, tsDVDRom, tsZIP);
var
  SupTest: TTypeSupport;
```

Pour ce qui est de l'écriture, il faudra passer par une variable temporaire de type "Integer" qui stockera la valeur ordinale de "SupTest". Cette valeur entière sera ensuite écrite normalement par un "BlockWrite". Voici un extrait de code qui démontre cela :

```
var
  SupTest: TTypeSupport;
  tmp, nbOct: Integer;
begin
  { ouverture et effacement ... }
  tmp := Ord(SupTest);
  {$I-}
  BlockWrite(Fichier, tmp, SizeOf(Integer), nbOct);
  if (IOResult <> 0) or (nbOct <> SizeOf(Integer)) then exit;
  {$I+}
  { fermeture ... }
end;
```

Lorsqu'on a besoin de lire un élément de type énuméré (que l'on a écrit, et donc dont on est sûr du format de stockage), il faut lire une valeur entière, qui est la valeur ordinale de l'élément à lire. Par une simple opération dite de « transtypage » (nous ne nous étendrons pas pour l'instant sur ce vaste sujet), il sera alors possible de transformer cette valeur ordinale en une valeur du type énuméré. Voici un extrait de code qui effectue une lecture correspondante à l'écriture ci-dessus.

```
var
  SupTest: TTypeSupport;
  tmp, nbOct: Integer;
begin
  { ouverture ... }
  {$I-}
  BlockRead(Fichier, tmp, SizeOf(Integer), nbOct);
  if (IOResult <> 0) or (nbOct <> SizeOf(Integer)) then exit;
  {$I+}
  SupTest := TTypeSupport(tmp);
  { fermeture ... }
end;
```

La lecture est très conventionnelle : elle s'effectue en donnant une variable de type "Integer" à la fonction "BlockRead". Vous remarquez au passage que la lecture et l'écriture manipulent exactement le même nombre d'octets. Mais l'instruction qui nous intéresse est plutôt celle-ci :

```
SupTest := TTypeSupport(tmp);
```

Sous l'apparence d'un appel de fonction, c'est une fonctionnalité très intéressante de Pascal Objet qui est utilisée ici : le transtypage des données. Cette instruction ne se lit pas comme un appel de fonction, mais réalise en fait une conversion de la valeur de "tmp" en une valeur de type "TTypeSupport". Cette valeur est alors affectée à "SupTest". Ce transtypage dit « explicite » est obligatoire, sans quoi le compilateur vous retournerait une erreur.
Lorsqu'on parle d' « explicite », c'est qu'il existe une version « implicite ». En effet, lorsque vous additionnez les valeurs de deux variables : une de type "Integer" et une de type "Real" par exemple, vous écrivez simplement l'addition et vous stockez le résultat dans une variable de type "Real" par exemple. Il faut savoir qu'un transtypage implicite s'opèère lors de cette addition : l'entier est converti en nombre à virgule flottante (dont la partie décimale est nulle mais non moins existante), puis l'addition s'effectue, et enfin le résultat est éventuellement converti dans le type de la variable à laquelle vous affectez le résultat. Si j'ai pris le temps de vous rappelez ceci, c'est pour que vous ne vous effrayez pas devant un transtypage du style « TTypeSupport(tmp) » qui n'a rien de bien compliqué, à condition de penser à le faire.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XIII-D-4-e - Types chaîne de caractères

Les chaînes de caractères sont un morceau de choix lorsqu'il s'agit de les écrire ou de les lire depuis un fichier binaire. En effet, il existait auparavant un seul type de chaîne de caractère, nummé "String", limité à 255 caractères. Depuis Delphi 3, ce type "String" désigne une nouvelle implémentation du type chaîne de caractères, limité à (2 puissance 32) caractères, soit pratiquement pas de limite. L'ancien type "String" est dorénavant accessible par le type "ShortString". Pour éviter toute ambiguité, il est également possible (mais je vous le déconseille) d'utiliser "AnsiString" à la place de "String".
Comme si deux types ne suffisaient pas, le type "String" peut, suivant les cas, contenir des chaînes de caractères à 1 octet (les chaînes classiques), ce qui correspond dans la plupart des cas à l'implémentation par défaut, mais ce type peut également stocker des chaînes Unicode, c'est-à-dire des chaînes de caractères sur 2 octets (chaque caractère est stocké sur 2 octets, ce qui donne un alphabet de 65536 caractères au lieu de 256). Comme la deuxième implémentation n'est pas encore très couramment employée, et qu'elle est nettement plus délicate à manipuler, nous nous limiterons ici à la première exclusivement.
Que nous parlions de chaînes courtes ("ShortString") ou de chaînes longues ("String"), les notions suivantes sont valables :

1. Chaque caractère est stocké dans 1 octet en mémoire
2. Le premier caractère est le caractère n°1. Ca a peut-être l'air bête, dit comme ça, mais si S est une chaîne, S[1] est son premier caractère (si S en contient au moins 1) et S[0] ne signifie rien pour les chaînes longues, alors qu'il contient une valeur intéressante pour les chaînes courtes, ce qui nous permettra de gagner du temps.
3. La fonction "Length", qui admet un paramètre de type chaîne courte ou longue, renvoie la longueur d'une chaîne.
4. La procédure "SetLengrh", qui admet en premier paramètre une chaîne courte ou longue, et en second un nombre entier, fixe la longueur d'une chaîne à cette valeur entière. Cette procédure nous servira lors des lectures.

Le format de stockage d'une chaîne de caractères est assez intuitif, bien qu'il n'ait rien d'obligatoire (vous pouvez en inventer un autre, charge à vous de le faire fonctionner) : on écrit d'abord une valeur entière qui est le nombre d'octets dans la chaîne, puis on écrit un octet par caractère (puisqu'on se limite à des caractères de 1 octet).
Ainsi, pour les chaînes courtes, un octet sera suffisant pour indiquer la taille de la chaîne, car un octet peut coder une valeur entre 0 et 255, ce qui est précisément l'intervalle de longueur possible avec une chaîne courte. Après cet octet initial, viendront de 0 à 255 octets (1 par caractère). Les chaînes courtes sont particulièrement adaptées à ce traitement car elles sont stockées de manière continue en mémoire (un bloc d'un seul tenant) en commençant par le caractère 0 qui stocke la longueur actuelle de la chaîne. En fait, ce caractère 0, de type Char comme les autres, ne sera pas exploitable directement. Pour avoir la valeur entière qu'il représente, il faudra au choix passer par la fonction "Ord" pour obtenir sa valeur ordinale ou alors effectuer un transtypage vers le type "Byte" (« Byte(S[0]) » si S est une chaîne de caractères), ou encore passer par la fonction "Length".
En ce qui concerne les chaînes longues, une valeur sur 4 octets (un "Integer") sera nécessaire pour stocker la longueur. Les caractères seront écrits tout comme ci-dessus après cette valeur. le nouveau type "String" n'offre pas, comme l'anciern, de facilité pour obtenir la longueur. Il faudra donc en passer par "Length".
Voici ci-dessous un exemple démontrant l'écriture d'une chaîne courte (la fonction d'écriture et la procédure de test. Les deux fonctions OuvrirFichier et FermeFichier sont celles du paragraphe précédent) :

```
function EcrireChaineC(S: ShortString): boolean;
var
  nbOct: Integer;
begin
  Result := false;
  { ouverture }
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  {$I-}
  { effacement initial }
  Seek(FOpts, 0);
  Truncate(FOpts);
  if IOResult <> 0 then exit;
  { écriture de la chaîne }
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  BlockWrite(FOpts, S[0], Byte(S[0]) + 1, nbOct);
  if (IOResult <> 0) or (Byte(S[0]) + 1 <> nbOct) then exit;
  {$I+}
  { fermeture }
  Result := FermeFichier(FOpts);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  S: ShortString;
begin
  S := 'Bonjour à tous !';
  EcrireChaineC(S);
end;
```

L'appel a "BlockWrite" est la seule instruction qui nous intéresse ici : le premier paramètre est toujours le fichier dans lequel on écrit. Le second est plus original : ici, on ne transmet pas une variable, opération qui consisterait à donner l'adresse du premier octet de cette variable (vous vous souvenez des adresses, n'est-ce pas ? Sinon, retournez faire un tour **ici**), mais le premier de la suite d'octets qu'on souhaite écrire dans le fichier (c'est l'adresse de ce premier octet qui sera prise). Dans notre cas, le premier octet stocke le nombre de caractères. Les caractères suivants étnat les caractères. Ceci explique d'ailleurs la valeur donnée en nombre d'octets à transmettre : c'est le nombre de caractères obtenu par un transtypage de "S[0]" en "Byte" (qui donne le nombre de caractères) auquel on ajoute 1 pour écrire non seulement l'octet qui donne ce nombre au début, plus les Byte(S[0]) caractères de la chaîne.
Une fois la procédure de test éxécutée, une édition héxadécimale du fichier résultant donne :
10 42 6F 6E 6A 6F 75 72 20 E0 20 74 6F 75 73 20 21
Vous voyez que la première valeur héxadécimale est 10, c'est-à-dire 16 en décimal. Après cette valeur, viennent 16 autres qui sont les valeurs héxadécimales des 16 caractères de la chaîne 'Bonjour à tous !'.
Venons-en à l'écriture de chaînes longues. La fonction suivante réalise cela :

```
function EcrireChaineL(S: String): boolean;
var
  Longu, nbOct: Integer;
begin
  Result := false;
  { ouverture }
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  {$I-}
  { effacement initial }
  Seek(FOpts, 0);
  Truncate(FOpts);
  if IOResult < 0 then exit;
  { écriture de la chaîne }
  Longu := Length(S);
  BlockWrite(FOpts, Longu, SizeOf(Longu), nbOct);
  if (IOResult <> 0) or (SizeOf(Longu) <> nbOct) then exit;
  BlockWrite(FOpts, S[1], Longu, nbOct);
  if (IOResult <> 0) or (Longu <> nbOct) then exit;
  {$I+}
  { fermeture }
  Result := FermeFichier(FOpts);
end;
```

Un listing hexadécimal du fichier résultant serait alors :
Un listing hexadécimal du fichier résultant serait alors :
10 00 00 00 42 6F 6E 6A 6F 75 72 20 E0 20 74 6F 75 73 20 21
10 00 00 00 42 6F 6E 6A 6F 75 72 20 E0 20 74 6F 75 73 20 21
Vous remarquez que les 16 dernières valeurs sont identiques à celles présentes dans le fichier précédent, ce qui est normal puisque les mêmes caractères ont été écrits. Ce qui change, c'est la séquence : "10 00 00 00" qui remplace "10". Cette séquence correspond à la valeur 16, mais en 32 bits (4 octets) cette fois au lieu des 8 bits (1 octet) l'autre fois (La valeur de 16 en héxadécimal et sur 32 bits est 00000010, ce qui donne 10 00 00 00 dans un fichier car c'est une convention de stocker les octets dans l'ordre inverse pour les nombres).
Voilà pour l'écriture qui est, comme vous pouvez le voir, quelque chose d'assez simple à réaliser. La lecture est un peu plus délicate pusqu'il faut savoir quel type de chaîne a été écrite dans le fichier, puis lire la taille de la chaîne en


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

spécifiant le bon nombre d'octets (1 ou 4), fixer la longueur de la chaîne, opération souvent oubliée par les néophytes, et enfin lire autant d'octets que de caractères.

*Note : cela n'a rien à voir avec la notion dont nous sommes en train de parler, mais à partir de ce point, les codes source Pascal que vous pourrez voir sont générés automatiquement par une petite application Delphi, étant donné qu'ils commencent à s'allonger et que la conversion manuelle est assez pénible. Si vous constatez des problèmes de mise en forme ou de mise en page, n'hésitez pas à* ***me contacter***

```
function LireChaineC(var S: Shortstring): boolean;
var
  Longu: Byte;
  nbOct: Integer;
begin
  Result := false;
  { ouverture }
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  {$I-}
  Seek(FOpts, 0);
  if IOResult <> 0 then exit;
  { lecture de l'octet indiquant la longueur }
  BlockRead(FOpts, Longu, 1, nbOct);
  if (IOResult <> 0) or (nbOct <> 1) then exit;
  SetLength(S, Longu);
  { lecture des caractères }
  BlockRead(FOpts, S[1], Longu, nbOct);
  if (IOResult <> 0) or (nbOct <> Longu) then exit;
  {$I+}
  { fermeture }
  Result := FermeFichier(FOpts);
end;
```

La lecture se fait donc en deux temps : un élément de type "Byte" sert à lire l'octet qui donne la longueur de la chaîne. Une fois cet octet lu avec succès, on commence par fixer la longueur de la chaîne. Que ce soit pour une chaîne courte et à plus forte raison pour une chaîne longue, cette étape est indispensable pour préparer la chaîne à recevoir le bon nombre de caractères. Il ne faut pas ici fixer un nombre trop petit car alors les caractères supplémentaires seront au mieux perdus et provoqueront au pire un plantage de votre application et/ou de Windows (vous avez remarqué comme Windows est un environnement stable, n'est-ce pas ?). La lecture des caractères se base sur le nombre lu précédemment. La valeur transmise en deuxième paramètre donne, comme précédemment pour les écritures, le premier octet mémoire dans lequel écrire les données lues dans le fichier.
La lecture d'une chaîne longue se fait exactement de la même manière, mis à part que la variable temporaire "Longu" doit être de type Integer. Voici la fonction qui démontre la lecture d'une chaîne longue :

```
function LireChaineL(var S: string): boolean;
var
  Longu,
  nbOct: Integer;
begin
  Result := false;
  { ouverture }
  if not OuvrirFichier(FOpts, OptsFichier) then exit;
  {$I-}
  Seek(FOpts, 0);
  if IOResult <> 0 then exit;
  { lecture de l'octet indiquant la longueur }
  BlockRead(FOpts, Longu, 4, nbOct);
  if (IOResult <> 0) or (nbOct <> 4) then exit;
  SetLength(S, Longu);
  { lecture des caractères }
  BlockRead(FOpts, S[1], Longu, nbOct);
  if (IOResult <> 0) or (nbOct <> Longu) then exit;
  {$I+}
  { fermeture }
  Result := FermeFichier(FOpts);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;
```

Et voilà le travail !
Essayez de manipuler tout cela un peu vous-même, en combinant par exemple les chaînes courtes, les chaînes longues et d'autres éléments.

## XIII-D-4-f - Autres types

Les types de base étant maîtrisés, les types plus élaborés ne vous poseront pas de problème. Pour écrire un enregistrement, il suffira d'écrire ses éléments un par un dans le fichier dans un format qui permet de relire les données sans ambiguité (comme avec les chaînes de caractères). Pour écrire les tableaux (dynamiques ou non), il faudra faire comme avec les chaînes : écrire dans une variable sur 1, 2 ou 4 octets le nombre d'éléments, puis écrire les éléments un par un.
En ce qui concerne les pointeurs, les écrire sur disque relève du non-sens, puisqu'ils contiennent des adresses qui sont spécifique à l'ordinateur, à l'utilisation de la mémoire, et également à l'humeur du système d'exploitation. Les objets ou composants ne s'écrivent pas non plus sur disque. Si vous voulez mémoriser certaines de leurs propriétés, il faudra connaître leur type et écrire des valeurs de ce type dans le fichier.

## XIII-D-4-g - Structures avancées dans des fichiers binaires

Ce que j'ai décrit dans les paragraphes précédents suffit la plupart du temps lorsqu'on a une structure de fichier figée ou linéaire, c'est-à-dire dont les éléments ont un ordre déterminé et un nombre d'occurences prévisibles. Par contre, dés qu'on s'aventure dans le stockage de données complexes, les structures linéaires deviennent insuffisantes et on doit avoir recours à d'autres structures. Ce que j'ai expliqué dans un grand nombre de pages écran n'est qu'un petit bout de ce qu'il est possible de faire avec les fichiers binaires.
Le but de ce paragraphe n'est pas de passer en revue les différentes structures plus ou moins évoluées de fichiers, qui existent virtuellement en nombre infini, mais de donner un exemple partant d'une structure bien réelle : celle du GIF.
Notre exemple sera la structure des ficbiers GIF, que vous avez certainement déjà eu l'occasion de rencontrer : ces fichiers sont en effet omniprésents sur l'Internet. Ces fichiers GIF ont une structure bien définie, mais aucunement linéaire. La base du fichier comporte un petit nombre d'octets qui donnent la "version" du fichier, c'est-à-dire une indication sur le format des données du fichier. Ensuite, viennent un certain nombre de blocs, dont certains sont obligatoires, dont certains rendent obligatoires certains autres, dont l'ordre d'apparition est parfois libre, parfois arbitraire, ... bref, il n'y a pas de structure figée.
Pour lire et écrire de tels fichiers, il faut élaborer des algorithmes parfois complexes, qui font appel à différentes procédures ou fonctions qui s'appelent mutuellement. Ce que j'essaie de vous dire, c'est qu'à ce niveau, il m'est impossible de vous guider dans la lecture ou l'écriture de tels fichiers : vous devrez essayer par vous-mêmes.
La même remarque est valable si vous avez besoin d'une structure de stockage personnalisée : dés que vous sortez des sentiers bien balisés des fichiers texte et des fichiers séquentiels, vous aterrissez dans le domaine complètement libre des fichiers binaires, ou vous êtes certes maître de la structure de vos fichiers, mais où cette liberté se paye en complexité de conception des méthodes de lecture et d'écriture.
Si je prends le temps de vous assomer de généralités à la fin d'un aussi long chapitre, c'est avant tout pour vous inciter à vous méfier de vous-même : il est en effet très tentant, dés lors qu'on se sent libre de choisir la structure d'un fichier, d'élaborer avec soin la structure d'un fichier de sauvegarde. Seulement, il faudra alors penser au temps que vous consacrerez à l'écriture des procédures/fonctions de lecture et d'écriture dans vos fichiers : le jeu en vaudra-t-il la chandelle ?


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XIV - Découverte des composants les plus utilisés - 2ème partie

Après un long chapitre consacré aux fichiers, nous revenons à la charge sur les composants. Ce chapitre est la suite du **chapitre 10** : il commence là ou s'arrètait ce dernier. Dans le chapitre 8, nous n'avons étudiée que les composants les plus simples, car il vous manquait des connaissances indispensables sur les pointeurs et l'utilisation des objets. Le **chapitre 11** et le **chapitre 12**, outre leur intérêt indéniable dans leur domaine (pointeurs et objets), vous ont aussi permis d'acquérir les connaissances de base nécessaires à l'utilisation de composants plus évolués. Ces composants, nous allons en étudier quelques-uns ici.

*Attention : ce chapitre ne sera pas écrit tout de suite car des sujets plus importants vont être traités en priorité. Je m'excuse à l'avance à ceux qui comptaient sur ce chapitre.*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XV - Manipulation de types abstraits de données

## XV-A - Introduction

Si vous avez pratiqué un tant soit peu la programmation par vous-même en dehors de ce guide, vous vous êtes certainement rendu compte que dés qu'un programme doit manipuler des données un peu volumineuses ou structurées, il se pose le problème souvent épineux de la représentation de ces données à l'intérieur du programme. Pourquoi un tel problème ? Parce que la plupart des langages, Pascal Objet entre autres, ne proposent que des types "simples" de données (mêmes les tableaux et les enregistrements seuls suffisent rarement à tout faire).
La plupart des programmes que vous serez amenés à écrire nécessiteront l'emploi de structures de données plus complexes que celles proposées par Pascal Objet, seules capables de manipuler par exemple des données relatives à une file d'attente ou des données hiérarchisées. La création de types de données évolués et efficaces pose souvent de gros problèmes aux débutants, d'où ce chapitre présentant ces types, appelés ici "Types abstraits de données", ou TAD. Ils sont introduits (pour ceux d'entre vous qui ne les connaissent pas), puis implémentés à partir des possibilités offertes par le langage Pascal Objet (ceci est possible dans beaucoup d'autres langages). Ils permettent de modèliser à peu près n'importe quel ensemble de données soi-même. En contrepartie de leur puissance, l'emploi de ces structures complexes est souvent malaisé et nécessite une attention et une rigueur exemplaire.
Delphi propose des classes toutes faites permettant de manipuler des TAD simples comme les "piles", les "files" et les "listes" (classes TStack, TFile et TList), mais le but de ce chapitre n'est en aucun cas de vous apprendre à vous servir de ces classes en vous laissant tout ignorer de leur fonctionnement : au contraire, ce chapitre parlera des notions théoriques (sans toutefois vous noyer dans la théorie) et nous écrirons ensemble des unités complètes servant à la manipulation de ces types élaborés. Libre à vous ensuite d'utiliser ces classes qui ne sont pas si mal faites que ça. Le langage Pascal Objet préconise d'ailleurs l'utilisation d'objets plutôt que de procédures/fonctions classiques. Nous aurons l'occasion de créer une classe à partir d'une de ces structures (l'arbre général) dans les prochains chapitres, ce qui procurera une bien meilleure méthode d'implémentation (si ce terme vous laisse froid(e), sachez qu'il signifie à peu près "traduction d'un ensemble d'idées ou d'un schémas de fonctionnement ou de raisonnement dans un langage de programmation, ce qui ne devrait pas être beaucoup mieux, désolé...).
Ce chapitre se propose de vous familiariser avec les plus simples des types abstraits de données que vous aurez à manipuler : "piles", "files", "listes" et "arbres" (juste un aperçu pour ces derniers). En parallèle de l'étude générale puis spécifique au language Pascal Objet de ces TAD, des notions plus globales seront étudiées, comme le tri dans une liste de données. Plusieurs méthodes de tri seront ainsi abordées parmi lesquelles les célèbres (ou pas...) Tri à Bulles, Tri Shell et Tri Rapide ("QuickSort" pour les intimes). Enfin, avant de vous lancer dans la lecture de ce qui suit, assurez-vous de bien avoir lu et compris le **chapitre 11** sur les pointeurs car les pointeurs vont être largement employés ici et dans tous les sens car ils permettent une implémentation de presque tous les TAD vus ici.
N'ayons pas peur des mots : les notions abordées dans ce chapitre sont nettement moins faciles à assimiler que celles des chapitres précédents mais sont néanmoins des plus importantes. J'insiste assez peu sur la théorie et me focalisant sur des exemples concrets d'implémentation en Pascal Objet. Un mini-projet très connu attend les plus téméraires à la fin de ce chapitre. Temporairement, je ne donnerai pas la solution, et je vous laisserai patauger un peu. Si vous avez des questions, utilisez exclusivement la **mailing-list** (je répondrais aux questions si d'autres ne le font pas avant moi). Cette expérience vous sera d'ailleurs utile pour apprendre à formuler vos questions et à travailler, dans une certaine mesure, en commun. Si vous avez réalisé le mini-projet et souhaitez être corrigé, **envoyez-le moi** si possible zippé (SANS le fichier .EXE, mon modem 56K vous en remercie d'avance !). N'envoyez pas de fichier source ni exécutable attaché sur la liste de diffusion, ce serait inutile car elle est configurée pour les refuser.
Une dernière remarque avant de commencer (j'ai encore raconté ma vie dans cette intro !) : Certains d'entre vous vont sans aucun doute se demander avant la fin de ce chapitre, voire dés maintenant, ce qui est plus génant encore, pourquoi je m'ennuie à vous parler de tout cela alors qu'on peut tout faire avec des bases de données. Ma réponse est double :

- l'exécution d'un programme faisant appel à une base de données est beaucoup plus lente que celle d'un programme classique, car l'accès aux bases de données nécessite souvent des temps d'attente qui se ressentent pendant l'exécution, à encore plus forte raison si la ou les bases accédée(s) n'est (ne sont) pas locale(s). De plus, la distribution de tels programme nécessite la distribution d'applications externes dont on peut se passer dans les cas les plus simples.
- La notion des structures avancées faisant intensivement appel aux pointeurs et aux techniques de programmation introduites ici est indispensable à tout programmeur digne de ce nom. En effet, ce sont


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

des procédés auquels un programmeur aura rapidemnt affaire dans d'autres langages comme le C, très couramment employé.

# XV-B - Piles

## XV-B-1 - Présentation

Les "piles", que nous désignerons plutôt par « TAD Pile », permettent de gèrer un ensemble d'éléments de même type de base, avec comme principe de fonctionnement « premier dedans, dernier dehors ». Ce type de fonctionnement est très pratique pour modèliser au niveau du stockage des données des phénomènes existants.
Un exemple parlant est une de ces boites cylindriques dans lesquelles on met les balles de tennis : lorsque vous voulez une balle, vous n'avez directement accès qu'à la première balle. Si vous voulez la deuxième, il vous faudra d'abord enlever la première, puis la deuxième, et ensuite éventuellement remettre la première. Un autre exemple plus classique mais moins parlant est la pile d'assiettes. Lorsque vous voulez une assiette, vous prennez celle du dessus, sans allez prendre la troisième en partant du haut. Pour constituer une telle pile, vous mettez une assiette n°1, puis une assiette n°2, et ainsi de suite. Lorsque vous récupèrez les deux assiettes, vous devrez d'abord prendre l'assiette n°2 et ensuite l'assiette n°1. Un troisième exemple serait un tube de comprimés (de la vitamine C UPSA par exemple, pour ne pas faire de publicité) par exemple : lorsque vous ouvrez le tube, vous ne voyez que le premier comprimé, que vous êtes obligé de prendre pour avoir accès au second. Lorsque ce tube a été rempli, un comprimé n°1 a été disposé au fond du tube, puis un comprimé n°2, puis... jusqu'au comprimé n°10. Quoi que vous fassiez, à moins de tricher en découpant le tube, vous DEVREZ prendre le comprimé n°10, puis le n°9 ... puis le n°1 pour constater que le tube est vide, comme avant que le fabriquant ne le remplisse.
La suite du paragraphe va certainement vous paraître ennuyeuse, voire saoulante, mais c'est un passage incontournable avant de passer à la pratique. Essayez de lire toute la théorie sans vous endormir, puis essayez les exemples, puis enfin revenez sur la théorie : vous la découvrirez alors sous un jour différent, nettement moins antipathique.

## XV-B-2 - Une définition plus formelle

Sans pousser le vice jusqu'à vous servir les spécifications du TAD Pile comme je les ai apprises en Licence Informatique (je ne suis pas méchant à ce point), ce paragraphe se propose de donner une définition plus formelle des piles. En effet, on ne peut pas faire tout et n'importe quoi à propos des piles, pour au moins une (simple) raison : ce sont des piles d'éléments, mais de quel type sont ces éléments ? Comme nous n'avons aucun droit ni moyen de répondre à cette question, il va falloir en faire abstraction, ce qui est, je vous l'assure, tout-à-fait possible mais exige une certaine rigueur.
Une pile peut être manipulée au moyen d'opérations de base. Ces opérations sont les suivantes (pour chaque opération, une liste des paramètres avec leur type et un éventuel résultat avec son type est indiqué. Ceci devra être respecté dans la mesure du possible) :

- Création d'une nouvelle pile vide
  paramètres : (aucun)
  résultat : pile vide
- Destruction d'une pile
  paramètres : une pile
  résultat : (aucun)
- Empilement d'un élément (sur le sommet de la pile)
  paramètres : une pile et un élément du type stocké par la pile
  paramètres : une pile et un élément du type stocké par la pile
  résultat : une pile remplaçant complètement celle donnée en paramètre (l'élément a été empilé si possible)
- Dépilement d'un élément (retrait du sommet, sous réserve qu'il existe)
  paramètres : une pile
  résultat : une pile remplaçant complètement celle donnée en paramètre (l'élément a été dépilé si possible)
- Test de pile vide qui indiquera si la pile est vide ou non
  paramètres : une pile
  résultat : un booléen (vrai si la pile est vide)


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Test de pile pleine, qui indiquera si la capacité de la pile est atteinte et qu'aucun nouvel élément ne peut être ajouté (cette opération n'est pas toujours définie, nous verrons pourquoi dans les exemples).
paramètres : une pile
résultat : un booléen (vrai si la pile est pleine)
- Accès au sommet de la pile (sous réserve qu'il existe)
paramètres : une pile
résultat : un élément du type stocké par la pile

Bien que cela paraisse rudimentaire, ce sont les seules opérations que nous pourrons effectuer sur des piles. Pour utiliser une pile, il faudra d'abord la créer, puis autant de fois qu'on le désire : empiler, dépiler, accéder au sommet, tester si la pile est vide ou pleine. Enfin, il faudra penser à la détruire. Cela devrait vous rappeler, bien que ce soit un peu hors propos, le fonctionnement des objets : on commence par créer un objet, puis on l'utilise et enfin on le détruit. Tout comme les objets, considérés comme des entités dont on ne connait pas le fonctionnement interne depuis l'extérieur, les piles devront être considérées depuis l'extérieur comme des boites noires manipulables uniquement avec les opérations citées ci-dessus.
Au niveau abstrait, c'est à peu près tut ce qu'il vous faut savoir. Cette théorie est valable quel que soit le language que vous serez amené(e) à manipuler. Nous allons maintenant voir l'application concrète de cette théorie au seul language qui nous intéresse ici : le Pascal. Deux manières d'implémenter (de créer une structure et des fonctions/ procédures pratique à manipuler) des piles vont être étudiées : en utilisant un tableau, et en utilisant des pointeurs. Place à la pratique.

## XV-B-3 - Implémentation d'une pile avec un tableau

Un moyen simple d'implémenter une pile et d'utiliser un tableau. chaque élément de la pile sera stocké dans une case du tableau. Une telle structure ne sera pas suffisante : rien ne permettrait de retrouver facilement le sommet ni de déterminer si la pile est vide ou pleine. Pour ces raisons, nous allons utiliser une valeur entière qui indiquera le numéro de la case qui contient le sommet de la pile, ou -1 par exemple si la pile est vide. Pour rassembler les deux éléments, nous créerons un enregistrement. Enfin, une constante va nous permettre de définir la taille du tableau et ainsi de savoir lorsque la pile sera pleine. Dans nos exemples, nous allons manipuler des chaînes de caractères. Voici les déclarations dont nous auront besoin :

```
const
  TAILLE_MAX_PILE = 300;

type
  TPileTab = record
    Elem: array[1..TAILLE_MAX_PILE] of string;
    Sommet: Integer;
  end;
```

Comme vous pouvez le constater dans le code ci-dessus, un enregistrement de type TFileTab permettra de manipuler une pile. Les éléments (des chaînes, dans le cas présent) seront stockés dans le tableau Elem qui peut contenir TAILLE_MAX_PILE éléments. La position du sommet sera mémorisée dans Sommet. Une valeur de -1 signifiera une pile vide et MAX_TAILLE_PILE une pile pleine (dans laquelle on ne pourra plus rajouter d'éléments puisque toutes les cases seront remplies). Mais attention, il va falloir travailler à deux niveaux : le niveau intérieur, où ces détails ont une signification, et le niveau extérieur où n'existera que le type TPileTab et les opérations de création, empilement, dépilement, accès au sommet, test de pile vide ou pleine et destruction. A ce second niveau, TPileTab devra être considéré comme une boite noire dont on ne connait pas le contenu : ce n'est plus un enregistrement, mais une pile.
Il va maintenant falloir nous atteler à créer chacune des opérations servant à manipuler une pile. Chacune de ces opérations va être créée au moyen d'une procédure ou d'une fonction. Depuis l'extérieur, il nous suffira ensuite d'appeler ces procédures et ces fonctions pour faire fonctionner notre pile. Ensuite, si nous devions modifier la structure TPileTab, il faudrait juste réécrire les opérations de manipulation et si cela a été bien fait, aucune des lignes de code utilisant la pile n'aura besoin d'être modifiée.

*Note 1 : sous Delphi, il est possible pour une fonction de retourner un type enregistrement. Nous nous servons de cette possibilité assez rare pour simplifier les choses. Cependant, la plupart des languages ne permettent pas de retourner un enregistrement, c'est pourquoi*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

*il faudra alors impérativement passer par des pointeurs comme nous le ferons dans la deuxième implémentation.*

*Note 1 : certaines opérations comme Empiler, Dépiler et Sommet devront être utilisées avec précaution. Ainsi, il faudra systématiquement tester que la pile n'est pas pleine avant d'empiler, même si ce test sera à nouveau réalisé lors de l'empilement. Il faudra également toujours tester que la pile n'est pas vide avant de dépiler ou d'accèder au sommet de la pile. Voici tout de suite le code source de PTNouvelle qui crée une nouvelle pile vide.*

```
function PTNouvelle: TPileTab;
begin
  result.Sommet := -1;
end;
```

Le code ci-dessus ne devrait pas poser de problème : on retourne un enregistrement de type TPileTab dans lequel on fixe le sommet à -1, ce qui dénote bien une pile vide. Passons tout de suite aux indicateurs de pile vide ou pleine :

```
function PTVide(Pile: TPileTab): Boolean;
begin
  result := Pile.Sommet = -1;
end;

function PTPleine(Pile: TPileTab): Boolean;
begin
  result := Pile.Sommet = TAILLE_MAX_PILE;
end;
```

La fonction PTVide compare la valeur du champ Sommet de la pile transmise à -1, et si le résultat est vrai, alors la pile est vide et on retourne True, sinon, on retourne False. Nous aurions pû écrire un bloc **if** faisant la comparaison et retournant True ou False selon les cas, mais il est plus rapide de constater que la valeur de Result est la valeur booléenne renvoyée par la comparaison Pile.Sommet = -1, d'où l'affectation directe qui peut dérouter au début mais à laquelle il faudra vous habituer. PTPleine marche suivant le même principe mais compare la valeur de Sommet à TAILLE_MAX_PILE : si les valeurs correspondent, la pile est pleine et le résultat renvoyé est True, sinon False, d'où encore une fois l'affectation directe qui évite d'écrire inutilement un bloc **if**. Passons maintenant à un morceau de choix : l'empilement.

```
function PTEmpiler(Pile: TPileTab; S: string): TPileTab;
begin
  result := Pile;
  if not PTPleine(result) then
    begin
      if result.Sommet = -1 then
        result.Sommet := 1
      else
        inc(result.Sommet);
      result.Elem[result.Sommet] := S;
    end;
end;
```

L'empilement se doit de fournir une nouvelle pile contenant si cela est possible l'élément à empiler. Dans notre cas, cela paraît un peu farfelu, mais lorsque nous parlerons de pointeurs, vous verrez mieux l'intérêt de la chose. Le résultat est d'abord fixé à la valeur de la pile transmise, puis si la pile n'est pas pleine (s'il reste de la place pour le nouvel élément), ce résultat est modifié. En premier lieu, le sommet est incrémenté, ce qui permet de le fixer à la position du nouveau sommet. Ce nouveau sommet est fixé dans l'instruction suivante. La pile ainsi modifiée est retournée en tant que résultat. Voyons maintenant le dépilement :

```
function PTDepiler(Pile: TPileTab): TPileTab;
begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  result := Pile;
  if not PTVide(result) then
    begin
      result.Elem[result.Sommet] := '';
      if result.Sommet > 1 then
        dec(result.Sommet)
      else
        result.Sommet := -1;
    end;
end;
```

PTDepiler fonctionne sur le même principe que la fonction précédente : le résultat est d'abord fixé à la pile transmise, puis modifié cette fois si la pile n'est pas vide (s'il y a un élément à dépiler). A l'intérieur du bloc if testant si la pile n'est pas vide, on doit mettre à jour la pile. On commence par fixer la chaîne indexée par Sommet à la chaîne vide (car cela permet de libérer de la mémoire, faites comme si vous en étiez convaincu(e) même si ça ne paraît pas évident, j'expliquerai à l'occasion pourquoi mais ce n'est vraiment pas le moment), puis la valeur de Sommet est soit décrémentée, soit fixée à -1 si elle valait 1 avant (car alors le dépilement rend la pile vide puisqu'il n'y avait qu'un élément avant de dépiler).
Après ces deux fonctions un peu difficiles, voici la procédure qui détruit une pile : dans le cas particulier de notre structure utilisant un enregistrement, nous n'avons rien à faire, mais la procédure ci-dessous a un but purement éducatif : elle vous-montre comment il faut normalement détruire une pile.

```
procedure PTDetruire(Pile: TPileTab);
var
  P: TPileTab;
begin
  P := Pile;
  while not PTVide(P) do
    P := PTDepiler(P);
end;
```

Le fonctionnement est le suivant : on dépile jusqu'à ce que la pile soit vide. Ensuite, il suffit de supprimer la pile vide, ce qui dans notre cas se résume à ne rien faire, mais ce ne sera pas toujours le cas. Voici enfin la fonction qui permet d'accèder à l'élément au sommet de la pile. Notez que cette fonction ne devra être appelée qu'après un test de pile vide, car le résultat n'a aucune signification si la pile est vide !

```
function PTSommet(Pile: TPileTab): string;
begin
  if PTVide(Pile) then
    result := ''
  else
    result := Pile.Elem[Pile.Sommet];
end;
```

La fonction est relativement simple : si la pile est vide, on est bien obligé de retourner un résultat, et donc on renvoie une chaîne vide. Dans le cas favorable où la pile n'est pas vide, la chaîne dans la case indexée par Pile.Sommet est renvoyée, car c'est cette chaîne qui est au sommet de notre pile.
Enfin, il va nous falloir dans la suite de l'exemple une opération permettant de connaître le contenu de la pile. Cette opération, qui ne fait pas partie des opérations classiques, est cependant indispensable pour nous afin de montrer ce qui se passe lorsque les autres opérations sont appliquées. Voici le code source de la procédure en question :

```
procedure PTAffiche(Pile: TPileTab; Sortie: TStrings);
var
  indx: integer;
begin
  Sortie.Clear;
  if Pile.Sommet = -1 then
    Sortie.Add('(pile vide)')
  else
    begin
      if Pile.Sommet = TAILLE_MAX_PILE then
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        Sortie.Add('(pile pleine)');
      for indx := Pile.Sommet downto 1 do
        Sortie.Add(Pile.Elem[indx]);
    end;
end;
```

La procédure ci-dessus accepte deux paramètres : la pile à afficher et la sortie dans laquelle afficher, de type TStrings. Je vous rappelle que TStrings est le type de la propriété Lines des composants "Mémo" ainsi que de la propriété Items des composants "ListBox". Ainsi, nous pourrons utiliser une zone de liste ou un mémo et transmettre une de leur propriété et l'affichage se fera dans le composant en utilisant le paramètre objet "Sortie" qui permet de dialoguer avec le composant sans savoir ce qu'il est exactement. L'intérêt d'utiliser un type TStrings est justement cela : la procédure qui "affiche" la pile est indépendante d'un type de composant particulier puisque tout se fait en dialoguant avec un objet et pas un composant particulier. La première instruction vide la liste des chaînes. Ensuite, soit la pile est vide et seule une ligne indiquant cet état de fait est insérée, soit l'affichage des chaînes a lieu. Un test supplémentaire permet de signaler une pile pleine en insérant un message explicite. Ensuite, les chaînes sont insérées en commençant par celle qui a le numéro Pile.Sommet, qui est donc le sommet de pile, jusqu'à celle qui a le n°1.
C'en est assez d'explications dans le vide. Nous allons créer un projet utilisant tout cela. Vous avez ici trois possibilités :

1. Soit vous êtes courageux(se) et vous avez reconstitué une unité avec l'ensemble des sources proposés, dans ce cas ne vous privez pas de l'utiliser.
2. Soit vous êtes mi-courageux(se), mi-paresseux(se), et vous pouvez télécharger seulement l'unité toute faite ici : **piles_tab.pas** et réaliser le projet en suivant les indications ci-dessous.
3. Soit vous êtes paresseux(se) et vous pouvez télécharger le projet complet et suivre les indications (ne faites ça que si vous n'avez pas réussi à le faire vous-même) : **piles1.zip**

Passons maintenant à la réalisation pratique. Créez un projet et créez une interface ressemblant à celle-ci :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Gestion d'une pile de chaînes

fmPrinc

Contenu actuel de la pile (sommet en premier) :

btEmpile
Empiler une chaîne

btDepile
Dépiler une chaîne

btVidePile
Vider la pile

mePile

btQuitter
Quitter

Le but de cette interface va être de démontrer visuellement comment fonctionne une pile et les effets des diverses opérations sur une pile. Déclarez dans l'unité principale une variable "Pile" de type TPileTab (à vous de voir où c'est le plus avantageux). Générez la procédure associée à l'événement OnCreate de la fiche et entrez le code suivant :

```
procedure TfmPrinc.FormCreate(Sender: TObject);
begin
  Pile := PTNouvelle;
end;
```

Cette simple ligne de code nous permet de créer la pile au démarrage de l'application (lors de la création de la fiche principale). L'opération de création renvoie une nouvelle pile vide qui initialise "Pile". Générez de même la procédure associée à OnDestroy et entrez ce qui suit, qui détruit la pile lors de la destruction de la fiche, lorsque l'application se termine :

```
procedure TfmPrinc.FormDestroy(Sender: TObject);
begin
  PTDetruire(Pile);
end;
```

Nous allons maintenant écrire une procédure qui va nous permettre de mettre l'interface de notre petite application à jour en fonction des modifications apportées à la pile. Cette procédure réaffichera la pile et permettra d'activer ou non certains boutons. Voici ce que cela donne :

```
procedure MajInterface;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
var
  vide: boolean;
begin
  PTAffiche(Pile, fmPrinc.mePile.Lines);
  fmPrinc.btEmpile.Enabled := not PTPleine(Pile);
  vide := PTVide(Pile);
  fmPrinc.btDepile.Enabled := not vide;
  fmPrinc.btVidePile.Enabled := not vide;
end;
```

Dans la code ci-dessus, la pile est affichée au moyen de PTAffiche auquel on transmet la pile et la propriété Lines du méoo : ainsi, le méme va se voir ajouter les lignes correspondantes aux divers cas possibles pour la pile par la procédure PTAffiche. Ensuite, les 3 boutons de manipulation de la pile sont (dés)activés. Le premier n'est activé que lorsque la pile n'est pas pleine, alors que les deux derniers ne le sont que lorsqu'elle n'est pas vide. Notez que le test de pile vide n'est effectué qu'une fois et qu'ensuite la valeur booléenne résultante est utilisée, ce qui permet d'économiser du temps en n'appelant qu'une seule fois la fonction PTVide. Reste maintenant à écrire les procédures associées aux 4 boutons. Voici celle associée au bouton "Empiler" :

```
procedure TfmPrinc.btEmpileClick(Sender: TObject);
var
  S: String;
begin
  if InputQuery('Empilement d''une chaîne', 'Saisissez une chaîne à empiler', S) then
    begin
      Pile := PTEmpiler(Pile, S);
      MajInterface;
    end;
end;
```

Cette procédure demande une chaîne à l'utilisateur à l'aide de la fonction InputQuery. Si l'utilisateur rentre effectivement une chaîne, elle est empilée sur Pile et l'interface est mise à jour : rien de très difficile à comprendre ici. La procédure associée au bouton Dépiler est encore plus simple :

```
procedure TfmPrinc.btDepileClick(Sender: TObject);
begin
  Pile := PTDepiler(Pile);
  MajInterface;
end;
```

Lors d'un clic sur le bouton Dépiler, une chaîne est dépilée de Pile, puis l'interface est mise à jour pour reflèter le changement. Passons enfin à la procédure associée au bouton "Vider la pile" qui présente plus d'intérêt :

```
procedure TfmPrinc.btVidePileClick(Sender: TObject);
begin
  while not PTVide(Pile) do
    Pile := PTDepiler(Pile);
  MajInterface;
end;
```

Cette procédure utilise les opérations standard pour vider la pile : tant qu'on peut dépiler, on le fait, jusqu'à épuisement du stock de chaînes. L'interface est ensuite actualisée. Le projet est a peu près terminé : il ne vous reste plus qu'à programmer le bouton "Quitter" et à lancer l'application : ajoutez des chaînes, retirez-en, et videz. Si vous avez un problème insoluble, téléchargez le **projet complet**.

## XV-B-4 - Compléments sur les pointeurs : chaînage et gestion de la mémoire

Un sujet importantissime qu'il me faut aborder ici est celui du chaînage. Si vous êtes un(e) initié(e), vous vous dites sûrement : "enfin !", sinon : "mais de quoi il parle ???" (et je vais vous faire languir un petit peu). Le chaînage est une notion née d'un besoin assez simple à comprendre : celui de gèrer dynamiquement la mémoire disponible pendant


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

l'exécution d'une application. Les deux notions évoquées dans le titre de ce paragraphe sont donc relativement proches l'une de l'autre puisque la seconde est une nécessité qu'aucun programmeur sérieux ne saurait ignorer, et la première est un moyen très conventionnel la facilitant.
Mais assez parlé par énigmes : la gestion de la mémoire disponible est souvent nécessaire pour de moyennes ou de grosses applications. En effet, il est souvent impossible de prévoir quelle quantité de mémoire sera nécessaire pour éxécuter une application, car cela dépend essentiellement de la taille des données manipulées par cette application. Puisque la quantité de mémoire nécessaire est inconnue, une (très) bonne solution est d'en allouer (de s'en réserver) au fur et à mesure des besoins, et de la libérer ensuite. Ainsi, l'application utilisera toujours le minimum (parfois un très gros minimum) possible de mémoire, évitant ainsi de s'incruster en monopolisant la moitié de la mémoire comme certaines "usines à gaz" que je ne citerai pas car mon cours est déjà assez long comme ça.
L'allocation dynamique passe la plupart du temps par l'utilisation de pointeurs. Ces pointeurs permettent, via l'utilisation de new et de dispose, d'allouer et de libèrer de la mémoire. Mais voilà : une application a très souvent besoin de stocker un nombre quelconque d'éléments du même type. Les types abstraits de données que nous étudions dans ce chapitre constituent alors un très bon moyen de regrouper ces données suivant le type de relation qui existent entre elles. Mais alors un problème se pose : si on utilise des tableaux, la taille occupée en mémoire est fixe (si l'on excepte les artifices des tableaux dynamiques non présents dans la plupart des languages). Une solution est alors le chaînage, notion très simple qui est déclinée de plusieurs manières, dont je vais présenter ici la plus simple, et qui a l'avantage d'être adaptable à tous les languages permettant l'utilisation de pointeurs.
L'idée de départ est très simple : chaque élément va être chaîné à un autre élément. Chaque élément va donc en pratique être doté d'un pointeur venant s'ajouter aux données de l'élément. Ce pointeur pointera vers un autre élément ou vaudra nil pour indiquer une absence d'élément chaîné. Concrètement, voici un petit morceau de code Pascal Objet qui illustre cela :

```
type
  PPileElem = ^TPileElem;
  TPileElem = record
    Elem: string;
    Suiv: PPileElem;
  end;
```

Observez bien cet extrait de code : un premier type PPileElem est déclaré comme étant un pointeur vers un type TPileElem. Ce type, bien qu'encore non défini, n'est pas nécessaire dans le cas d'un type pointeur (Lisez la note ci-dessous si vous voulez en savoir plus). Il faudra tout de même que l'on déclare le type en question, TPileElem. Ceci est fait juste après : TPileElem est un type enregistrement qui contient deux champs, ce qui fait de PPileElem est un pointeur vers un enregistrement. Le premier champ de TPileElem, très conventionnel, est une chaîne. Le second mérite toute notre attention : c'est un champ de type PPileElem, c'est-à-dire un pointeur. Cette définition de type vous interloque peut-être du fait qu'elle a l'air de se "mordre la queue". Cette construction est cependant autorisée.

***Définitions récursives de types***
*La déclaration présente dans le code source ci-dessus est possible et autorisée. La raison se trouve dans le fait que c'est le compilateur qui autorise cela. En effet, il a besoin, lorsque vous définissez un type, de deux informations, qui peuvent être données en une ou deux fois. Si mes propos vous paraissent obscurs, repensez à l'extrait de code :*
*PPileElem = ^TPileElem;*
*Cette définition donne une seule des deux informations nécessaires à la création d'un type : la quantité de mémoire occupée par un élément de ce type. La seconde information, à savoir la nature exacte du type, n'est pas nécessaire ici, étant donné qu'on déclare un type pointeur, dont la taille est toujours de 4 octets. Par conséquent, le compilateur accepte la déclaration, en exigeant cependant que TPileElem soit défini par la suite.*
*Dès lors que cette déclaration est acceptée, le type en question est utilisable pour construire d'autres types. Ceci est fait tout de suite après dans la déclaration du type TPileElem, ce qui permet d' « entremèler » les deux déclarations.*

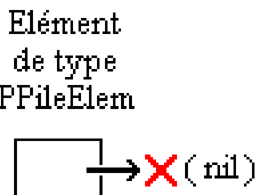
Cette construction éteroclite est dite "simplement chaînée" : chaque élément peut (mais n'est pas obligé de) pointer vers un autre élément, ce qui fait que les éléments sont chaînés entre eux dans un seul sens (d'où le "simplement" ; il existe également des constructions plus complexes dites doublement chaînées qui permettent aux éléments d'être liés dans les deux sens, ou même des chaînages dits circulaires où une boucle est réalisée par le biais de pointeurs).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

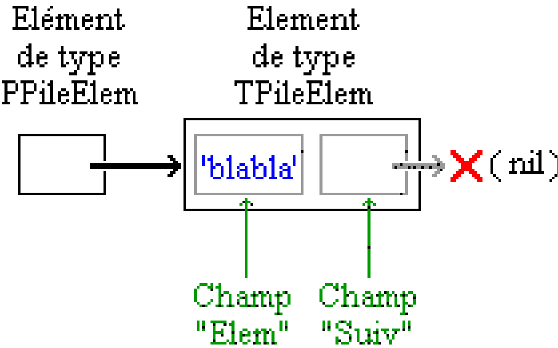
En utilisant ce type de chaînage, il est possible de constituer des listes d'éléments. La méthode est relativement simple : on utilise un seul pointeur de type "PPileElem" par exemple, qui pointe sur un élément de type "TPileElem", qui lui-même pointe sur un autre... jusqu'à ce qu'un d'entre eux ne pointe vers rien (son pointeur vers l'élément suivant vaut **nil**). Il est ainsi possible, si on conserve soigneusement le pointeur vers le premier élément, de retrouver tous les autres par un parcours de la « chaîne ».

## XV-B-5 - Implémentation par une liste chaînée

Nous allons tout de suite mettre ces connaissances en pratique en implémentant un type Pile en utilisant une liste à simple chaînage. Le principe est ici assez simple, mais je vais bien détailler cette fois-ci puisque c'est peut-être la première fois que vous manipulez une liste chaînée. Le principe va être le suivant : on va utiliser un pointeur de type PPileElem égal à nil pour représenter la pile vide.




Lors de l'ajout d'un premier élément dans la pile, voici ce que cela donnera :



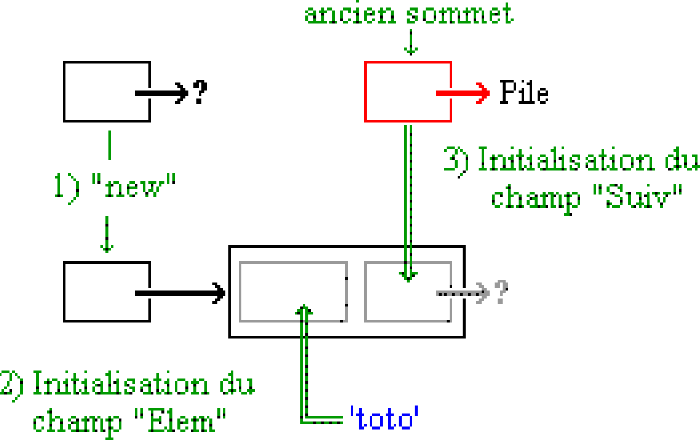

Cela peut paraître contraignant et compliqué, mais ce genre de structure a deux avantages :

1. Lorsque la pile est vide, elle ne prend que 4 octets de mémoire, contrairement à un encombrant tableau.
2. La taille limite de la pile est la taille mémoire disponible, et non pas une constante fixée arbitrairement au moment du développement de l'application.

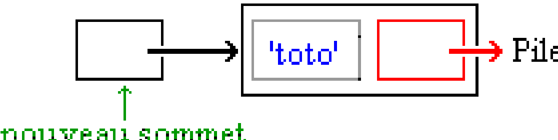
Lorsqu'on devra « empiler » un élément, il suffira de créer un nouvel élément par un appel à "new", puis de fixer le champ "Suiv" de cet élément à l'ancien sommet de pile. Le pointeur vers l'élément nouvellement créé devient alors le nouveau sommet de pile. Voici un petit schémas qui illustre cela :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

**Empilement d'un élément dans une pile**




**Résultat :**




Notes :
- Les flèches simples servent de commentaires
- Les doubles flèches sont des affectations
- Les flèches épaisses symbolisent des pointeurs

Le dépilement est effectué en inversant les étapes précédentes, à condition toutefois que la pile ne soit pas vide. La suppression complète se fait en dépilant jusqu'à vider la pile. Enfin, l'opération de test de pile pleine renverra toujours faux, car on ne tiendra pas compte ici d'un éventuel remplissage de la mémoire. Assez de théorie et de beaux schémas, passons au code source.
La fonction PLNouvelle crée une pile vide. Par convention, la pile vide sera un pointeur nil (rappelez-vous que depuis l'extérieur, la structure véritable cachée derrière la Pile ne doit pas être considéré comme connu ; le pointeur **nil** a donc une signification en "interne", mais pas en "externe"). Voici cette fonction très simple :

```
function PLNouvelle: PPileElem;
begin
  result := nil;
end;
```

La fonction qui indique si une pile est vide teste simplement si le pointeur transmis est égal à **nil**.

```
function PLVide(Pile: PPileElem): Boolean;
begin
  result := Pile = nil;
end;
```

La fonction qui permet d'empiler un élément est nettement plus intéressante. Examinez le code source ci-dessous à la lumière du schémas présenté plus haut. Une variable temporaire de type pointeur est déclarée. La première instruction initialise le pointeur en réservant une zone mémoire pointé par celui-ci. Les deux instructions suivantes initialisent l'élément pointé : le champ "Elem" reçoit le nouvel élément, tandis que le champ "Suiv" reçoit l'ancien


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

pointeur vers le sommet de pile (que ce pointeur soit **nil** ou non importe peu). La quatrième instruction fixe le résultat au pointeur qui vient d'être créé et qui n'est pas détruit ici. C'est là un élément nouveau dans l'utilisation que nous faisons des pointeurs : l'initialisation et la destruction sont séparées ; il faudra donc penser à détruire tous ces pointeurs à un moment donné. Lors de l'appel à "PLEmpiler", la pile actuelle (qui est en fait un pointeur vers le premier élément ou **nil**) est transmise en paramètre. Le résultat de l'appel à la fonction devra être affecté à ce pointeur pour le fixer à sa nouvelle valeur (l'emplacement du sommet de pile a changé puisque ce sommet a changé).

```
function PLEmpiler(Pile: PPileElem; S: string): PPileElem;
var
  temp: PPileElem;
begin
  new(temp);
  temp^.Elem := S;
  temp^.Suiv := Pile;
  result := temp;
end;
```

L'opération de dépilement est sujette à une condition : la pile ne doit pas être vide. Le code de la fonction "PLDepiler" va donc se diviser en deux pour les deux cas : pile vide ou pile non vide. Le second cas est très simple : on ne peut rien dépiler et donc on retourne la pile vide (**nil**). Le premier cas commence par sauvegarder ce qui sera la nouvelle pile : le champ "Suiv" du sommet de pile, c'est-à-dire **nil** si la pile ne contenait qu'un seul élément, ou un pointeur vers le second élément de la pile. La seconde étape consiste à libèrer la mémoire associée à l'ancien sommet de pile, en appelant dispose. Notez que l'ordre de ces deux instructions est important car si on détruisait d'abord le pointeur "Pile", on ne pourrait plus accèder à "Pile^.Suiv".

```
function PLDepiler(Pile: PPileElem): PPileElem;
begin
  if Pile <> nil then
    begin
      Result := Pile^.Suiv;
      Dispose(Pile);
    end
  else
    Result := nil;
end;
```

La procédure de destruction d'une pile est assez simple : tant que la pile n'est pas vide, on dépile. Voici le code source qui est presque une traduction litérale de ce qui précède :

```
procedure PLDetruire(Pile: PPileElem);
begin
  while Pile <> nil do
    Pile := PLDepiler(Pile);
end;
```

La fonction permettant l'accès au sommet de la pile est également sujette à condition : si la pile est vide, on se doit de retourner un résultat, mais celui-ci sera sans signification. Sinon, il suffit de retourner le champ "Elem" du premier élément de la liste chaînée, à savoir Pile^ .

```
function PLSommet(Pile: PPileElem): string;
begin
  if Pile <> nil then
    Result := Pile^.Elem
  else
    Result := ''; // erreur !!!
end;
```

La dernière procédure qui nous sera utile ici est l'affichage du contenu de la pile. On utilise la même sortie que précédemment : un objet de type TStrings. Le principe est ici de parcourir la pile élément par élément. Pour cela,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

nous avons besoin d'un pointeur qui pointera sur chacun des éléments de la pile. Ce pointeur est initialisé à la valeur de Pile, ce qui le fait pointer vers le premier élément de la pile. Ensuite, une boucle teste à chaque itération si le pointeur en cours est **nil**. Si c'est le cas, le parcours est terminé, sinon, on ajoute le champ "Elem" de l'élément pointé et on passe à l'élément suivant en affectant au pointeur la valeur du champ "Suiv", c'est-à-dire un pointeur sur l'élément suivant.

```
procedure PLAffiche(Pile: PPileElem; Sortie: TStrings);
var
  temp: PPileElem;
begin
  temp := Pile;
  Sortie.Clear;
  while temp <> nil do
    begin
      Sortie.Add(temp^.Elem);
      temp := temp^.Suiv;
    end;
end;
```

Si vous avez tout suivi, vous avez de quoi reconstituer une unité complète comprenant la déclaration des types et les fonctions et procédures expliquées plus haut. Vous avez ici trois possibilités :

1. Soit vous êtes courageux(se) et vous avez reconstitué l'unité par vous-même et vous pouvez continuer.
2. Soit vous êtes mi-courageux(se), mi-paresseux(se), et vous pouvez télécharger seulement l'unité toute faite ici : **piles_ptr.pas** et réaliser le projet ci-dessous en suivant les indications données.
3. Soit vous êtes paresseux(se) et vous pouvez télécharger le projet complet et suivre les indications (ne faites ça que si vous n'avez pas réussi à le faire vous-même) : **piles2.zip**

Afin de gagner du temps et de vous démontrer qu'on peut faire abstraction de la structure réelle des données qu'on manipule, nous allons réécrire le projet précédent en utilisant cette fois une pile gérée par une liste chaînée. Le code source va beaucoup ressembler à celui du projet précédent, je serais donc plus bref dans les explications. Recréez la même interface que précédemment, et générez les 3 procédures de réponse aux clics sur les 3 boutons de manipulation de liste. Voici le code qu'il vous faut obtenir :

```
procedure TfmPrinc.btEmpileClick(Sender: TObject);
var
  S: String;
begin
  if InputQuery('Empilement d''une chaîne', 'Saisissez une chaîne à empiler', S) then
    begin
      Pile := PLEmpiler(Pile, S);
      MajInterface;
    end;
end;

procedure TfmPrinc.btDepileClick(Sender: TObject);
begin
  Pile := PLDepiler(Pile);
  MajInterface;
end;

procedure TfmPrinc.btVidePileClick(Sender: TObject);
begin
  while not PLVide(Pile) do
    Pile := PLDepiler(Pile);
  MajInterface;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Déclarez également une variable Pile de type PPileElem à l'endroit où nous l'avions déclaré dans le premier projet. Insèrez la procédure "MajInterface" suivante (vous noterez que le bouton "Empiler" n'est plus désactivable puisque l'opération de test de pile pleine n'existe pas dans cette implémentation) :

```
procedure MajInterface;
var
  vide: boolean;
begin
  PLAffiche(Pile, fmPrinc.mePile.Lines);
  vide := PLVide(Pile);
  fmPrinc.btDepile.Enabled := not vide;
  fmPrinc.btVidePile.Enabled := not vide;
end;
```

Programmez l'action du bouton "Fermer" (un simple "Close;" suffit), générez les procédures de réponse aux événements OnCreate et OnDestroy de la fiche et complètez-les à l'aide du listing ci-dessous :

```
procedure TfmPrinc.FormCreate(Sender: TObject);
begin
  Pile := PLNouvelle;
end;

procedure TfmPrinc.FormDestroy(Sender: TObject);
begin
  PLDetruire(Pile);
end;
```

Voilà, vous avez votre nouveau projet complet. Si vous le lancez, vous constaterez que son fonctionnement est rigoureusement identique au précédent, puisque les changements se situent à un niveau dont l'utilisateur ignore tout. Vous voyez bien par cet exemple qu'il est possible d'isoler une structure de données avec les opérations la manipulant. Si pour une raison X ou Y vous deviez changer la structure de données sous-jacente, les opérations de manipulation devront être réécrites, mais si vous avez bien fait votre travail, comme nous l'avous fait ci-dessus, le code utilisant ces opérations changera de façon anecdotique (la majorité des changements ont consister à renommer les appels de procédures et à ne plus faire appel à PTPleine, que l'on aurait d'ailleurs tout aussi bien pû implémenter par une fonction qui renverrait toujours un booléen faux).
Nous nous sommes limités ici à manipuler des piles de chaînes. Mais rien ne nous interdit de manipuler des structures plus complexes, telles des enregistrements ou même des pointeurs. Il faudra bien faire attention à ne pas confondre dans ce dernier cas les pointeurs servant à structurer la pile et ceux pointant vers les éléments "stockés" dans la pile. Je donnerais un exemple de ce genre de chose dans le paragraphe consacré aux Listes.

## XV-C - Files

### XV-C-1 - Présentation et définition

Le deuxième type de données que nous allons étudier ici est la File. Si vous connaissez ce mot, c'est qu'il est utilisé par exemple dans l'expression "File d'attente". Une file d'attente fonctionne sur le principe suivant (s'il n'y a pas de gens de mauvaise fois dans cette file) : le premier arrivé est le premier servi, puis vient le deuxième, jusqu'au dernier. Lorsqu'une personne arrive, elle se met à la queue de la file et attend son tour (sauf si elle est pressée ou impolie, mais aucune donnée informatique bien élevée ne vous fera ce coup-là). Le type File général fonctionne sur le même principe : on commencer par créer une file vide, puis on "enfile" des éléments, lorsqu'on défile un élément, c'est toujours le plus anciennement enfilé qui est retiré de la file. Enfin, comme pour tout type abstrait, on détruit la file lorsqu'on en a plus besoin. Les opérations permises sur un type File sont :

- Création d'une nouvelle file vide
  paramètres : (aucun)
  résultat : file vide
- Enfilement d'un élément (ajout)


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

paramètres : une file et un élément du type stocké dans la file
paramètres : une file et un élément du type stocké dans la file
résultat : une file contenant l'élément si la file n'était pas pleine

- Défilement d'un élément (retrait)
paramètres : une file
résultat : une file dont la tête a été supprimée si la file n'était pas vide
- Tête (accès à l'élément de tête)
paramètres : une file
résultat : un élément du type stocké dans la file, si elle n'est pas vide
- Test de file vide qui indiquera si la file est vide ou non
paramètres : une file
résultat : un booléen (vrai si la file est vide)
- Test de file pleine, qui indiquera si la capacité de la file est atteinte et qu'aucun nouvel élément ne peut être ajouté (cette opération n'est pas toujours définie, selon les implémentations).
paramètres : une file
résultat : un booléen (vrai si la file est pleine)

Il existe diverses manières d'implémenter un type file. Par exemple, il est possible d'utiliser un tableau conjugué avec deux indices de tête et de queue. Vous pourrez essayer de réaliser cela par vous-même car je n'en parlerai pas ici. Sachez seulement qu'une telle implémentation exige une case "neutre" dans le tableau. **Contactez-moi** si vous voulez plus de détails. Un moyen qui reste dans la lignée de ce que nous venons de faire avec les piles est une implémentation utilisant une liste chaînée.

## XV-C-2 - Implémentation d'une file par une liste chaînée

Une file peut être représentée par une liste à simple chaînage : Chaque élément est alors un "maillon" de cette chaîne", comme pour les piles, sauf que les opérations de manipulation seront différentes. On doit décider dés le début si le premier élément de la liste chaînée sera la tête ou la queue de la liste. Les deux choix sont possibles et l'un comme l'autre défavorisent une ou plusieurs opérations : si on choisit que le premier élément désigne la tête, c'est l' "enfilement" qui est défavorisé car il faudra "parcourir" la chaîne (comme dans la procédure "PLAfficher" décrite plus haut) pour atteindre le dernier élément (la queue de file). Si on choisit le contraire, ce sont les opérations "Défiler" et "Sommet" qui sont défavorisées, pour la même raison. De tels inconvénients pourraient être évités en utilisant un double chaînage, mais cela consomme plus de mémoire et entraîne des complications inutiles... Nous allons donc choisir ici la première solution, à savoir que le premier élément sera la tête.
Nous allons écrire les types, fonctions et procédures nécessaires à la manipulation d'une file d'attente. Voici les déclarations de types nécesaires. Il ne devrait y avoir aucune surprise :

```
type
  TPersonne = record
    Nom: string;
    Prenom: string;
  end;

  PFileElem = ^TFileElem;
  TFileElem = record
    Elem: TPersonne;
    Suiv: PFileElem;
  end;
```

On définit d'abord un type TPersonne très standard, puis un type pointeur PFileElem vers un élément de type TFileElem. TFileElem est ensuite déclaré comme étant un enregistrement, donc un "maillon" de la liste chaînée, pouvant contenir les infos d'une personne et un pointeur vers l'élément suivant de la chaîne. Comme pour une pile, une file sera représentée à l' « extérieur » par un pointeur de type PFileElem. La file vide sera représentée par un pointeur égal à **nil**. On introduira des éléments dans la file comme pour une pile. la tête s'obtiendra comme s'obtenait le sommet de pile. Par contre, le défilement nécessitera un parcours de la file et quelques petits tests. Voici deux


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

fonctions très simples : FLNouvelle (création d'une file vide) et FLVide (test de file vide). Vous noterez qu'on ne peut pas utiliser l'identificateur "File" puisqu'il désigne un type fichier.

```
function FLNouvelle: PFileElem;
begin
  result := nil;
end;

function FLVide(F: PFileElem): Boolean;
begin
  result := F = nil;
end;
```

Rien de très compliqué dans ce qui précède. Venons-en à l'enfilement d'un élément, qui est probablement la fonction la plus délicate à programmer. Le principe est de créer un nouvel élément et de l'accrocher en fin de liste chaînée. Le principe est d'utiliser une boucle **while** qui parcours la liste. Attention cependant, comme il nous faut pouvoir raccrocher le nouvel élément créé à la fin de la liste, il nous faut le champ "Suiv" du dernier élément. La boucle **while** parcours donc la liste élément par élément jusqu'à ce que le "successeur" de l'élément soit *nil*. Ce dernier maillon obtenu, il est ensuite facile d'y accrocher le nouveau maillon. Cette boucle **while** pose un problème si la liste est vide, car alors aucun champ "Suiv" n'est disponible. Dans ce cas, on effectue un traitement spécifique en retournant l'élément nouvellement créé qui devient notre file à un élément.

```
function FLEnfiler(F: PFileElem; Nom, Prenom: string): PFileElem;
var
  f_par,
  temp: PFileElem;
begin
  new(temp);
  temp^.Elem.Nom := Nom;
  temp^.Elem.Prenom := Prenom;
  // attention, fixer Suiv à nil sinon impossible de
  // détecter la fin de file la prochaine fois !
  temp^.Suiv := nil;
  if F = nil then
    result := temp
  else
    begin
      // initialisation du parcours
      f_par := F;
      // parcours
      while f_par^.Suiv <> nil do
        // passage au maillon suivant
        f_par := f_par^.Suiv;
      // accrochage de temp en fin de liste.
      f_par^.Suiv := temp;
      result := F;
    end;
end;
```

La première partie de la fonction crée et initialise le nouveau "maillon". La seconde a pour but d'accrocher ce maillon. Deux cas se présentent : soit la file est vide (F = **nil**) et dans ce cas le nouvel élément créé est retourné. Sinon, après avoir initialisé un pointeur f_par, on parcours les éléments un à un jusqu'à ce que le champ "Suiv" de f_par^ soit **nil**, ce qui signifie alors que f_par est un pointeur vers le dernier élément. Le nouveau maillon créé précédemment est alors "accroché". Passons à l'opération de défilement :

```
function FLDefiler(F: PFileElem): PFileElem;
begin
  if F <> nil then
    begin
      result := F^.Suiv;
      Dispose(F);
    end
  else
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    result := F;
end;
```

Ici, c'est un peu plus simple : si la file est vide, rien n'est fait : la file est retournée vide. Sinon, le premier élément est supprimé (la tête de file). Pour cela, le champ Suiv de F^ est affecté à Result, puis l'appel à Dispose permet de libèrer la mémoire associée à l'élément pointé par F. Le résultat est donc une file privée de son ancienne "tête". Venons-en à la fonction d'accès au sommet.

```
function FLSommet(F: PFileElem): TPersonne;
begin
  if F <> nil then
    result := F^.Elem;
end;
```

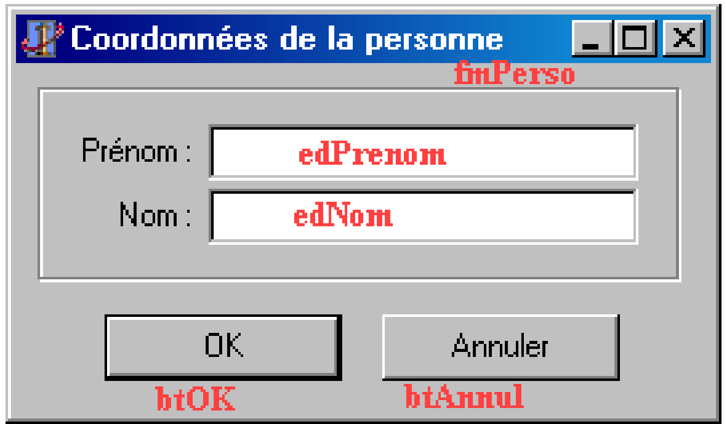
Cette fois, aucun résultat n'est renvoyé lorsque la file est vide, ce qui donnera des résultats imprévisibles : au programmeur utilisant la fonction de faire le test avant. Sinon, on retourne directement un élément de type TPersonne, car il est impossible de retourner plusieurs résultats. Retourner un type enregistrement n'est pas très élégant (dans d'autres languages, c'est tout bonnement impossible), mais c'est aussi ce qu'il y a de plus simple. Venons-en à la procédure de suppression d'une file. C'est une procédure très standard de suppression de tous les éléments d'une liste simplement chaînée. Je n'ai pas, comme pour les piles, délégué les détails à l'opération de retrait d'un élément. Voici ce que cela donne :

```
procedure FLDetruire(F: PFileElem);
var
  temp, sauv: PFileElem;
begin
  temp := F;
  while temp <> nil do
    begin
      sauv := temp^.Suiv;
      dispose(temp);
      temp := sauv;
    end;
end;
```

le principe est d'initialiser un pointeur temporaire au pointeur de départ, et de le faire "avancer" dans la liste chaînée en détruisant les éléments parcourus. Pour cela, on est obligé de sauvegarder le champ "Suiv" à chaque fois car le fait de libèrer la mémoire associée au pointeur le rend innaccessible au moment où on en a besoin.
Nous allons maintenant réaliser une petite application à but purement didactique (son intérêt, comme les deux précédentes, étant assez limité...). Créez un nouveau projet, et ajoutez une fiche. Utilisez les captures d'écran ci-dessous pour dessiner votre interface :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Gestion d'une file d'attente
fmPrinc
Contenu actuel de la file (tête en premier) :
meFile
btAjout
Ajouter une personne
btRetrait
Retirer une personne
btVide
Vider
btQuitter
Quitter




Ajoutez maintenant une unité où vous rentrerez manuellement les types/fonctions/procédures décrites plus haut, ou alors téléchargez l'unité toute faite ici : **files_prt.pas**, ou bien encore téléchargez le projet complet : **files.zip**. N'oubliez pas de rajouter l'unité de la fiche de saisie des coordonnées et celle contenant l'implémentation du TAD File dans la clause uses de l'unité principale. Générez la procédure associée à l'événement OnShow de la fiche de saisie des coordonnées. Complètez le code avec ce qui suit :

```
procedure TfmPerso.FormShow(Sender: TObject);
begin
  edNom.Text := '';
  edPrenom.Text := '';
  ActiveControl := edNom;
```



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;
```

Ceci a pour effet de vider les zones d'édition et d'activer la zone d'édition du nom lorsque la fiche est affichée. Il nous faut maintenant initialiser et détruire la file d'attente aux bons moments. Les événements OnCreate et OnDestroy de la fiche principale sont des endroits idéaux pour cela. Générez donc les procédures associées à ces deux événements et saisissez le code source ci-dessous :

```
procedure TfmPrinc.FormCreate(Sender: TObject);
begin
  F := FLNouvelle;
end;

procedure TfmPrinc.FormDestroy(Sender: TObject);
begin
  FLDetruire(F);
end;
```

Afin de voir l'état de la file, nous devrons régulièrement afficher son contenu dans le Mémo prévu à cet effet. Nous aurons également besoin de faire certaines mises à jour de l'interface en fonction des actions effectuées par l'utilisateur. Le plus simple pour une si petite application est une procédure "MajInterface" qui se chargera des mises à jour. Cette procédure sera appelée chaque fois que nécessaire et réalisera entre autre l'affichage de la file et l'activation de certains boutons.

```
procedure MajInterface;
var
  vide: boolean;
begin
  FLAffiche(F, fmPrinc.meFile.Lines);
  vide := FLVide(F);
  fmPrinc.btRetrait.Enabled := not vide;
  fmPrinc.btVide.Enabled := not vide;
end;
```

Dans la procédure ci-dessous, pour que l'affichage puisse se faire dans le mémo, on doit transmettre sa propriété "Lines" qui est bien de type "TStrings". Pour accèder à cette propriété, il nous faut bien la « qualifier » avec "fmPrinc.meFile" car la procédure est indépendante de la fiche et il faut donc commencer par accèder à celle-ci puis au mémo et enfin à la propriété "Lines". Le reste ne présente aucune difficulté. Nous pouvons maintenant nous occuper des opérations effectuées par chacun des trois boutons (je vous laisse programmer l'action du bouton "Quitter" vous-même). Le premier bouton permet de saisir une nouvelle personne et de l'insèrer dans la file d'attente. Pour saisir les coordonnées de la personne, nous aurions pû utiliser deux "InputQuery", mais cela manquerait quelque peu de convivialité. C'est pour cette raison que nous avons créé une fiche annexe servant de fenêtre de saisie. Cette fiche sera appelée par sa méthode "ShowModal" (c.f. **chapitre 10** pour plus de détails) ; il faut donc fixer les propriétés "ModalResult" des deux boutons "Ok" et "Annuler" de cette fiche aux valeurs respectives de "mrOk" et "mrCancel". Une petite précaution réalisée ci-dessus nous permet de vider automatiquement les zones de saisie et de réactiver la zone de saisie de nom (car sinon ce serait le bouton cliqué lors de la dernière utilisation qui serait activé) à chaque nouvelle utilisation de cette fiche.

```
procedure TfmPrinc.btAjoutClick(Sender: TObject);
begin
  // si l'utiliasteur a cliqué sur OK dans la fenêtre de saisie
  if fmPerso.ShowModal = mrOK then
    begin
      // Enfilement d'un nouvel élément
      F := FLEnfiler(F, fmPerso.edNom.Text, fmPerso.edPrenom.Text);
      // Mise à jour de l'interface pour reflèter ce changement
      MajInterface;
    end;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

On commencer par afficher la fiche de saisie. Comme on doit tester le résultat de l'appel à "ShowModal", on met cet appel dans un bloc if qui compare la valeur de retour à "mrOK". Si c'est la valeur de retour, l'utilitaire a confirmé les coordonnées et l'ajout dans la file peut être fait. Pour cela, on utilise la fonction "FLEnfiler" et on transmet la file et les champs "Text" des composants "edNom" et "edPrenom" de la fiche de saisie "fmPerso". Le résultat est réaffecté à F. Enfin, l'interface est mise à jour, ce qui permet d'afficher le nouveau contenu de la file. Vous verrez que le nouvel élément s'ajoute à la suite des autres contrairement au cas d'une pile où il s'affichait avant tous les autres.
Le bouton "Retirer une personne" permet de défiler un élément. Ce bouton n'est activé que lorsque la file n'est pas vide. En fait, ce n'est pas tout à fait le cas, car lors du lancement de l'application, ce bouton est activé alors qu'il ne devrait pas l'être. Changez cet état de fait en fixant les propriétés "Enabled" des boutons "btRetrait" et "btVide" en "false". Les instructions éxécutées par un clic sur ce bouton sont assez simples.

```
procedure TfmPrinc.btRetraitClick(Sender: TObject);
begin
  F := FLDefiler(F);
  MajInterface;
end;
```

La première instruction défile (l'élément de tête). La seconde met l'interface à jour. Venons-en enfin à l'action du bouton "Vider" :

```
procedure TfmPrinc.btVideClick(Sender: TObject);
begin
  while not FLVide(F) do
    F := FLDefiler(F);
  MajInterface;
end;
```

Le code ci-dessus vide la file dans le sens où il défile élément par élément tant que la file est non vide. L'interface est ensuite mise à jour. Si vous avez suivi toutes les explications ci-dessus, vous êtes maintenant en possession d'un projet en état de fonctionner. Lancez-le, ajoutez des personnes, retirez-en, en notant la différence par rapport aux piles. Si vous n'avez pas réussi une ou plusieurs des étapes ci-dessus, vous pouvez télécharger le code source complet du projet ici : **files.zip**. Nous allons maintenant passer aux listes.

## XV-D - Listes

### XV-D-1 - Présentation et définition

La notion de liste d'éléments est une notion très naturelle dans le sens où nous manipulons des listes assez régulièrement, ne serait-ce que lorsque nous faisons notre liste de courses (liste d'articles) ou lorsque nous consultons un extrait de compte (liste d'opérations bancaires). Ces listes famillières cachent cependant plusieurs concepts : comment sont-elles construites, leur applique-t-on un tri ? Quel ordre dans ce cas ? Pour un extrait de compte, c'est simplement un ordre chronologique, dans de nombreux cas ce sera un tri numérique (à partir de nombres), alphabétique (à partir de lettres) ou plus souvent alphanumérique (lettres et chiffres). Les listes étant souvent plus intéressantes lorsqu'elles sont triées, ce paragraphe présente un TAD Liste trié. Il est assez simple de réaliser un type non trié en éliminant tout ce qui se rapporte au tri dans ce qui suit. Nous allons faire un détour préliminaire par les notions de tri dont la difficulté est souvent sous-estimée avant de nous attaquer à deux manières d'implémenter les listes triées.
Au niveau abstrait, une liste est une collection d'objets de même nature dans lequel il peut y avoir des redondances. Si on veut éviter ce cas, on peut directement l'interdire ou l'autoriser au moment de la programmation. On peut aussi dans ce dernier cas créer une liste dite d' «, occurrences » qui contiendra en plus de l'élément son nombre d'occurrences (au lieu d'ajouter deux fois "toto", on ajoutera "toto" une fois avec une valeur 2 (dite de comptage) associée). Une liste peut être triée ou non. Il existe des nuances, comme les listes non triées sur lesquelles on peut opèrer un tri spécifique au moment voulu : La classe TList fournie par Delphi permet de gèrer ce genre de liste. Nous parlerons ici d'abord des listes triées directement lors de l'insertion des éléments, ce qui impose un ordre de tri fixé, puis nous modifierons l'une de nos implémentations pour permettre différents types de tris sur les éléments. Contrairement aux piles et aux files dans lesquelles l'emplacement d'insertion et de retrait des éléments est fixée, la


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

position d'insertion dépend exclusivement de l'ordre de tri pour les listes triées, et la position de retrait est imposée par l'élément qu'on souhaite supprimer (élément qu'on devra explicitement indiquer). La suppression ne devra pas non plus interférer sur le tri.
Au niveau de l'implémentation des listes, il existe plusieurs possibilités parmi lesquelles :

- les tableaux
- les listes à simple chaînage
- les listes à double chaînage

Bien que la première soit la plus simple, nous avons déjà vu son principal inconvénient et son principal avantage : respectivement une taille fixe en contrepartie d'une plus grande facilité d'implémentation. La deuxième possibilité utilise une liste à simple chaînage, ce qui peut se révèler plus intéressant mais un peu lourd ; de plus, vous venez de voir deux implémentations utilisant le chaînage simple et il est temps de passer à l'étape suivante : la liste à double, qui est la solution idéale pour les listes triées Ces listes sont plus pratique à utiliser mais moins évidentes à faire fonctionner. Nous utiliserons la première et la troisième possibilité essentiellement pour éviter de tomber dans la répétition. Après une présentation théorique, la suite du cours sera divisée en quatre : la première partie (§ 4.2) aborde des notions sur les tris. La deuxième (§ 4.3) présente une implémentation d'une liste "triée à l'insertion" par un tableau. La troisième (§ 4.4) présente une implémentation par une liste doublement chaînée. Enfin, une quatrième section (§ 4.5) viendra améliorer la précédente implémentation (§ 4.4) en supprimant le tri à l'insertion des éléments et en permettant d'effectuer un tri arbitraire sur les éléments).
Un peu de théorie maintenant. Lorsqu'on souhaite utiliser une liste, il faudra commencer par l'initialiser. Ensuite, un certain nombre d'autres opérations classiques seront possibles, parmi lesquelles l'ajout, la suppression (en spécifiant l'élément), l'obtention d'un élément par son numéro et le nombre d'éléments. Toutes ces opérations maitiendront un ordre décidé à l'avance pour les éléments. Ceci impose évidemment qu'un ordre soit choisi dans l'ensemble des éléments. Pour ceux qui n'ont pas fait une licence de mathématiques, cela signifie simplement qu'en présence de deux éléments, on devra pouvoir dire lequel doit être classé avant l'autre. La dernière opération réalisée sur une liste devra être sa suppression. Une opération de test de présence d'un élément dans une liste sera également nécessaire. Dans la pratique, ce modèle théorique pourra être adapté aux nécessités du moment en permettant par exemple la suppression d'un élément en donnant son numéroau lieu de l'élément lui-même, ce qui est souvent beaucoup plus simple.
Voici les opérations possibles sur un TAD Liste triée :

- Création d'une nouvelle liste vide
  paramètres : (aucun)
  résultat : liste vide
- Destruction d'une liste
  paramètres : une liste
  résultat : (aucun)
- Ajout d'un élément
  paramètres : une liste, un élément du type stocké dans la liste
  résultat : une liste contenant l'élément : cette opération classe l'élément à sa place parmi les autres.
- Suppression d'un élément
  paramètres : une liste, un élément du type stocké dans la liste
  résultat : une liste où l'élément transmis a été supprimé
  (Note : on se permettra souvent, dans la pratique de remplacer l'élément par son numéro)
- Nombre d'éléments
  paramètres : une liste
  résultat : un entier indiquant le nombre d'éléments dans la liste
- Accès à un élément
  paramètres : une liste, un entier
  résultat : sous réserve de validité de l'entier, l'élément dont le numéro est transmis

Comme d'habitude, nous ajouterons une procédure permettant d'afficher le contenu de la liste dans un mémo. Nous conviendrons que les listes que nous allons manipuler tolèrent les répétitions et que les éléments sont numérotés à partir de 0 (lorsqu'on aura n éléments, ils seront accessibles par les indices 0 à n-1).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XV-D-2 - Notions de tri

Si vous êtes débutant en programmation (j'espère que vous l'êtes maintenant un peu moins grâce à moi ;-), vous vous demandez sûrement pourquoi je consacre un paragraphe aux tris. En effet, nous avons tous une notion assez intuitive du tri que l'ordinateur ne connait absolument pas. Prenons par exemple la liste suivante : 1276, 3236, 2867, 13731, 138, 72, 4934. Je vous faciliterai la vie, certainement, en vous la présentant comme ceci :
1276
3236
2867
13731
138
72
4934
Mais l'ordinateur, lui, ne fera pas cette petite présentation pour y voir plus clair. Si je vous demande maintenant de classer cette liste dans l'ordre décroissant, vous allez procèder à votre manière (pensez que d'autres pourraient procèder autrement) et me répondre après rèflexion :
13731
4934
3236
2867
1276
138
72
Arrêtons nous un instant sur la méthode que vous venez d'employer : avez-vous raisonné de tête ou sur papier ? avec le bloc-notes de Windows peut-être ? L'ordinateur, lui, n'a aucune idée de ce que tout cela représente. Avez-vous été contrarié que je demande l'ordre décroissant et non l'ordre croissant ? Même pas un tout petit peu ? L'ordinateur, lui, ne se contrariera pas. Vous avez probablement cherché d'abord le plus grand élément (vous savez reconnaître le "plus grand" élément parce que je vous ai dit que l'ordre était décroissant). Pour obtenir ce plus grand élément, vous avez très probablement triché : vous avez en une vision d'ensemble des valeurs, comportement que j'ai d'ailleurs encouragé en vous présentant les valeurs d'une manière propice. L'ordinateur, lui, n'a pas de vision d'ensemble des données. Mais examinons plus avant votre raisonnement : vous avez vu un premier élément, vu qu'un autre était plus grand (vous avez donc "comparé" les deux valeurs, ce qui n'est pas forcément évident pour un ordinateur), et donc abandonné le premier au bénéfice du second, et répété ceci jusqu'à avoir parcouru la liste entière : c'est la seule et unique méthode, même si on pourrait lui trouver des variantes.
Une fois cette valeur trouvée, qu'en avez-vous fait ? Si vous êtiez sur papier, vous avez probablement recopié la valeur puis barré dans la liste. Ce faisant, vous avez entièrement reconstruit la liste : pensez à l'occupation mémoire de votre feuille de papier : si vous n'avez pas optimisé, vous avez maintenant une liste entièrement barrée (qui prend de la place, donc de la mémoire), et une liste toute neuve, qui prend autant de place que son homologue déchue. N'aurait-il pas été plus intelligent de localiser le premier élément (le "plus grand"), puis de le permuter avec le premier de la liste pour le mettre à sa place définitive, et de continuer avec le reste de la liste ? Si vous avez travaillé avec le bloc-notes de Windows, vous avez probablement appliqué une troisième méthode : après avoir localisé le "plus grand" élément, vous avez "coupé" la ligne, puis l'avez "collée" en tête, décalant ainsi les autres éléments.
Si je vous fait ce petit discours qui ressemble davantage à de l'analyse de comportement qu'à de l'informatique, c'est pour vous montrer combien l'ordinateur et vous êtes inégaux devant les tris de données. Et encore, je ne vous ai pas parlé de voitures à trier par ordre de préférence... Comprenez bien que si pour vous le tri de données est affaire d'intelligence/d'organisation, et de temps si la liste est longue, il en est de même pour l'ordinateur, à ceci près que lui se contente d'exécuter des programmes alors que vous avez un cerveau. Le principal problème, lorsqu'on s'attaque aux tris informatisés, c'est leur vitesse. Lorsque vous êtes sous Microsoft Excel, tolèreriez-vous qu'un tri sur 10000 lignes prenne 2 heures ? Bien sûr que non. Or, si nous faisions l'expérience de traduire en language informatique ce que vous venez de faire avec la petite liste de valeurs, on serait très probablement assez proches de cet ordre de grandeur (ce serait long !).
Pour accélérer les choses, informaticiens et mathématiciens ont travaillé d'arrache-pied et mis au point des techniques, aussi appelées algorithmes par les afficionados. Ces algorithmes permettent en théorie de diminuer le temps nécessaire aux tris (quoique dans certaines situations très particulières, certains algorithmes "pètent les plombs" et font perdre du temps !). Parmi les méthodes qui existent, j'ai choisi d'en présenter 4 : le tri par Sélection, le tri à Bulles ("BubbleSort"), le tri Rapide ("QuickSort") et le tri Shell ("ShellSort"). Ces quatres méthodes sont envisageables pour des listes peu volumineures, mais seules les deux dernières (la première est enviseagable mais


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

à éviter) sont à prescrire pour les listes volumineuses. Les deux dernières méthodes utilisent des algorithmes assez complexes et je n'entrerai donc pas dans les détails, histoire de ne pas vous noyez dans des détails techniques. A la place, j'expliquerai l'idée générale de la méthode.
Toutes ces méthodes permettent de séparer la partie "intelligence" (l'algorithme, exprimé sous formes d'instructions Pascal), des traitements mécaniques. Ces derniers sont réduits au nombre de deux : la comparaison de deux éléments, et la permutation de deux éléments. En séparant ainsi l'"intelligence" de la partie "mécanique", on se complique certes un peu la vie, mais on se donne la possibilité d'utiliser la partie "intelligence" avec divers types de données, uniquement identifiés au niveau "mécanique". Pour vous faire comprendre ceci, pensez aux impôts sur le revenu : l'état les collecte, et délègue les détails de la collecte : l'état est dans notre exemple la partie "intelligence". Pour chacun d'entre nous, payer les impôts sur le revenu passe par une déclaration au format papier : c'est la partie "mécanique" de l'exemple. Imaginez demain que nous faisions tous cette déclaration sur Internet, il n'y aurait plus de papier, et donc la partie "mécanique" changerait, mais la partie "intelligence" resterait inchangée : l'impôt sera toujours collecté.
Pour en revenir à l'informatique, le tri par Sélection est celui qui se rapproche le plus de ce que nous faisons habituellement. Le principe est à chaque étape d'identifier l'élément à placer en tête de liste, et de permuter le premier élément de la liste avec cet élément. On poursuit le tri en triant uniquement le reste de la liste de la même manière (si la liste contenait n éléments, on trie les n-1 derniers éléments). Voici un tableau montrant les diverses étapes de ce genre de tri sur notre petite liste d'éléments, avec à chaque fois mentionné le nombre de comparaisons nécessaires :

| 1276 | 72 | 72 | 72 | 72 | 72 | 72 |
|---|---|---|---|---|---|---|
| 3236 | 3236 | 138 | 138 | 138 | 138 | 138 |
| 2867 | 2867 | 2867 | 1276 | 1276 | 1276 | 1276 |
| 13731 | 13731 | 13731 | 13731 | 2867 | 2867 | 2867 |
| 138 | 138 | 3236 | 3236 | 3236 | 3236 | 3236 |
| 72 | 1276 | 1276 | 2867 | 13731 | 13731 | 4934 |
| 4934 | 4934 | 4934 | 4934 | 4934 | 4934 | 13731 |
| comparaisons | 6 | 5 | 4 | 3 | 2 | 1 |

On arrive à la liste triée avec 6 permutations et 21 comparaisons, ce qui peut paraître bien, mais qui aura tendance à grimper de manière dramatique : avec 10000 éléments, il faudrait 9999 permutations et 49985001 comparaisons.
Le tri à Bulles fonctionne sur un principe différent et son algorithme est des plus faciles à comprendre : la liste est considèrée comme des éléments placés de haut (tête de liste) en bas (queue de liste), un peu comme sur du papier ou un écran d'ordinateur. On parcours la liste du "bas" vers le "haut" (le "haut" s'abaissera d'un élément à chaque fois puisqu'à chaque itération, l'élément arrivant en première position est trié) et à chaque étape, un élément, considéré comme une "bulle", est remonté en début de liste : comme une bulle qui remonterait à la surface de l'eau, on remonte progressivement dans la liste et dés que deux éléments consécutifs sont dans le désordre, on les permute (sinon on ne fait rien et on continue à remonter). En fait, à chaque parcours, plusieurs éléments mal triés sont progressivement remontés à une place temporaire puis définitive au fil des itérations. Voici ce que cela donne avec notre petite liste, mais uniquement pour la première itération :
Etape 1 : remontée de la bulle n°1 : on regarde 4934 (élément n°7) 4934 et 72 sont mal triés, on les permute. On regarde l'élément n°6 : 4934 et 138 sont mal triés, on les permute. On regarde l'élément n°5 : 4934 et 13731 sont bien triés. On regarde l'élément n°4 : 13731 et 2867 sont mal triés, on les permute. On regarde l'élément n°3 : 3236 et 13731 sont mal triés, on les permute. On regarde l'élément n°2 : 1276 et 13731 sont mal triés, on les permute.

| 1276 | 1276 | 1276 | 1276 | 1276 | 13731 |
|---|---|---|---|---|---|
| 3236 | 3236 | 3236 | 3236 | 13731 | 1276 |
| 2867 | 2867 | 2867 | 13731 | 3236 | 3236 |
| 13731 | 13731 | 13731 | 2867 | 2867 | 2867 |
| 138 | 138 | 4934 | 4934 | 4934 | 4934 |
| 72 | 4934 | 138 | 138 | 138 | 138 |
| 4934 | 72 | 72 | 72 | 72 | 72 |

Comme vous le voyez, le premier élément, 13731, est bien trié (il est à sa place définitive). A l'étape suivante, on appliquera la méthode aux six derniers éléments et ainsi de suite jusqu'à avoir trié toute la liste. Comme vous vous en rendez probablement compte, les performances de cet algorithme ne sont vraiment pas excellentes et il ne vaut mieux pas dépasser quelques centaines d'éléments sous peine de mourrir d'ennui devant son ordinateur.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'algorithme du Tri Shell, généralement plus rapide que le tri par sélection, compare d'abord des éléments éloignés de la listr à trier, puis réduit l'écart entre les éléments à trier, ce qui diminue très rapidement le nombre de comparaisons et de permutations à effectuer. L'application en téléchargement ci-dessous implémente un tri shell sur un tableau d'entiers. Je parle de cette application un peu plus bas. Le tri Rapide, enfin, utilise une technique de "partition" de la liste à trier : un élément "pivot" est choisi dans la liste et les éléments se retrouvant du mauvais coté de ce pivot sont permutés, puis chaque moitié est triée à son tour. Cet algorithme est un des plus rapides, mais il arrive qu'il "dégénère", c'est-à-dire qu'il mette un temps incroyablement long à trier peu d'éléments.
Pour que vous puissiez observer et juger par vous-même les différentes méthodes de tri abordées ici, je vous ai concocté une petite application. Téléchargez le **code source** du projet, examinez-le un peu, et essayez les divers tris (un petit conseil : évitez plus de 300 valeurs pour le BubbleSort et 3000 pour le tri par Sélection). N'hésitez pas à modifier le code source à votre guise ou à réutiliser des morceaux de code dans vos applications.
Le but de ce paragraphe étant avant tout de vous initier à la dure réalité des tris de données, je n'entre pas dans trop de détails techniques pas plus que je ne donne les algorithmes des méthodes évoquées. Internet regorge d'exemples et d'implémentations de chacun de ces tris. Sachez enfin que la méthode Sort de la classe TList utilise la méthode QuickSort pour trier les données. Pour fonctionner, cette méthode a besoin que vous lui donniez le nom d'une fonction de comparaison. Ce genre de pratique étant encore un peu délicate à comprendre, je la réserve pour plus tard (§ 4.5).

## XV-D-3 - Implémentation par un tableau

Nous allons réaliser une première implémentation d'une liste triée. Le type de tri sera figé et donc effectué lors de l'iinsertion des éléments. Nous allons ici utiliser un tableau pour stocker les éléments de la liste, tout en essayant d'adopter une structure de données plus élaborée que celles utilisées pour les piles et les files. Voici les types utilisés :

```
const
  MAX_ELEM_LISTE_TRIEE = 200;

type
  // un type destiné à être stocké dans la liste
  TPersonne = record
    Nom, Prenom: string;
    Age: integer;
  end;

  // ElemListe pourraît être changé sans trop de difficultés
  PPersonne = ^TPersonne;

  // enreg. contenant une liste triée
  _ListeTabTriee = record
    // nombre d'éléments
    NbElem: integer;
    // éléments (pointeurs)
    Elem: array[1..MAX_ELEM_LISTE_TRIEE] of PPersonne;
  end;

  { le vrai type : c'est un pointeur, beaucoup plus indiqué
    qu'un simple enregistrement ou un tableau. }
  TListeTabTriee = ^_ListeTabTriee;
```

Le premier type ci-dessus (TPersonne), est choisi arbitrairement pour cet exemple. Nous utilisons un type intermédiaire nommé PPersonne qui est un pointeur vers un élément de type TPersonne. Les éléments stockés seront donc non pas réellement les éléments de type TPersonne mais des pointeurs vers ces éléments. Le type utilisé pour implémenter la liste est un enregistrement. Le champ NbElem nous permettra de mémoriser le nombre d'éléments présents dans la liste. Le tableau Elem stockera les pointeurs de type PPersonne vers les éléments de type TPersonne. Il faut savoir que le choix effectué pour les piles et les files d'utiliser un type enregistrement seul n'est en fait pas une très bonne idée. Il est préférable d'utiliser un pointeur vers un tel élément. Le type utilisé sera alors TListeTabTriee qui est un pointeur vers un enregistrement stockant une liste triée.
Nous allons maintenant implémenter les diverses opérations de manipulation d'un tel type de donnée. En tout premier, les deux opérations de création et de destruction d'une liste :

```
function LTTNouvelle: TListeTabTriee;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
begin
  new(result);
  result^.NbElem := 0;
end;

procedure LTTDetruire(Liste: TListeTabTriee);
var
  i: integer;
begin
  if Liste <> nil then
    begin
      for i := 1 to Liste^.NbElem do
        Dispose(Liste^.Elem[i]);
      Dispose(Liste);
    end;
end;
```

Dans le code ci-dessus, un élément de type _ListeTabTriee est créé par New, puis le nombre d'éléments est initialisé à 0. Du fait de l'utilisation de pointeurs, aucune autre action initiale comme une allocation de mémoire n'est nécessaire. La suppression commence par vérifier que la liste transmise n'est pas le pointeur **nil** (c'est la seule chose qu'on peut tester facilement quant à la validité de la liste) puis, dans ce cas, commence par libèrer toute la mémoire allouée aux éléments de type TPersonne (nous choisissons de gèrer la mémoire associée aux éléments de la liste depuis les opérations de manipulation, ce qui évite, depuis l'extérieur, des allocations mémoire fastidieuses). Enfin, la mémoire associée à l'élément de type _ListeTabTriee est libérée à son tour (comprenez bien qu'on ne peut pas faire cela au début, car sinon plus d'accès possible aux pointeurs vers les éléments qu'on souhaite retirer de la mémoire).
Passons aux opérations de Compte d'élément, de test de liste vide ou pleine. Ces opérations sont très simples :

```
function LTTCompte(Liste: TListeTabTriee): integer;
begin
  // on prend comme convention -1 pour une liste incorrecte
  if Liste = nil then
    result := -1
  else
    result := Liste^.NbElem;
end;

function LTTVide(Liste: TListeTabTriee): boolean;
begin
  result := true;
  if Liste <> nil then
    result := Liste^.NbElem = 0;
end;

function LTTPleine(Liste: TListeTabTriee): boolean;
begin
  result := true;
  if Liste <> nil then
    result := Liste^.NbElem = MAX_ELEM_LISTE_TRIEE;
end;
```

La première opération retourne -1 si le pointeur transmis est incorrect, ou renvoie la valeur du champ NbElem dans le cas contraire. La seconde renvoie vrai si la valeur NbElem est égale à 0. Enfin, la troisième renvoie vrai si NbElem vaut la valeur maximale fixée par la constante MAX_ELEM_LISTE_TRIEE. Passons tout de suite à l'insertion d'un élément. L'insertion se fait en trois étapes : un emplacement mémoire est créé pour stocker l'élément transmis. Cet élément sera référencé par un pointeur stocké dans le tableau Elem de la liste. La seconde étape consiste à déterminer la position d'insertion de l'élément dans la liste. Enfin, la troisième étape consiste en l'insertion proprement dite. Voici la procédure complète, que j'explique ci-dessous, ainsi que la fonction de comparaison de deux éléments de type TPersonne :

```
// fonction de comparaison de deux "personnes"
function CompPersonnes(P1, P2: PPersonne): integer;
begin
  // 0 est le résultat par défaut
  result := 0;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  // si erreur dans les paramètres, sortie immédiate
  if (P1 = nil) or (P2 = nil) then exit;
  // on compare d'abord les noms
  result := AnsiCompareText(P1^.Nom, P2^.Nom);
  // s'ils sont égaux, on compare les prénoms
  if result = 0 then
    result := AnsiCompareText(P1^.Prenom, P2^.Prenom);
  // s'ils sont égaux, c'est l'écart d'âge qui fixe le résultat
  if result = 0 then
    result := P1^.Age - P2^.Age;
end;
```

Cette fonction retourne un résultat négatif si le nom de P1 est "inférieur" à celui de P2 au sens de la fonction "AnsiCompareText". Si vous consultez l'aide de cette fonction, vous constaterez qu'elle fait tout le travail de comparaison et sait comparer des valeurs alphanumériques. AnsiCompareText(A, B) retourne un résultat inférieur à 0 si A < B et un résultat positif sinon. Un résultat nul est obtenu en cas d'égalité. On compare d'abord les noms, puis en cas d'égalité les prénoms, et enfin les âges au moyen d'une simple soustraction (résultat encore négatif si A < B).

```
function LTTAjout(Liste: TListeTabTriee; Pers: TPersonne): TListeTabTriee;
var
  PersAj: PPersonne;
  PosInser, i: integer;
begin
  result := Liste;
  if Liste = nil then exit;
  // liste pleine, impossible d'ajouter un élément
  if Liste^.NbElem = MAX_ELEM_LISTE_TRIEE then exit;
  // création de l'élément
  New(PersAj);
  PersAj^ := Pers;
  // recherche de la position d'insertion
  if Liste^.NbElem = 0 then
    PosInser := 1
  else
    begin
      PosInser := Liste^.NbElem + 1; // Inséré par défaut en fin de liste
      // diminue la position jusqu'à ce qu'un élément rencontré soit inférieur strictement
      while (PosInser > 1) and (CompPersonnes(PersAj, Liste^.Elem[PosInser - 1]) < 0) do
        dec(PosInser);
    end;
  { deux cas sont possibles :
    1) l'élément est inséré en fin de liste et alors tout va bien
    2) l'élément doit être inséré à la place occupée par un autre élément, il faut
       donc décaler les éléments suivants pour libèrer une place }
  if PosInser <= Liste^.NbElem then
    // décalage des éléments
    for i := Liste^.NbElem downto PosInser do
      Liste^.Elem[i+1] := Liste^.Elem[i];
  // insertion
  Liste^.Elem[PosInser] := PersAj;
  Inc(Liste^.NbElem);
end;
```

Les 5 premières instructions font la première étape et quelques tests d'usage, entre autres, si la liste est pleine, l'ajout est refusé. Une zone mémoire est créé (new) et initialisée (affectation). Le bloc **if** se charge de déterminer la position d'insertion de l'élément. Si la liste est vide, la position est 1, sinon, on part de l'hypothèse de l'insertion en fin de liste, puis on décroit cette position jusqu'à rencontrer un élément précédent cette position et qui soit inférieur à l'élément à insèrer. Ainsi, si la liste contenait des nombres (1, 3 et 7 par exemple) et si on désirait insèrer 2, la position d'insertion sera celle de 3.
La troisième et dernière étape effectue, sauf cas d'ajout en fin de liste, un décalage des pointeurs pour libèrer une place à celui qui doit être inséré. Notez l'ordre de parcours et les éléments parcourus : un parcours dans l'ordre inverse aurait des conséquences fâcheuses (essayez sur papier, vous verrez que les éléments sont tous supprimés avant d'être lus). Enfin, l'élément est placé dans le tableau d'éléments. Notez que cette méthode d'ajout est optimisable car la recherche dans une liste triée peut s'effectuer beaucoup plus rapidement que dans une liste non triée. Ainsi,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

une recherche dichotomique pourrait gagner énormément de temps sur de grosses listes, mais comme nous nous limitons à 200 éléments, ce n'est pas assez intéressant.
Nous avons également besoin d'une fonction permettant de supprimer un élément de la liste. Pour simplifier, puisqu'il nous serait difficile de fournir l'élément à supprimer, on fournit plutôt son numéro. Voici la fonction qui réalise cela :

```
function LTTSupprIdx(Liste: TListeTabTriee; index: integer): TListeTabTriee;
var
  i: integer;
begin
  result := Liste;
  if Liste = nil then exit;
  if (index < 1) or (index > Liste^.NbElem) then exit;
  Dispose(Liste^.Elem[index]);
  for i := index to Liste^.NbElem - 1 do
    Liste^.Elem[i] := Liste^.Elem[i+1];
  Dec(Liste^.NbElem);
end;
```

Les premières instructions effectuent quelques vérifications. L'élément à supprimer est ensuite libéré de la mémoire, puis les éléments suivants dans le tableau sont décalés pour combler la place laissée vide par l'élément supprimé.
Enfin, voici les fonctions et procédures permettant l'affichage du contenu de la liste dans une "ListBox" ou un "Memo" :

```
// fonction renvoyant une chaîne décrivant une personne
function AffichPers(P: PPersonne): string;
begin
  if P = nil then
    result := ''
  else
    result := P^.Nom + ' ' + P^.Prenom + ' (' + IntToStr(P^.Age) + ' ans)';
end;

procedure AfficheListe(Liste: TListeTabTriee; Sortie: TStrings);
var
  i: integer;
begin
  Sortie.Clear;
  if Liste = nil then
    Sortie.Add('(Liste incorrecte)')
  else
    for i := 1 to Liste^.NbElem do
      Sortie.Add(AffichPers(Liste^.Elem[i]));
end;
```

L'affichage d'une personne est effectué via une fonction qui crée une chaîne à partir d'une personne. Cette chaîne est ensuite générée pour chaque personne et ajoutée comme d'habitude à la sortie de type TStrings. Venons-en maintenant au programme de test. Pour plus de facilité, **téléchargez-le maintenant** et suivez à partir du code source, car je ne vais décrire que ce qui est essentiel à la compréhension de l'utilisation des listes.
L'interface est très conventionnelle. Notez cependant que le mémo que nous utilisions auparavant a été remplacé par une zone de liste (ListBox) permettant d'avoir facilement accès au numéro de ligne sélectionnée (propriété ItemIndex). Une petite fiche de saisie des coordonnées d'une personne fait également partie de l'interface. Une variable "Lst" de type TListeTabTriee a été déclarée dans l'unité de la fiche principale du projet. Remarquez, si vous avez lu le chapitre concernant les objets, qu'on aurait tout aussi bien pû faire de cette variable un champ de l'objet fmPrinc. J'ai choisi de ne pas le faire tant que je n'aurai pas abordé les objets de manière plus sérieuse.
Comme dans les précédents projets exemple, une procédure MajInterface a pour charge de mettre l'interface principale à jour en réactualisant le contenu de la liste et en dés(activant) les boutons nécessaires. La procédure associée au bouton "Ajouter" affiche d'abord la fenêtre de saisie (l'événement OnShow de cette fiche se charge de la réinitialiser à chaque affichage), puis récupère les valeurs saisies pour en faire un élément de type TPersonne, transmis à l'opération d'ajout. Les procédurs associées aux événements OnCreate et OnDestroy de la fiche principale permettent respectivement d'initialiser et de détruire la liste Lst. Pour retirer un élément, il faut le sélectionner et cliquer sur "Supprimer". L'action de ce bouton consiste à tester la propriété ItemIndex de la liste (valeur -1 si aucun élément sélectionné, numéro commencant à 0 sinon) et selon le cas supprime ou affiche un message d'erreur. Le bouton permettant de vider la liste se contente de supprimer les éléments en commençant par le dernier. J'ai un peu


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

exagéré les choses pour vous montrer ce qu'il vaut mieux éviter de faire : cette méthode de suppression tient presque compte de la structure interne de la liste, ce qui est fort regretable. Vous pouvez corriger cela à titre d'exercice et me contacter en cas de problème.
Voilà pour l'implémentation sous forme de tableau. Vous avez peut-être trouvé le rythme un peu élevé, mais il est nettement plus important de comprendre les mécanismes de base et les notions abstraits plutôt que les détails d'une implémentation.

## XV-D-4 - Implémentation par une liste chaînée

Nous allons reprendre la même interface de test que précédemment sans changer beaucoup de code concernant les listes. L'unité utilisée va être réécrite complètement en utilisant le procédé de double chaînage, histoire à la fois de vous donner une autre implémentation d'une liste triée mais aussi de vous familiariser avec cette notion (pas si évidente que ça) qu'est le double chaînage. La liste "Lst" sera déclarée d'un nouveau type, les procédures et fonctions vont également changer un tout petit peu de nom, mais il ne devrait pas y avoir grand changement. Je vais me permettre un petit complément de cours concernant le double chaânage pour ceux à qui cette notion est étrangère.
Le double chaînage est, dans les grandes lignes, un simple chaînage dans les deux sens. Contrairement au simple chaînage où chaque pointeur ne pointe que sur l'élément suivant dans la liste, avec le procédé de double chaînage, chaque élément pointe à la fois vers un élément "précédent" et vers un élément "suivant" (les deux exceptions sont le premier et le dernier élément. Il est possible, en « raccordant » la fin et le début d'une liste à double chaînage, d'obtenir une liste circulaire très pratique dans certains cas, mais c'est hors de notre propos.
Concrêtement, voici une déclaration en Pascal Objet d'un « maillon » d'une liste à double chaînage simplifiée (les éléments stockés sont des entiers, qu'on remplacera par la suite par des pointeurs, histoire d'attaquer un peu de vrai programmation ;-)

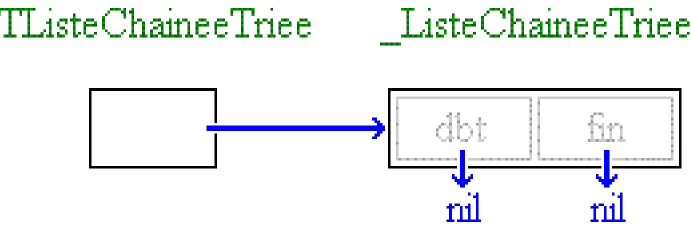
```
type
  PMaillon = ^TMaillon;
  TMaillon = record
    Elem: integer;
    Suiv: PMaillon;
    Pred: PMaillon;
  end;
```

Vous voyez que l'on déclare un pointeur général, puis que l'élément pointé par le pointeur peut a son tour contenir deux pointeurs vers deux autres éléments. Ce genre de maillon, tel quel, n'est pas très utile, utilisons donc un élément de type PPersonne à la place de l'entier et déclarons un type utilisable de l'extérieur :

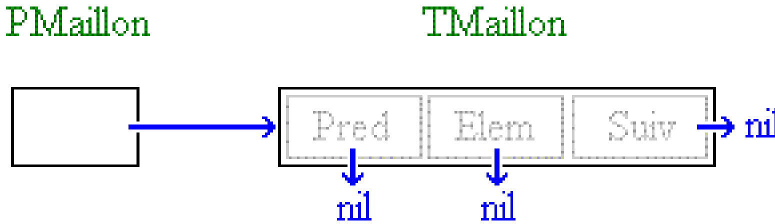
```
type
  // un type destiné à être stocké dans la liste
  PPersonne = ^TPersonne;
  TPersonne = record
    Nom, Prenom: string;
    Age: integer;
  end;

  // Maillon d'une liste à double chaînage
  PMaillon = ^TMaillon;
  TMaillon = record
    Elem: PPersonne;
    Suiv: PMaillon;
    Pred: PMaillon;
  end;

  // Une liste à double chaînage sera représentée par un pointeur vers le premier élement.
  TListeChaineeTriee = ^_ListeChaineeTriee;
  _ListeChaineeTriee = record
    Dbt: PMaillon;
    Fin: PMaillon;
  end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

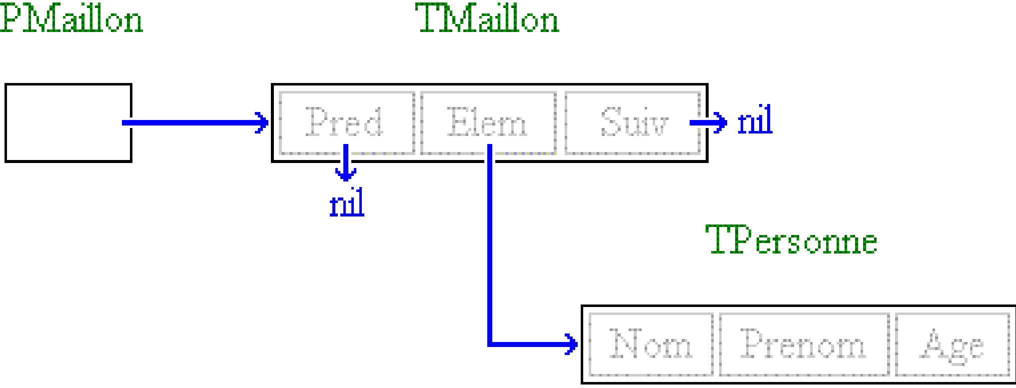
Voici une représentation plus intuitive d'une liste vide sous forme de schémas. Vous pouvez voir l'élément de type TListeChaineeTriee intialisée et pointant vers un élément _ListeChaineeTriee également initialisé (champs Dbt et Fin à **nil**) :




Voici maintenant le schémas d'un chaînon non initialisé : son champ Elem ne pointe sur rien et ses champs Suiv et Pred non plus :




Enfin, voici le schémas d'un maillon dont le champ Elem est initialisé et pointe donc sur un élément de type TPersonne. Notez que le pointeur de type PPersonne n'est plus une variable à part mais un champ du maillon. Cependant, les champs Pred et Suiv ne sont pas encore initialisé, car ce maillon n'est pas encore intégré à une liste.




De l'extérieur, seul le type "TListeChaineeTriee" devra être utilisé. Nous allons petit à petit implémenter les opérations de manipulation de ces listes. Commençons par la plus simple : la création d'une liste vide, qui consiste à renvoyer un pointeur vers un record de type _ListeChaineeTriee. On conserve un pointeur vers la fin et un vers le début pour permettre de commencer le parcours de la liste depuis la fin (en utilisant les pointeurs Pred) ou depuis le début (en utilisant les pointeurs Suiv).

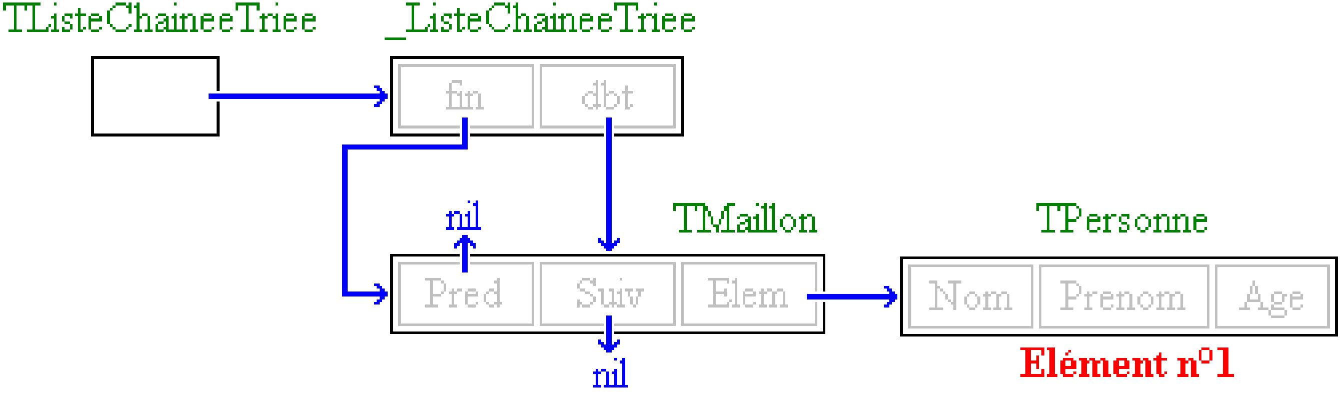
```
function LCTNouvelle: TListeChaineeTriee;
begin
  new(result);
  result^.Dbt := nil;
  result^.Fin := nil;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

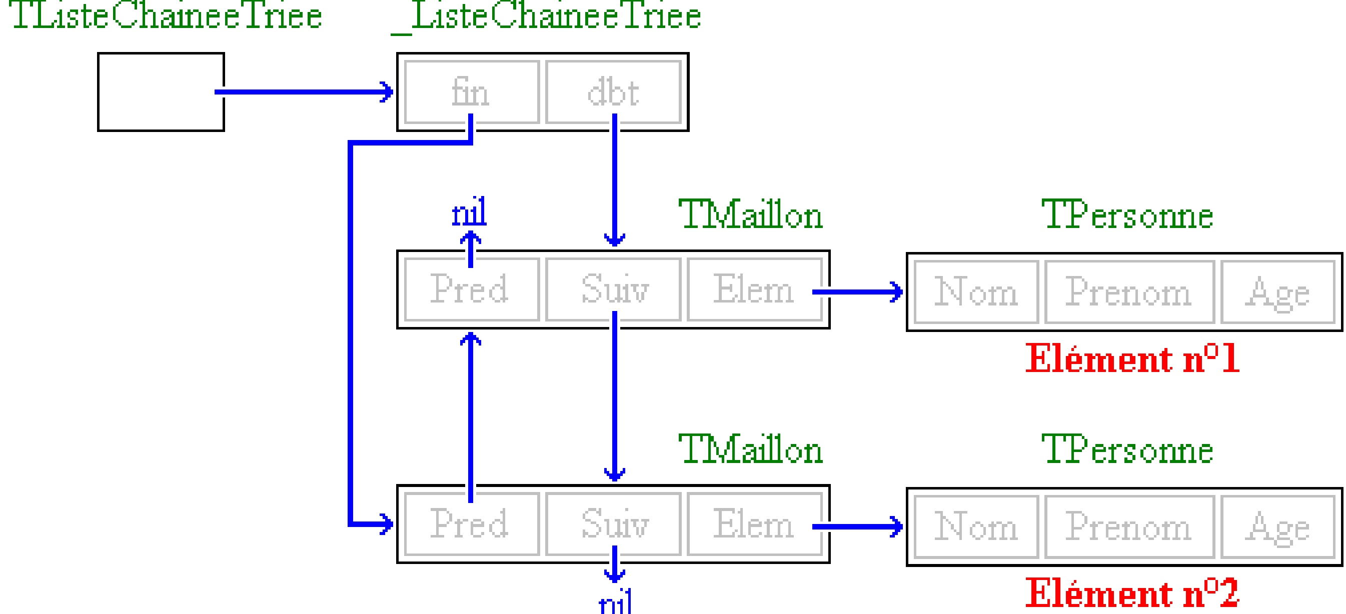
L'opération de test de liste pleine n'a pas de sens ici. On n'implémente donc que celui qui teste si une liste est vide. En pratique, si l'élément est nil, la liste est considérée comme vide, sinon, on teste les deux pointeurs Fin et Dbt et s'ils valent **nil** tous les deux, la liste est vide.

```
function LCTVide(Liste: TListeChaineeTriee): boolean;
begin
  result := Liste = nil;
  if not result then
    result := (Liste^.Dbt = nil) and (Liste^.Fin = nil);
end;
```

Avant de passer à la manipulation des éléments, nous allons voir encore une fois à l'aide de schémas comment les listes manipulées ici sont constituées. Nous détaillerons par la suite la procédure d'insertion ou de retrait d'un élément. Voici une liste à un seul élément (Attention, pour faciliter le dessin et sa compréhension, la position de certains champs (Elem et Suiv, Fin et Dbt) seront dorénavant permutées) :

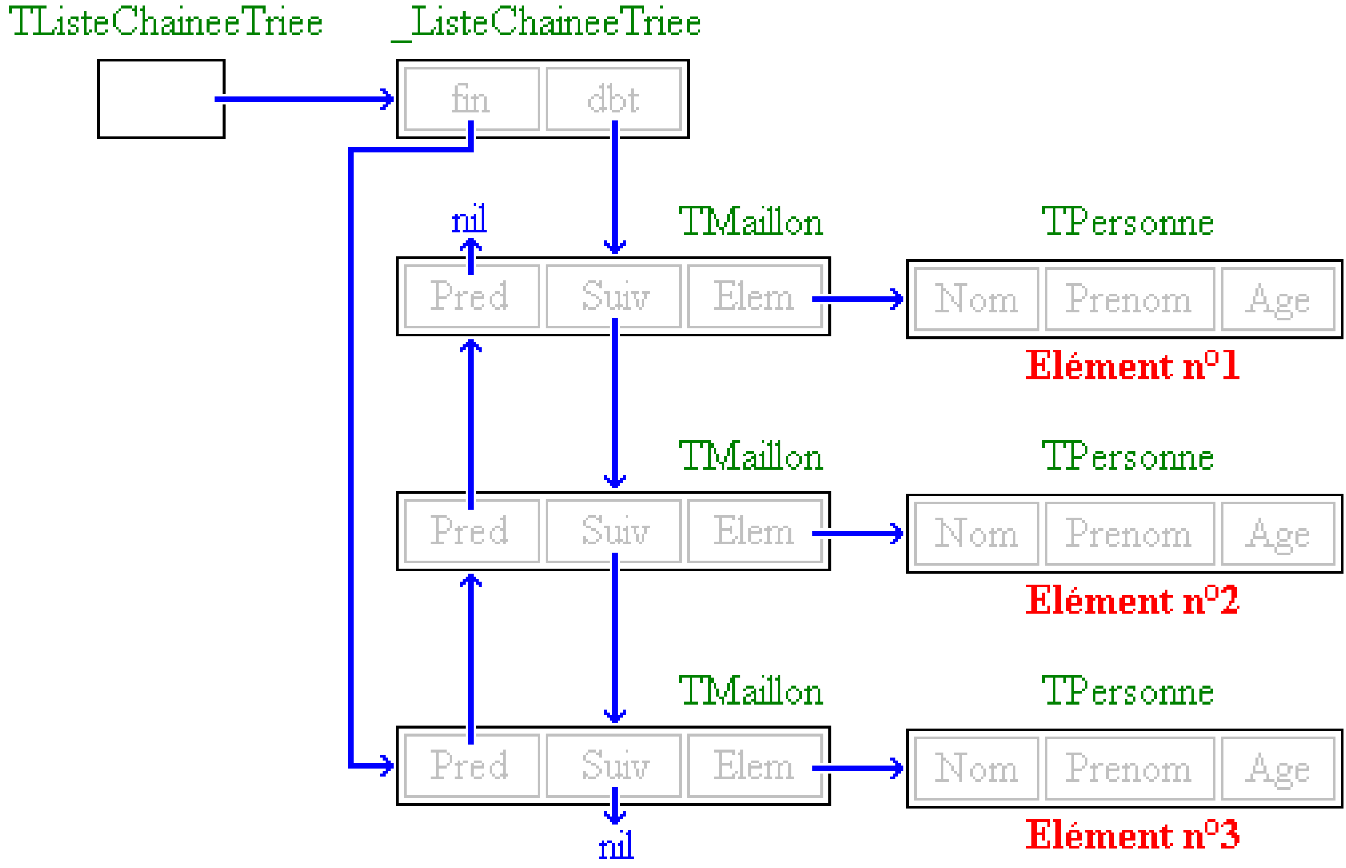



Les étapes indispensables pour construire cette structure seront présentées et expliquées ci-dessous. En attendant, compliquons un peu les choses en ajoutant un élément :




Et, puisque nous y sommes, histoire de vous faire sentir la disposition générale, voici une liste à 3 éléments :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Maintenant que vous voyez à quoi ressemble la structure d'une liste, passons à l'opération d'insertion d'un élément. Plutôt que de commencer en vous lâchant l'insertion avec tri d'une façon brutale, nous allons progresser doucement en insérant au début, à la fin, puis au milieu d'une liste à double chaînage pour bien vous faire comprendre les mécanismes à acquérir. Dans chaque cas, il faut faire attention car le cas où la liste est vide est particulier et doit être traité à part. Voici ce que cela donne (Ne vous effrayez pas) :

```
function LCTInsereDebut(Liste: TListeChaineeTriee; P: TPersonne): TListeChaineeTriee;
var
  Temp: PMaillon;
begin
  result := Liste;
  if Liste = nil then exit;
  // création d'un nouvel élément (maillon)
  New(Temp);
  New(Temp^.Elem);
  // initialisation du nouveau maillon
  Temp^.Elem^ := P;
  // 2 cas, si la liste est vide ou non
  if LCTVide(Liste) then
    // initialise une liste à 1 élément (début et fin)
    begin
      Temp^.Pred := nil;
      Temp^.Suiv := nil;
      Liste^.Dbt := Temp;
      Liste^.Fin := Temp;
    end
  else
    // insertion avant le premier élément
    begin
      // l'ancien premier élément doit être mis à jour
      Liste^.Dbt^.Pred := Temp;
      Temp^.Suiv := Liste^.Dbt;
      Temp^.Pred := nil;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
      Liste^.Dbt := Temp;
      // Liste^.Fin ne change pas
    end;
end;
```

L'insertion doit effectuer plusieurs traitements : le premier consiste à créer le maillon, à créer la mémoire associée au maillon et à recopier les valeurs à mémoriser dans cette mémoire. Une fois le maillon créé, il faut l' « accrocher » en début de liste. Ici, deux cas sont possibles : soit la liste est vide, soit elle ne l'est pas. Ces deux cas nécessitent un traitement à part. Si la liste est vide, Les champs Pred et Suiv sont fixés à **nil** (cf. illustration de la liste vide ci-dessus) car il n'y a pas d'autre élément dans la liste : le premier élément est aussi le dernier et le premier a toujours son pointeur Pred égal à **nil** et de même pour le pointeur Suiv du dernier élément. Du fait que l'élément inséré est le premier et le dernier de la liste, les champs Dbt et Fin pointent vers cet élément.

⚠ *Remarque : il aurait été possible de faire pointer le champ Pred du premier élément sur le dernier, et le champ Suiv du dernier élément sur le premier, ce qui nous aurait donné une liste « circulaire », un peu plus pratique à manipuler, mais j'ai préféré ne pas le faire pour ne pas entraîner de confusion inutile.*

Si la liste n'est pas vide, le traitement est un peu plus compliqué : l'élément en tête de liste avant l'insertion doit être mis à jour pour que son pointeur Pred, initialement à **nil**, pointe sur le nouvel élément inséré en tête de liste, qui le précédera donc immédiatement dans la liste. La première instruction réalise cela. Le champ Suiv du nouvel élément inséré est fixé à l'ancien premier élément, encore accessible par "Liste^.Dbt", que l'on modifiera par la suite. Quand au champ Pred, il est fixé à **nil** car on insère en début de liste. Enfin, l'insertion du premier élément est concrétisée par la modification du pointeur "Dbt" de la liste.
Intéressons-nous à l'ajout en fin de liste. Il ressemble beaucoup à l'ajout en début de liste, puisque c'est en quelque sorte son « symétrique ». Voici le code source :

```
function LCTInsereFin(Liste: TListeChaineeTriee; P: TPersonne): TListeChaineeTriee;
var
  Temp: PMaillon;
begin
  result := Liste;
  if Liste = nil then exit;
  // création d'un nouvel élément (maillon)
  New(Temp);
  New(Temp^.Elem);
  // initialisation du nouveau maillon
  Temp^.Elem^ := P;
  // 2 cas, si la liste est vide ou non
  if LCTVide(Liste) then
    // initialise une liste à 1 élément (début et fin)
    begin
      Temp^.Pred := nil;
      Temp^.Suiv := nil;
      Liste^.Dbt := Temp;
      Liste^.Fin := Temp;
    end
  else
    // insertion après le dernier élément
    begin
      // l'ancien dernier élément doit être mis à jour
      Liste^.Fin^.Suiv := Temp;
      Temp^.Pred := Liste^.Fin;
      Temp^.Suiv := nil;
      Liste^.Fin := Temp;
      // Liste^.Dbt ne change pas
    end;
end;
```

Sans écrire la procédure qui le réalise, analysons l'insertion d'un élément en « milieu » de liste, c'est-à-dire entre deux éléments M1 et M2 déjà présents dans la liste. On appellera pour cela que l'élément inséré M3. Il y a 4 « connections » à réaliser :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

1 fixer le champ Suiv du maillon M3 à l'adresse de M2 ;
2 fixer le champ Prev du maillon M3 à l'adresse de M1 ;
3 fixer le champ Suiv du maillon M1 à l'adresse de M3 ;
4 fixer le champ Prev du maillon M2 à l'adresse de M3 ;

La procédure d'insertion va donc consister à parcourir la liste depuis son début, et dés que l'endroit où l'élément doit être inséré est déterminé, on réalise l'insertion adéquate. Cette recherche n'a d'ailleurs pas lieu si la liste est initialement vide. Voici le code source ; prenez bien le temps de l'analyser, il n'est pas si long qu'il n'y paraît, car je n'ai pas été avare de commentaires :

```
function LCTInsere(Liste: TListeChaineeTriee; P: TPersonne): TListeChaineeTriee;
var
  Temp, Posi: PMaillon;
begin
  result := Liste;
  if Liste = nil then exit;
  // 1. Création du maillon
  New(Temp);
  New(Temp^.Elem);
  // initialisation du nouveau maillon par le paramètre P
  Temp^.Elem^ := P;
  // 2. Si la liste est vide, on insère l'élément seul
  if LCTVide(Liste) then
    // initialise une liste à 1 élément (début et fin)
    begin
      Temp^.Pred := nil;
      Temp^.Suiv := nil;
      Liste^.Dbt := Temp;
      Liste^.Fin := Temp;
      exit;
    end;
  // 3. On recherche la position d'insertion de l'élément
  { pour cela, on fixe un pointeur au dernier élément de la liste
    puis on recule dans la liste jusqu'à obtenir un élément
    "intérieur" à celui inséré. 3 cas se présentent alors :
    1) on est en début de liste : on sait faire l'ajout
    2) on est en fin de liste : idem
    3) sinon, on applique la méthode décrite ci-dessus. }
  // initialisation du parcours
  Posi := Liste^.Fin;
  { tant que l'on peut parcourir et que l'on DOIT parcourir...
    (on peut parcourir tant que Posi est non nil et on DOIT
    parcourir si la comparaison entre l'élément actuel et l'élément
    à insèrer donne le mauvais résultat. }
  while (Posi <> nil) and (CompPersonnes(Posi^.Elem, Temp^.Elem) > 0) do
    Posi := Posi^.Pred;
  // 4. c'est maintenant qu'on peut détecter l'un des trois cas.
  // cas 1 : Posi vaut nil, on a parcouru toute la liste
  if Posi = nil then
    begin
      // insertion au début
      Liste^.Dbt^.Pred := Temp;
      Temp^.Suiv := Liste^.Dbt;
      Temp^.Pred := nil;
      Liste^.Dbt := Temp;
    end
  // cas 2 : Posi n'a pas changé, et vaut donc Liste^.Fin.
  else if Posi = Liste^.Fin then
    begin
      // insertion à la fin
      Liste^.Fin^.Suiv := Temp;
      Temp^.Pred := Liste^.Fin;
      Temp^.Suiv := nil;
      Liste^.Fin := Temp;
    end
  // cas 3 : Posi <> Liste^.Fin et <> nil, insertion "normale"
  else
    begin
```



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
      { Posi pointe sur un élément "inférieur", on doit donc
        insérer entre Posi et Posi^.Suiv }
      Temp^.Suiv := Posi^.Suiv;
      Temp^.Pred := Posi;
      Posi^.Suiv^.Pred := Temp;
      Posi^.Suiv := Temp;
    end;
end;
```

En avant les explications !
Il est à noter ici que l'ordre considéré est le même qu'au paragraphe précédent, j'ai donc repris la fonction de comparaison. La première partie est très conventionnelle puisqu'elle se contente de faire quelques tests et initialise le maillon à insèrer. La section 2 traite le cas où la liste est vide. Dans ce cas, l'ordre n'a pas d'influence, et on se contente d'initialiser une liste à un élément, chose que nous avons déjà fait deux fois précédemment. La partie 3 est plus intéressante car elle effectue la recherche de la position d'insertion du maillon. On adopte la même technique que pour l'implémentation avec un tableau : on part du dernier élément et on "remonte" dans la liste jusqu'à obtenir un élément plus petit que celui à insèrer. Si l'élément est plus grand que le dernier de la liste, le parcours s'arrètera avant de commencer car la première comparaison "cassera" la boucle. Si l'élément est plus petit que tous ceux présents dans la liste, c'est la condition Posi <> **nil** qui nous permettra d'arrêter la boucle lorsque l'on sera arrivé au début de la liste et qu'il sera devenu impossible de continuer la "remontée", faute d'éléments. Le troisième cas, intermédiaire, est lorsqu'un élément parmi ceux de la liste est plus petit que celui inséré. Il y a alors arrêt de la boucle et Posi pointe sur cet élément.
L'étape 4 traite ces trois cas à part. La distinction des cas se fait en examinant la valeur de Posi. Si l'élément est plus grand que le dernier de la liste, Posi n'a jamais été modifié et vaut donc toujours Liste^.Fin. Si Posi vaut **nil**, c'est qu'on est dans le second cas, à savoir que le nouveau maillon est plus petit que tous ceux de la liste, ce qui force son insertion en début de liste. Le troisième et dernier cas utilise la valeur de Posi qu'on sait utilisable puisqu'elle ne vaut pas **nil**. On effectue donc l'insertion entre l'élément pointé par Posi et celui pointé par Posi^.Suiv. Les deux premières méthodes d'insertion (début et fin de liste) ont déjà été vues, intéressons-nous donc plutôt à la troisième. On commencer par fixer les champs Suiv et Pred du nouveau maillon. Comme on doit insèrer entre l'élément pointé par Posi et Posi^.Suiv, on affecte ces valeurs respectivement à Pred et à Suiv. Les deux autres instructions ont pour but de faire pointer le champ Suiv de l'élément pointé par Posi vers le nouveau maillon, et de même pour le champ Pred de l'élément pointé par Posi^.Suiv.
La suppression d'un élément fonctionne sur le même principe que pour l'implémentation avec un tableau : on donne le numéro d'un élément et si cet élément existe, il est retiré de la liste et détruit. La suppression est donc assez proche de l'insertion. Après quelques vérifications, une recherche est lancée sous la forme d'un parcours linéaire de la liste, afin de localiser l'élément. Si l'élément n'a pas été trouvé, la fonction se termine. Sinon, on doit d'une part retirer les références à cet élément dans la liste, ce qui se fait de 3 manières différentes suivant que l'élément est le premier, le dernier, ou un autre, et d'autre part supprimer l'élément en libérant la mémoire.

```
function LCTSupprIdx(Liste: TListeChaineeTriee; index: integer): TListeChaineeTriee;
var
  indx: integer;
  Posi: PMaillon;
begin
  result := Liste;
  if Liste = nil then exit;
  if index < 0 then exit;
  if LCTVide(Liste) then exit;
  // initialisation
  indx := 0;
  Posi := Liste^.Dbt;
  // parcours pour tenter d'obtenir l'élément
  while (Posi <> nil) and (indx < index) do
    begin
      inc(indx);
      Posi := Posi^.Suiv;
    end;
  // l'élément n°index existe pas, on quitte
  if Posi = nil then exit;
  { on supprime dans un premier temps les références à Posi
    dans la liste }
  if Posi = Liste^.Dbt then
    begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
      Liste^.Dbt := Posi^.Suiv;
      { suppression de la référence dans l'élément suivant :
        attention, celui-ci n'existe pas toujours }
      if Posi^.Suiv <> nil then
        Posi^.Suiv^.Pred := nil
      else
        Liste^.Fin := nil;
    end
  else if Posi = Liste^.Fin then
    begin
      Liste^.Fin := Posi^.Pred;
      { suppression de la référence dans l'élément précédent :
        attention, celui-ci n'existe pas toujours }
      if Posi^.Pred <> nil then
        Posi^.Pred^.Suiv := nil
      else
        Liste^.Dbt := nil;
    end
  else
    begin
      Posi^.Pred^.Suiv := Posi^.Suiv;
      Posi^.Suiv^.Pred := Posi^.Pred;
    end;
  // maintenant, on supprime le maillon
  Dispose(Posi^.Elem);
  Dispose(Posi);
end;
```

Quelques explications encore : dans le cas où l'élément supprimé est le premier ou le dernier, il faut non seulement mettre à jour respectivement le champ Dbt ou le champ Fin, mais faire attention à un cas particulier : si l'élément supprimé est le dernier (ce qui ne peut pas se produire si l'élément est au "milieu" de la liste), il faudra en plus fixer respectivement le champ Fin ou le champ Dbt de la liste à **nil**, afin de respecter notre convention pour la liste vide, à savoir Dbt = Fin = **nil**.
Il est maintenant temps de passer à un exercice. Vous allez programmer certaines des opérations restantes. Pour cela, vous pouvez vous aider de l'unité **lst_dbl_ch.pas** qui comporte tout ce que nous avons fait jusqu'à présent. Vous pourrez ainsi incorporer le code de cette unité dans un projet (Menu Projet, choix "Ajouter au projet...") et la compléter.

Exercice 1 : (voir la **solution et les commentaires**)
A l'aide de l'unité téléchargeable ci-dessus, écrivez le code source des deux opérations et de la procédure d'affichage décrits ci-dessous :

- LCTNbElem : fonction retournant le nombre d'éléments dans une liste.
- LCTDetruire : procédure libèrant toute la mémoire associée aux personnes, aux maillons et à la liste.
- AfficheListe : procédure affichant le contenu d'une liste dans une "Sortie". (voir les commentaires pour chaque fonction/procédure dans l'unité téléchargeable)

Une fois cet exercice réalisé, vous pouvez lire la correction proposée et passer à la suite où une unité complète intégrée dans un projet exemple est disponible.
En adaptant le projet utilisé pour tester les listes implémentées dans un tableau, on peut facilement tester les listes à double chaînage. Cette application a déjà été décrite plusieurs fois dans ce chapitre (elle marche presque toujours sur le même principe. Les modifications à apporter concernant les noms des opérations à utiliser, le type de la liste Lst, et consiste à remplacer l'unité implémentant une liste par un tableau par celle que vous venez de complèter si vous avez fait l'exercice précédent. Vous pouvez **télécharger le projet terminé**.
Voilà pour les listes triées à l'insertion. les deux méthodes présentées ici ont eu non seulement pour but de vous démontrer une fois de plus l'abstraction de la structure des données qui peut être réalisée dans une application (presque pas de changements dans l'application de tests alors que le type des données a complètement changé entre la représentation par un tableau et la représentation chaînée), mais également de vous familiariser avec ce concept trop souvent considéré comme difficile que sont les listes doublement chaînées. Je me suis justement attaché ici à donner des explications et des schémas permettant leur compréhension plutôt que d'essayer d'optimiser inutilement le code source qui les manipule.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Dans la section suivante, nous allons abandonner le tri à l'insertion pour le tri à la demande. C'est une approche différente qui permet notamment différents ordres de tri sur une même liste. C'est également l'approche adoptée pour les objets TList et TStringList fournis par Delphi.

## XV-D-5 - Implémentation permettant différents tris

### XV-D-5-a - Présentation

Lorsqu'on ne sait pas exactement quel ordre de tri sera nécessaire sur une liste, ou encore si on doit pouvoir choisir entre plusieurs méthodes de tri pour une seule liste, ou tout simplement si on ne souhaite pas du tout une liste triée, il est intéressant de séparer les deux concepts : la manipulation des entrées de la liste (ajout, insertion (ces deux notions étaient confondues pour les listes triées, elle ne vont plus l'être ici), déplacement (à noter que le déplacement était auparavant impossible car l'ordre de tri fixait la place des éléments), suppression) d'un coté, et le tri d'un autre coté. Ainsi, il sera possible de paramètrer un tri et de le lancer seulement lorsque ce sera nécessaire. Cette approche a cependant un inconvénient puisque le tri peut avoir à être relancé à chaque ajout/insertion d'un élément, ce qui peut ralentir les traitements.
Commençons comme d'habitude par formaliser un peu la notion de liste non triée :

- Création d'une nouvelle liste vide
  paramètres : (aucun)
  résultat : liste vide
- Destruction d'une liste
  paramètres : une liste
  résultat : (aucun)
- Insertion d'un élément
  paramètres : une liste, un élément du type stocké dans la liste, un index de position
  résultat : une liste contenant l'élément inséré à la position demandée ou à défaut en fin de liste
- Ajout d'un élément
  paramètres : une liste, un élément du type stocké dans la liste
  résultat : une liste contenant l'élément en fin de liste
- Suppression d'un élément
  paramètres : une liste, un élément du type stocké dans la liste
  résultat : une liste où l'élément transmis a été supprimé
  (Note : on se permettra souvent, dans la pratique de remplacer l'élément par son index)
- Nombre d'éléments
  paramètres : une liste
  résultat : un entier indiquant le nombre d'éléments dans la liste
- Accès à un élément
  paramètres : une liste, un entier
  résultat : sous réserve de validité de l'entier, l'élément dont le numéro est transmis
- Tri de la liste
  paramètres : une liste, un ordre
  résultat : une liste triée en fonction de l'ordre transmis

Comme d'habitude, nous ajouterons une procédure d'affichage du contenu d'une liste écrivant dans une Sortie de type TStrings. Quelques précisions sont à apporter sur les opérations ci-dessus. L'insertion consiste à placer un élément à la position spécifiée, en décalant si besoin les éléments suivants. Etant donné que les n éléments d'une liste seront numérotés de 0 à n-1, les valeurs autorisées iront de 0 à n, n signifiant l'ajout standard, c'est-à-dire en fin de liste. Bien qu'il soit normalement préférable de programmer une suppression ayant pour paramètre l'élément à supprimer, il est beaucoup plus simple et plus efficace, dans la plupart des cas, d'implémenter une suppression s'appliquant sur l'élément dont l'index est transmis.
Vous êtes peut-être intrigué(e) par le second paramètre donné à l'opération de tri. En effet, si vous êtes un peu dans les maths, l'ordre transmis doit constituer une relation d'ordre total sur l'ensemble des éléments stockable dans la liste. Si vous n'êtes pas dans les maths, sachez simplement que ce paramètre va nous permettre de décider pour deux éléments quelconques, lequel est le "plus petit" et lequel est le "plus grand". En pratique, le paramètre transmis à la fonction sera d'un type encore inconnu de vous, dont nous allons parler tout de suite.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XV-D-5-b - Paramètres fonctionnels

Nous allons faire un petit détour par ce qui s'appelle les paramètres fonctionnels. Par ce terme un peu trompeur de "paramètre fonctionnel", on désigne en fait un paramètre d'une fonction ou d'une procédure dont le type est lui-même une fonction (les paramètres procéduraux permettent la même chose avec les procédures). Il est ainsi possible de transmettre le nom d'une fonction existante comme paramètre d'une autre fonction/procédure. L'utilité de ce genre de procédé n'est pas à démontrer si vous avez pratiqué des langages tels que le C ou le C++. Sinon, sachez simplement que ce genre de paramètre sera d'une grande utilité pour spécifier l'ordre d'un tri. J'expliquerai pourquoi et comment en temps voulu. Mais revenons-en à nos moutons.
Un paramètre fonctionnel doit être typé par un type "fonctionnel" défini manuellement dans un bloc **type**. Voici un exemple qui va servir de base à la définition plus générale :

```
type
  TFuncParam = function (S: string): integer;
```

Comme vous pouvez le constater, le bloc ci-dessus définit un nouveau type fonctionnel appelé TFuncParam. La morphologie de la définition de ce type est nouvelle. Elle correspond en fait à la ligne de déclaration d'une fonction privée du nom de la fonction. Ainsi, lorsqu'on définit une fonction

```
function Longueur(S: string): integer;
```

son type est :

```
function (S: string): integer;
```

ce qui fait que notre fonction Longueur est de type "TFuncParam".
Une fois un type fonctionnel défini, il peut être employé comme type d'un ou de plusieurs paramètres d'une fonction/ procédure. Ainsi, voici le squelette d'une fonction recevant un paramètre fonctionnel et un autre de type chaîne :

```
function Traite(S: string; FonctionTraitement: TFuncParam): integer;
begin

end;
```

Un appel possible de cette fonction peut alors se faire en donnant pour "valeur" du paramètre le nom d'une fonction du type fonctionnel demandé. Ainsi, on pourrait appeler "Traite" de la manière suivante :

```
Traite('Bonjour !', Longueur);
```

car "Longueur" est bien du type fonctionnel défini pour le second paramètre. A l'intérieure d'une fonction/procédure acceptant un paramètre fonctionnel, le paramètre doit être considéré comme une fonction à part entière, dont la définition est donnée par son type (paramètres, résultat) que l'on peut appeler comme n'importe quelle fonction. Ainsi, voici la fonction "Traite" complétée pour faire appel à son paramètre fonctionnel :

```
function Traite(S: string; FonctionTraitement: TFuncParam): integer;
begin
  result := FonctionTraitement(S);
end;
```

Ainsi, si on appelle Traite comme dans l'exemple ci-dessus (avec 'Bonjour !' et Longueur), le résultat sera 9, car la fonction Longueur sera appelée avec la chaîne 'Bonjour !' en paramètre et le résultat sera retourné par "Traite".


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Les paramètres fonctionnels (et leurs alter-ego procéduraux) sont à ma connaissance assez rarement utilisés, malgré la souplesse de programmation qu'ils peuvent procurer. Nous nous en servirons d'ailleurs uniquement pour les tris.

## XV-D-5-c - Implémentation du tri à la demande

Concrètement, les paramètres fonctionnels vont nous permettre de généraliser l'emploi d'une fonction de comparaison telle que CompPersonnes. Au lieu d'appeler une procédure nommée explicitement dans le code, nous allons utiliser un paramètre fonctionnel que devra fournir le programmeur utilisant la liste. Ainsi, il sera libre de définir ses fonctions de comparaison permettant d'obtenir des tri particuliers, et d'appeler ensuite l'opération de tri en donnant en paramètre l'une de ces fonctions de comparaison.
Nous allons reprendre ici l'implémentation par double chaînage pour nos listes. Les éléments manipulés seront donc de type TPersonne. Voici la déclaration du type fonctionnel qui nous servira à spécifier une fonction de comparaison. On demande toujours deux paramètres du type manipulé dans la liste et le résultat est toujours un entier négatif, positif ou nul si le premier élément est respectivement plus petit, plus grand ou égal au second :

```
TCompareFunction = function(P1, P2: PPersonne): integer;
```

Toutes les opérations sont identiques, sauf l'ajout qui se scinde en deux opérations distinctes : l'ajout ou l'insertion, ainsi que l'opération de tri qui est à créer entièrement.
En ce qui concerne l'ajout, ce n'est pas très compliqué puisque c'est un ajout en fin de liste. On retrouve donc un code source identique à celui de LCTInsereFin. Voici :

```
function LPAjout(Liste: TListePersonnes; P: TPersonne): TListePersonnes;
var
  Temp: PMaillon;
begin
  Result := Liste;
  if Liste = nil then exit;
  // création d'un nouvel élément (maillon)
  New(Temp);
  New(Temp^.Elem);
  // initialisation du nouveau maillon
  Temp^.Elem^ := P;
  // 2 cas, si la liste est vide ou non
  if LPVide(Liste) then
    // initialise une liste à 1 élément (début et fin)
    begin
      Temp^.Pred := nil;
      Temp^.Suiv := nil;
      Liste^.Dbt := Temp;
      Liste^.Fin := Temp;
    end
  else
    // insertion après le dernier élément
    begin
      // l'ancien dernier élément doit être mis à jour
      Liste^.Fin^.Suiv := Temp;
      Temp^.Pred := Liste^.Fin;
      Temp^.Suiv := nil;
      Liste^.Fin := Temp;
      // Liste^.Dbt ne change pas
    end;
end;
```

Pour ce qui concerne l'insertion, c'est un peu plus compliqué, car il y a divers cas. La première étape consistera à vérifier la validité de la liste et de l'index transmis. Ensuite, un premier cas particulier concerne une liste vide. Quel que soit l'index donné, l'élément sera inséré en tant qu'élément unique. Un second cas particulier est alors le cas où l'index vaut 0 : dans ce cas on effectue une insertion en début de liste. Si aucun de ces deux cas ne se présente, l'élément ne sera pas inséré en début de liste, et donc un élément le précédera. Partant de cela, on recherche l'élément précédent en parocurant la liste jusqu'à arriver à l'élément en question ou à la fin de la liste. Si c'est ce dernier cas qui se produit (index trop grand) l'élément est inséré en fin de liste. Sinon il y a encore deux cas : soit l'élément trouvé est le dernier


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

de la liste et l'insertion se fait en fin de liste, soit ce n'est pas le dernier et l'insertion se fait entre l'élément trouvé et son "suivant" (champ Suiv). je ne reviendrai pas sur les détails techniques de chaque cas d'insertion (liste vide, en début, en fin, au milieu) car je l'ai déjà fait. Voici le code source :

```
function LPInsere(Liste: TListePersonnes; P: TPersonne; index: integer): TListePersonnes;
var
  indxPosi: integer;
  Temp, Posi: PMaillon;
begin
  Result := Liste;
  if Liste = nil then exit;
  if index < 0 then exit;
  // 1. Création du maillon
  New(Temp);
  New(Temp^.Elem);
  // initialisation du nouveau maillon par le paramètre P
  Temp^.Elem^ := P;
  // 2. Si la liste est vide, on insère l'élément seul
  if LPVide(Liste) then
    // initialise une liste à 1 élément (début et fin)
    begin
      Temp^.Pred := nil;
      Temp^.Suiv := nil;
      Liste^.Dbt := Temp;
      Liste^.Fin := Temp;
      exit;
    end;
  // 3. Cas particulier : si index = 0, insertion en tête
  if index = 0 then
    begin
      Liste^.Dbt^.Pred := Temp;
      Temp^.Suiv := Liste^.Dbt;
      Temp^.Pred := nil;
      Liste^.Dbt := Temp;
      exit;
    end;
  // 4. Recherche de l'élément "précédent", c'est-à-dire de l'élément
  // qui précédera l'élément inséré après insertion.
  indxPosi := 1;
  Posi := Liste^.Dbt;
  while (Posi <> nil) and (indxPosi < index) do
    begin
      inc(indxPosi);
      Posi := Posi^.Suiv;
    end;
  { 3 cas sont possibles :
    - Posi pointe sur l'élément "précédent", comme souhaité, ce qui
      donne deux sous cas :
        - Posi pointe sur le dernier élément de la liste, ce qui donne
          une insertion à la fin
        - Dans le cas contraire, c'est une insertion classique entre deux éléments.
    - Posi vaut nil, ce qui donne une insertion en fin de liste. }
  // cas 1 : Posi est non nil et est différent de Liste^.Fin
  if (Posi <> nil) and (Posi <> Liste^.Fin) then
    begin
      Temp^.Suiv := Posi^.Suiv;
      Temp^.Pred := Posi;
      Posi^.Suiv^.Pred := Temp;
      Posi^.Suiv := Temp;
    end
  else
    begin
      // insertion à la fin
      Liste^.Fin^.Suiv := Temp;
      Temp^.Pred := Liste^.Fin;
      Temp^.Suiv := nil;
      Liste^.Fin := Temp;
    end;
end;
```



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Venons-en à la procédure de tri. Nous allons avoir besoin d'un accès à un maillon par son numéro. Le mieux est de créer une opération "interne" (non accessible depuis l'extérieur) qui permettra ce genre d'accès. Voici son code source :

```
function LPGetMaillon(Liste: TListePersonnes; index: integer): PMaillon;
var
  indx: integer;
begin
  indx := 0;
  Result := Liste^.Dbt;
  while indx < index do
    begin
      Result := Result^.Suiv;
      inc(indx);
    end;
end;
```

Etant donné que cette opération est interne à l'unité, elle n'effectue aucun test. Elle se contente d'utiliser un compteur et un élément pour parcourir la liste depuis son début. Lorsque le compteur est arrivé à la valeur désirée, l'élément en cours est lui-même le maillon désiré, que l'on retourne. Quant à la procédure de tri, elle accepte deux paramètres : une liste et une fonction de type "TCompareFunction". L'algorithme retenu pour effectuer le tri est le "Shell Sort" décrit dans la section consacrée aux tris. la fonction de comparaison fournie est utilisae pour comparer deux éléments lorsque c'est nécessaire, et le reste est simplement adapté à la structure triée. Referrez-vous au code source téléchargeable dans la section sur les tris pour avoir une version plus simple du Shell Sort. Voici le code source de la fonction de tri :

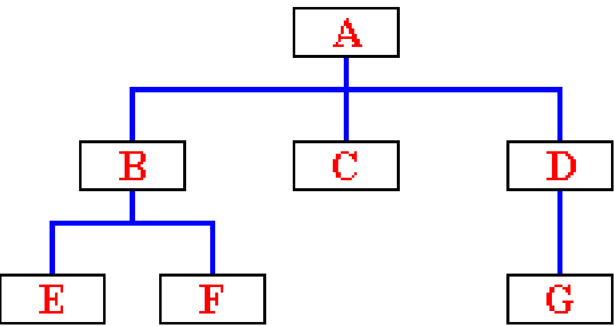
```
function LPTrieListe(Liste: TListePersonnes; Compare: TCompareFunction): TListePersonnes;
// l'algorithme retenu est celui du Shell Sort
// implémentation de ShellSort inspirée de "Le Language C - Norme ANSI" (ed. DUNOD)
var
  ecart, i, j, N: Integer;
  Mj, Mje: PMaillon;
  tmpPers: PPersonne;
begin
  result := Liste;
  N := LPNbElem(Liste);
  ecart := N div 2;
  // écart détermine l'écart entre les cases comparées entre elles
  while ecart > 0 do
    begin
      { parcours des éléments du tableau (tous seront
        parcourus puisque ecart s'annulera) }
      for i := ecart to N - 1 do
        begin
          j := i - ecart;
          // comparaisons et permutations d'éléments
          Mj := LPGetMaillon(Liste, j);
          Mje := LPGetMaillon(Liste, j + ecart);
          while (j >= 0) and (Compare (Mj^.Elem, Mje^.Elem) > 0) do
            begin
              // permutation des personnes et non des maillons
              tmpPers := Mj^.Elem;
              Mj^.Elem := Mje^.Elem;
              Mje^.Elem := tmpPers;
              // calcul de la nouvelle valeur de j
              j := j - ecart;
              { et obtention des nouveaux maillons }
              Mj := LPGetMaillon(Liste, j);
              Mje := LPGetMaillon(Liste, j + ecart);
            end;
        end;
      // écart diminie progressivement jusqu'à 0
      ecart := ecart div 2;
    end;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cette implémentation permet d'appliquer n'importe quel tri à une liste. Il suffit de créer une fonction de comparaison permettant d'obtenir le tri souhaité et le simple fait d'appeler l'opération de tri sur la liste effectue le tri dans l'ordre prévu. Ceci permet de se consacrer à la manière dont l'on veut trier et de ne pas s'embêter dans les détails d'implémentation de l'algorithme de tri.
Je suis conscient de vous avior noyé dans baeucoup trop de lignes de code. Je m'en excuse... Voici pour enfoncer le clou l'application permettant de tester les listes triées à la demande : **lsttriables.zip**.
Ceci termine la (trop longue) section consacrée aux listes. Les deux dernières sections de ce (trop long, également) chapitre sont consacrées aux structures non plus « linéaires » comme celles que nous venons d'étudier, mais arborescentes.

## XV-E - Arbres

Certaines données informatiques ne peuvent pas (facilement) être représentées par des structures linéaires comme celles présentées dans les sections précédentes de ce chapitre. Ces données doivent être représentées par une structure mettant en valeur leur type de relation, arborescent ou parfois "graphique". Les structures d'arbres permettent ainsi de représenter des structures en relation de type arborescent, comme des répertoires d'un disque dur par exemple. Le cas des relation de type "graphique" ne nous intéresse pas ici car il est plus complexe et s'appuie en informatique sur la théorie des graphes, sujet haut en couleur et beaucoup trop vaste et trop long pour être abordé ici.
Par arbre, on entend en informatique une structure où chaque élément peut avoir plusieurs sous-éléments, mais où chaque élément ne peut avoir qu'un sur-élément. Voici un exemple d'arbre :




Je ne parlerais pas ici de la signification d'un tel arbre, tout ce qui m'intéresse, c'est la représentation arborescente des éléments. Il serait certes possible de représenter cette structure dans une liste, mais ce serait un peu pénible à élaborer et à utiliser. Il serait plus intéressant d'élaborer des structures informatiques permettant une gestion de l'arborescence au niveau du code Pascal. Encore une fois, ceci se fait par l'utilisation intensive des pointeurs.
Avant de passer à la définition de structures Pascal, je vais me permettre de vous infliger quelques définitions un peu formelles. Un arbre est composé de noeuds et de branches. Une branche relie deux noeuds et chaque noeud possède une étiquette. Les informations à stocker dans l'arbre sont stockées exclusivement dans les étiquettes des noeuds. Une branche permet de relier deux noeuds. L'un de ces deux noeuds est appelé noeud parent et l'autre noeud enfant (appelé aussi fils). Un noeud parent peut avoir 0, un ou plus de noeuds enfants. Un noeud enfant a, sauf exception d'un seul noeud dans l'arbre, un et un seul noeud parent. Tout arbre non vide (contenant un noeud) possède un et un seul noeud ne possèdant pas de noeud parent. Ce noeud est nommé racine de l'arbre.
Un arbre peut contenir 0, 1 ou plus de noeuds. Les noeuds ne possédant pas de noeuds enfant sont appelés feuilles de l'arbre (que c'est original, tout ça !). A priori, pour un noeud qui n'est pas une feuille, il n'y a pas d'ordre dans l'ensemble de ses sous-noeuds, même si la représentation physique des données introduit souvent un ordre « artificiel ». Un sous-arbre engendré par un noeud est l'arbre dont la racine est ce noeud (tous les noeuds qui ne sont ni le noeud ni un de ses "descendants" sont exclus de ce sous-arbre). Par exemple, dans l'exemple ci-dessus, le sous-arbre engendré par A est l'arbre entier, tandis que le sous-arbre engendré par B est l'arbre dont B est la racine (les noeuds A, C, D et G sont exclus de cet arbre).
L'implémentation de types de données arborescents fait une utilisation intensive des pointeurs. Assurez-vous avant de continuer à lire ce qui suit que vous avez bien compris le concept de liste chaînée car il va être ici employé. Au niveau Pascal, on va créer un type TNoeud qui contiendra deux champs. Le premier sera l'étiquette du noeud, qui sera en pratique un pointeur vers une autre structure contenant les vraies données associées au noeud (un


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

pointeur de type PPersonne vers un élément de type TPersonne pour l'exemple). Le second champ sera une liste chaînée, simplement ou doublement, suivant les besoins (dans notre cas, elle le sera simplement, pour simplifier). Les éléments stockés par cette liste chaînée seront des pointeurs vers des noeuds fils du noeud en cours. Le type Arbre sera défini comme étant un pointeur vers un noeud qui sera le noeud racine. Lorsque l'arbre sera vide, ce pointeur sera **nil**. Voici quelques déclarations que je vais expliquer plus en détail ci-dessous :

```
type
  TFonction = (fcAucune, fcPDG, fcDirecteur, fcIngenieur, fcSecretaire);

  PElem = ^TPersonne;
  TPersonne = record
    Nom,
    Prenom: string;
    Age: integer;
    Fonction: TFonction;
  end;

  PNoeud = ^TNoeud;
  PListeNoeud = ^TListeNoeud;

  TNoeud = record
    Elem: PElem;
    Fils: PListeNoeud;
  end;

  TListeNoeud = record
    Noeud: PNoeud;
    Suiv: PListeNoeud;
  end;

  Arbre = PNoeud;
```

L'élément de type TPersonne servira à stocker les infos sur une personne. Chaque personne constituera l' « étiquette » d'un noeud. Le type PPersonne permettra d'utiliser des pointeurs sur ces éléments. Le type TNoeud est la structure centrale qui nous permettra de gèrer les arbres : il possède, comme je l'ai annoncé plus haut, deux champs. Le premier est un pointeur vers l'étiquette, de type TPersonne dans notre cas. Le second est un élément de type PListeNoeud qui est le début d'une liste simplement chaînée de maillons dont chacun pointe sur un autre noeud. Cette liste chaînée permettra de mémoriser les noeuds « enfant » du noeud en cours (chaque noeud possèdera un exemplaire propre de cette liste, qui sera simplement **nil** si le noeud n'a pas de fils (c'est une feuille).
Pour que vous voyez mieux à quoi ressemblera la structure de données utilisant ces diverses structures de base, j'ai transcrit le petit arbre présenté plus haut en un schémas utilisant les structures que nous venons de découvrir. Prenez le temps d'étudier et de comprendre ce schémas en repérant notamment le découpage logique en zones, où chaque « zone » sert à stocker un élément de l'arbre :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Comme vous pouvez le constater, la représentation de ce genre de structure de données est assez délicate, mais est d'une grande puissance car elle permet de représenter tout arbre de personnes, ce qui est précisément le but recherché.
Arrivé à ce point, je pourrais vous donner et expliquer les opérations destinées à manipuler un tel arbre. Ce serait, vous vous en doutez, passionnant et très long. Cependant, je vais me permettre de ne pas le faire, tout en vous promettant d'y revenir plus tard. En effet, lorsque vous aurez découvert la création d'objets dans un prochain chapitre,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

vous constaterez qu'il est nettement plus intéressant de construire un ensemble de classes manipulant un tel arbre général. Ceci nous permettra de plus de généraliser ce genre d'arbre en permettant à priori le stockage de n'importe quel élément dans l'arbre. Je consacrerai toute une partie du chapitre sur la création d'objets à implémenter cette structure d'arbre fort utile.

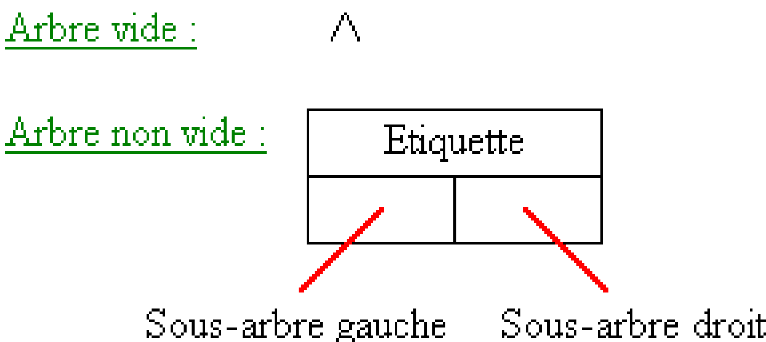
*Remarque : je sais que je m'expose à de vives critiques en faisant ce choix. Si vous ne partagez pas mon opinion, ou si vous voulez donner votre avis, **contactez-moi** et nous en discuterons : je suis ouvert à la discussion.*

En attendant, la dernière partie de ce chapitre va parler d'un cas particulier d'arbres : les arbres binaires.

## XV-F - Arbres binaires

*Note : Je ne donne ici qu'un bref aperçu des notions attachées aux arbres binaires. En effet, le sujet est extrêmement vaste et ce chapitre est avant tout destiné à vous aider à modèliser vos structures de données en Pascal et à les manipuler, et non pas à vous inculquer des notions qui n'ont pas leur place ici.*

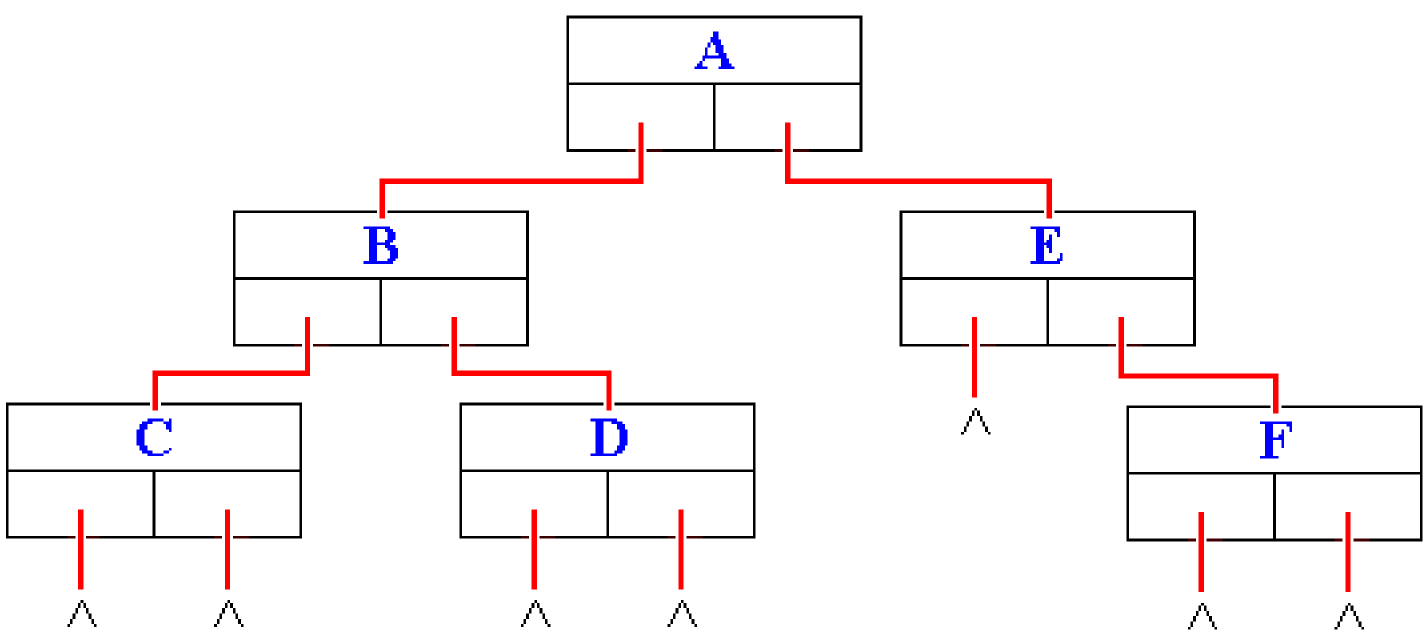
Les arbres binaires sont des cas particuliers d'arbres. Un arbre binaire est soit un arbre vide (aucun noeud), soit l'association d'un noeud et de deux arbres binaires nommés respectivement "sous-arbre gauche" et "sous-arbre droit". La racine de chacun de ces arbres, si elle existe (car chaque arbre, gauche ou droite, peut être vide) est nommée respectivement "fils gauche" et "fils droit". Voici les représentations formelles d'un arbre binaire vide ou non vide :



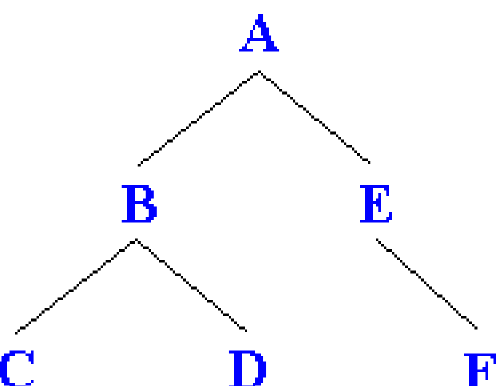

Habituellement, lorsqu'on fait une représentation d'un arbre binaire, on ne fait pas figurer les sous-arbres vides, (ce qui peut avoir pour conséquence de croire qu'un noeud n'a qu'un sous-noeud (voir-ci-dessous), ce qui est impossible, dans l'absolu). De plus, on évite la notation un peu trop formelle présentée ci-dessus, qui a cependant l'avantage d'être plus explicite. Ainsi, la représentation de l'arbre binaire suivant, quelque peu encombrante :



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




... sera avantageusement remplacée par la représentation suivante, nettement plus sympathique mais à prendre avec plus de précautions, car masquant tous les sous-arbres vides :




Au niveau informatique, un arbre binaire est bien plus maniable et simple à représenter qu'un arbre général : ayant seulement deux fils, il n'y a pas besoin d'utiliser une liste chaînée de fils pour chaque noeud. A la place, deux champs « Fils Gauche » et « FilsDroit » feront parfaitement l'affaire (leur représentation devra permettre de représenter le vide ou un arbre binaire, ce qui orientera notre choix vers les pointeurs, encore une fois). Contrairement aux structures linéaires telles que les piles, files et listes, il est utopique de vouloir garantir l'indépendance entre implémentation et utilisation de telles structures (seule une modélisation avec des objets permettrait de garantir ce genre d'indépendance, mais c'est une autre histoire !). Nous ne parlerons donc pas ici d'opérations mais seulement de procédures et de fonctions permettant le traitement d'arbres binaires. Rassurez-vous, dans la plupart des cas, ce sera amplement suffisant, mis à part pour les perfectionnistes qui trouveront de toute manière toujours quelque chose à reprocher à une modélisation.
Un arbre binaire sera donc représenté par un pointeur vers le noeud racine de l'arbre binaire (si l'arbre est non vide). Dans le cas d'un arbre vide, le pointeur sera **nil**). Chaque noeud sera constitué par un enregistrement comportant trois champs : le premier sera l'étiquette, qui en pratique pourra être un pointeur vers une autre structure ou plus simplement une variable ou un enregistrement. Les deux autres champs seront deux pointeurs vers le sous-arbre gauche et droite. Voici ce que cela donne en prenant une chaîne comme étiquette :

```
type
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  PNoeud = ^TNoeud;
  TNoeud = record
    Etiq: TEtiq;
    FG,
    FD: PNoeud;
  end;
  TEtiq = string;
  TArbin = PNoeud;
```

Le champ "Etiq" stockera l'étiquette du noeud, tandis que les champs "FG" et "FD" pointeront respectivement sur les sous-arbres gauche et droite (s'ils ne sont pas vides).
je ne parlerai ici que de la construction, de l'utilisation et de la destruction d'un arbre binaire. Les modifications de structure telles que les échanges de sous-arbres, les rotations (pour l'équilibrage d'arbres binaires) ne seront pas traitées ici, pour ne pas noyer la cible privilégiée de ce guide # les débutants # dans trop de détails. Il y a principalement deux méthodes pour construire un arbre binaire. Soit on commence par la racine, en créant ensuite chacun des deux sous-arbres, et ainsi de suite jusqu'aux feuilles, soit on commence par les feuilles, et on les rassemble progressivement par un « enracinement » (un sous-arbre gauche, un sous-arbre droit et une étiquette en entrée, un arbre binaire comprenant ces trois éléments en sortie) jusqu'à obtenir l'arbre binaire désiré. Cette construction ne pose en général aucun problème.
La première méthode se fera créant un noeud, et en modifiant les champs de l'enregistrement pointé pour créer les deux sous-arbres. La seconde méthode est en général plus élégante, quoique moins naturelle. Elle permet d'utiliser une fonction "Enraciner" qui permet de créer un arbre binaire à partir d'une étiquette et de deux arbres binaires. Pour créer une feuille, il suffira de fournir deux sous-arbres vides.
C'est cette dernière méthode que je vais décrire, car c'est celle qui vous servira le plus souvent, et entre autres pour le mini-projet proposé en fin de chapitre. Nous allons donc simplement écrire deux fonctions. La première sera chargée de donner un arbre vide, et la seconde effectuera l'enracinemetn de deux arbres dans une racine commune. Voici le code source, assez simple, de ces deux fonctions :

```
function ABVide: TArbin;
begin
  result := nil;
end;

function Enraciner(E: TEtiq; G, D: TArbin): TArbin;
begin
  new(result);
  result^.Etiq := E;
  result^.FG := G;
  result^.FD := D;
end;
```

La première fonction ne mérite aucune explication. La seconde alloue un espace mémoire pour stocker un noeud de l'arbre (la racine de l'arbre qui est créé par l'enracinement). L'étiquette et les deux sous-arbres sont ensuite référencés dans cet espace mémoire, ce qui crée un arbre dont la racine porte l'étiquette transmise et dont les deux sous-arbres gauche et droite sont ceux transmis.
Si nous voulons construire l'arbre binaire donné en exemple plus haut, il nous faudra deux variables temporaires recevant les branches. Soient X et Y ces deux variables. Le type de ces variables sera bien entendu TArbin. Voici le début de ce qu'il faudra faire :

```
X := Enraciner('C', ABVide, ABVide);
Y := Enraciner('D', ABVide, ABVide);
```

Ceci permet de créer les deux "feuilles" C et D (chacune de ces deux "feuilles" est en fait un arbre binaire dont les deux sous-arbres sont vides). Pour créer le branche de racine B et contenant les deux feuilles C et D, on utilise les deux éléments juste construits. Le résultat est affecté à X (on pourrait utiliser une troisième variable Z, mais ce serait inutile puisqu'on n'a plus besoin d'avoir accès à la feuille C. Voici l'instruction qui effectue cet enracinement :

```
X := Enraciner('B', X, Y);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

On ne peut pas tout de suite continuer à remonter dans l'arbre, car la partie droite n'existe pas encore. On commence donc par la créer. Pour cela, on ne doit pas toucher à la variable X qui conserve pour l'instant le sous-arbre gauche de l'arbre final. Voici les instructions qui construisent la partie droite :

```
Y := Enraciner('F', ABVide, ABVide);
Y := Enraciner('E', ABVide, Y);
```

Vous notez que dans la seconde instruction, le sous-arbre gauche étant vide, on fait à nouveau appel à ABVide. Voici enfin l'enracinement qui complète l'arbre. Le résultat est à nouveau stocké dans X :

```
X := Enraciner('A', X, Y);
```

Voilà pour ce qui est de la construction. Comme vous pouvez le constater, ce n'est pas très compliqué, à condition évidemment que les données se prêtent à ce genre de construction. Pour ce qui est de la destruction d'un arbre, la procédure qui réalise cela a quelque chose de particulier car elle peut être de deux choses l'une : très simple ou très compliquée ! Je m'explique : on peut programmer cette suppression de manière itérative, ou bien de manière récursive. La manière itérative utilise des boucles pour s'exécuter et parcourir tout l'arbre. Elle est plus rapide et consomme moins de mémoire. C'est plus compliqué qu'il n'y paraît car pour parcourir l'arbre, on doit mémoriser la branche en cours, pour aller supprimer son sous-arbre gauche (ce faisait, on perd la référence à la branche parent), puis son sous-arbre droit (mais comment va-t-on faire puisqu'on a perdu la branche parente ???). Bref, c'est tout-à-fait faisable, mais en utilisant une astuce : il faut utiliser une pile. C'est compliqué et il est plus intéressant de vous présenter la méthode récusrive (mais rien ne vous empêche de la programmer à titre d'exercice sur les piles et les arbres binaires).
je n'ai pas encore eu l'occasion de parler des procédures récursives dans ce guide, et c'est une excellente occasion. La particularité d'une procédure récursive (cela marche aussi avec les fonctions) est de s'appeler elle-même une ou plusieurs fois au cours de son exécution. Cette possibilité, intensivement employée dans d'autres langages tels LISP, l'est assez peu en Pascal. Lorsqu'on programme une fonction/procédure récursive, il faut prévoir ce qui s'appelle une condition d'arrêt, car sinon les appels imbriqués sont infinis et finissent par saturer la mémoire. C'est le même principe qu'avec une boucle **while** ou **repeat**. Dans notre cas, la procédure de suppression d'un arbre va s'appeler au plus deux fois pour supprimer chacun des deux sous-arbres (non vides). La condition d'arrêt sera trouvée lorsque les deux sous-arbres seront vides (on sera arrivé à une "feuille"). Le principe est le suivant : pour supprimer un arbre non vide, on supprime d'abord son sous-arbre gauche s'il n'est pas vide, puis de même avec son sous-arbre droit. Après ces deux appels, il ne reste plus qu'à supprimer le noeud racine en libérant la mémoire allouée. Voici ce que cela donne :

```
procedure DetrArbin(A: TArbin);
begin
  // destruction complète d'un arbre binaire
  if A <> nil then
    begin
      // destruction du sous-arbre gauche
      if A^.FG <> nil then
        DetrArbin(A^.FG);
      // destruction du sous-arbre droit
      if A^.FD <> nil then
        DetrArbin(A^.FD);
      // destruction du noeud racine
      Dispose(A);
    end;
end;
```

Voilà, c'est à peu près tout pour les arbres binaires. Je ne donne pas d'exemple concret de leur utilisation car le mini-projet qui suit va vous permettre de vous exercer un peu par vous-même. C'est un sujet qui pourraît être traité de manière beaucoup plus simple, mais le but est avant tout de vous faire manipuler les structures d'arbre binaire, de pile et de liste.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XV-G - Mini-projet : calculatrice

Le texte du mini-projet n'est pas encore terminé. Je le mettrai en ligne dés que possible et j'enverrai un message sur la **liste de diffusion** pour prévenir de sa disponibilité.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XVI - Programmation à l'aide d'objets

## XVI-A - Introduction

Ce chapitre est un prolongement du **chapitre 12** sur l'utilisation des objets. Ce chapitre vous donnait le rôle d'utilisateur des objets, en vous donnant quelques explications de base sur les mécanismes de l'utilisation des objets. Ce chapitre a pour ambition de vous donner toutes les clés indispensables pour bien comprendre et utiliser les objets tels qu'ils sont utilisables dans le langage de Delphi : Pascal Objet.
Certaines notions du chapitre 10 vont être ici reprises afin de les approfondir ou de les voir sous un jour différent. D'autres, complètement nouvelles pour vous, vont être introduites au fur et à mesure de votre progression dans cet immense chapitre. Afin de ne pas vous perdre dés le début, je vous propose un court descriptif de ce qui va suivre.
Dans un premier temps, nous allons aborder les concepts de base des objets. Ainsi, les termes de "programmation (orientée) objets", de classe, d'objet, de méthode, de champ vont être définis. Par la suite, nous aborderons les sections privées, protégées, publiques, publiées des classes, la notion de hiérarchie de classe, ainsi que des notions connexes comme l'héritage, le polymorphisme et l'encapsulation. Nous aborderons ensuite des notions plus spécifiques à Delphi : les propriétés, les composants et leur création. Enfin, nous aborderons les deux notions incontournables d'exception et d'interface.
Ne vous effrayez pas trop devant l'ampleur de la tâche, et bonne lecture !

## XVI-B - Concepts généraux

Cette partie a pour but de vous présenter les notions essentielles à la compréhension du sujet traité dans ce chapitre. Nous partirons des notions de base que vous connaissez pour aborder les concepts fondamentaux de la programmation objet. Nous aborderons ainsi les notions de classe, d'objet, de champ et de méthode. Cette partie reste très théorique : si vous connaissez déjà la programmation objet, vous pouvez passer à la suivante, sinon, lisez attentivement ce qui suit avant d'attaquer la partie du chapitre plus spécifiquement dédiée à Delphi.

### XVI-B-1 - De la programmation traditionnelle à la programmation objet

Le style de programmation que l'on appelle communément « programmation objet » est apparue récemment dans l'histoire de la programmation. C'est un style à la fois très particulier et très commode pour de nombreuses situations. Jusqu'à son invention, régnait, entre autres, la programmation impérative, c'est-à-dire un style faisant grande utilisation de procédures, de fonctions, de variables, de types de données plus ou moins évolués, et de pointeurs. Les langages C ou Pascal, qui figurent parmi les plus connus des non-initiés en sont de très bons exemples.
Le style de programmation impératif, très pratique pour de petits programme système ou de petites applications non graphiques existe toujours et a encore un avenir radieux devant lui, mais en ce qui concerne les grosses applications graphiques, ses limites ont rapidement été détectées, ce qui a forcé le développement d'un nouveau style de programmation, plus adapté au développement d'applications graphiques. L'objectif était de pouvoir réutiliser des groupes d'instructions pour créer de nouveaux éléments, et ce sans avoir à réécrire les instructions en question. Si cela ne vous paraît pas forcément très clair, abordons un petit exemple.
Lorsque vous utilisez une application graphique, vous utilisez une interface faite de fenêtres, de boutons, de menus, de zones d'édition, de cases à cocher. Chacun de ces éléments doit pouvoir être affiché et masqué à volonté pendant l'exécution d'une application les utilisant. En programmation impérative, il faudrait écrire autant de procédures d'affichage et de masquage que de types de composants utilisables, à savoir des dizaines. Leur code source présenterait d'évidentes similarités, ce qui serait assez irritant puisqu'il faudrait sans cesse réécrire les mêmes choses, d'où une perte de temps non négligeable. L'un des concepts de la programmation objets est de permettre d'écrire une seule fois une procédure d'affichage et une procédure de masquage, et de pouvoir les réutiliser à volonté dans de nouveaux éléments sans avoir à les réécrire (nous verrons comment plus tard).
Bien que délicate à aborder et à présenter au départ, la programmation objet réserve de très bonnes surprises, à vous de les découvrir dans les paragraphes qui suivent.

### XVI-B-2 - La programmation (orientée ?) objet

Pascal Objet est un langage dit orienté objet. Que signifie cette dénomination ?


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il existe en fait deux catégories de langages qui permettent la programmation utilisant les objets :

- Les langages ne permettant rien d'autre que l'utilisation d'objets, dans lesquels tout est objet. Ces langages sont assurément appréciés par certaines personnes mais imposent un apprentissage des plus délicats.
- Les langages permettant l'utilisation des objets en même temps que le style impératif classique. C'est le cas de Pascal Objet, et de nombreux autres langages comme Java et C++. Ces langages sont beaucoup plus souples même si nombre d'entre eux ne donnent pas les mêmes possibilités en termes de programmation objet.

Delphi supporte donc un langage orienté objet, ce qui signifie que quelqu'un qui ignore tout des objets peut presque entièrement s'en affranchir, ou les utiliser sans vraiment s'en rendre compte (ce qui a été votre cas jusqu'au chapitre 9 de ce guide). Limitée dans la version 1 de Delphi, la « couche » (l'ensemble des fonctionnalités) objet de la version 6 est réellement digne de ce nom. Mais trêve de blabla, passons aux choses sérieuses.

## XVI-B-3 - Classes

Le premier terme à comprendre lorsqu'on s'attaque à la programmation objet est incontestablement celui de "classe", qui regroupe une bonne partie de la philosophie objet. Ce paragraphe va aborder en douceur cette notion souvent mal comprise des débutants pour tenter de vous éviter ce genre de souci.
Une classe est le pendant informatique d'une notion, d'un concept. Si dans la vie courante vous vous dite : ceci est un concept, une notion bien précise possédant plusieurs aspects ou facettes, alors la chose en question, si vous devez la représenter en informatique, le sera sous forme de classe.
La classe est l'élément informatique qui vous permettra de transcrire au mieux un concept de la vie concrète dans un langage informatique. Une classe permet de définir les données relatives à une notion, ainsi que les actions qu'y s'y rapportent. Prenons tout de suite un exemple : celui de la télévision. Le concept général de télévision se rattache, comme toute notion, à des données (voire d'autres concepts) ainsi que des actions. Les données sont par exemple le poids, la longueur de la diagonale de l'écran, les chaînes mémorisées ainsi que l'état allumé/veille/éteint de la télévision. Les actions peuvent consister à changer de chaîne, à allumer, éteindre ou mettre en veille la télévision. D'autres actions peuvent consister en un réglage des chaînes. Le concept de télévision peut être raccordé à un concept appelé "magnétoscope", qui aura sa propre liste de données et d'actions propres. Voici une représentation possible du concept de télévision :



En informatique, la représentation idéale pour une télévision sera donc une classe. Il sera possible, au sein d'une seule et même structure informatique, de regrouper le pendant informatique des données # des variables # et le pendant informatique des actions, à savoir des procédures et des fonctions. Les variables à l'intérieur d'une classe seront appelées *champs*, et les procédures et les fonctions seront appelées *méthodes*. Les champs et les méthodes devront possèder des noms utilisables en temps normal pour des variables ou des procédures/fonctions. Voici un


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

schémas représentant non plus le concept de télévision, mais la *classe* « Television » ; notez les changements de couleurs, de noms et les mots importants tels que "classe", "champs" et "méthodes" :

**Classe "Television"**

champs :
- poids : entier
- longueur_diagonale : entier
- allume : vrai/faux
- chaines_memorisees

méthodes :
- eteindre
- allumer
- memoriser_chaine (chaine)

Tout comme un concept n'est pas une réalisation pratique, nous verrons qu'une classe ne l'est pas au niveau informatique : c'est une sorte, un moule. Le concept de télévision, qui ne vous permet pas d'en regarder une chez vous le soir, ni ne vous permet d'en régler les chaînes ne vous suffit pas à lui seul : il va vous falloir sa réalisation pratique, à savoir un exemplaire physique d'une télévision. Ceci sera également valable pour les classes, mais nous en reparlerons bientôt : ce seront les fameux *objets*.
Parlons maintenant des écrans d'ordinateur : ils sont également caractérisés par leur poids, leur diagonale, mais là où la télévision s'intéresse aux chaînes, l'écran d'ordinateur possède plutôt une liste de résolutions d'écran possibles, ainsi qu'une résolution actuelle. Un tel écran peut en outre être allumé, éteint ou en veille et doit pouvoir passer d'un de ces modes aux autres par des actions. En informatique, le concept d'écran d'ordinateur pourra être représenté par une classe définissant par exemple un réel destiné à enregistrer le poids, un entier destiné à mémoriser la diagonale en centimètres. Une méthode (donc une action transcrite au niveau informatique) permettra de l'allumer et une autre de l'éteindre, par exemple.
Si l'on s'en tient à ce qui vient d'être dit, et en adoptant la même représentation simple et classique que précédemment pour les concepts, nous voici avec ceci :

**Télévision**

données :
- poids
- longueur de la diagonale
- allumée ?
- chaînes mémorisées

actions :
- éteindre
- allumer
- mémoriser une chaîne

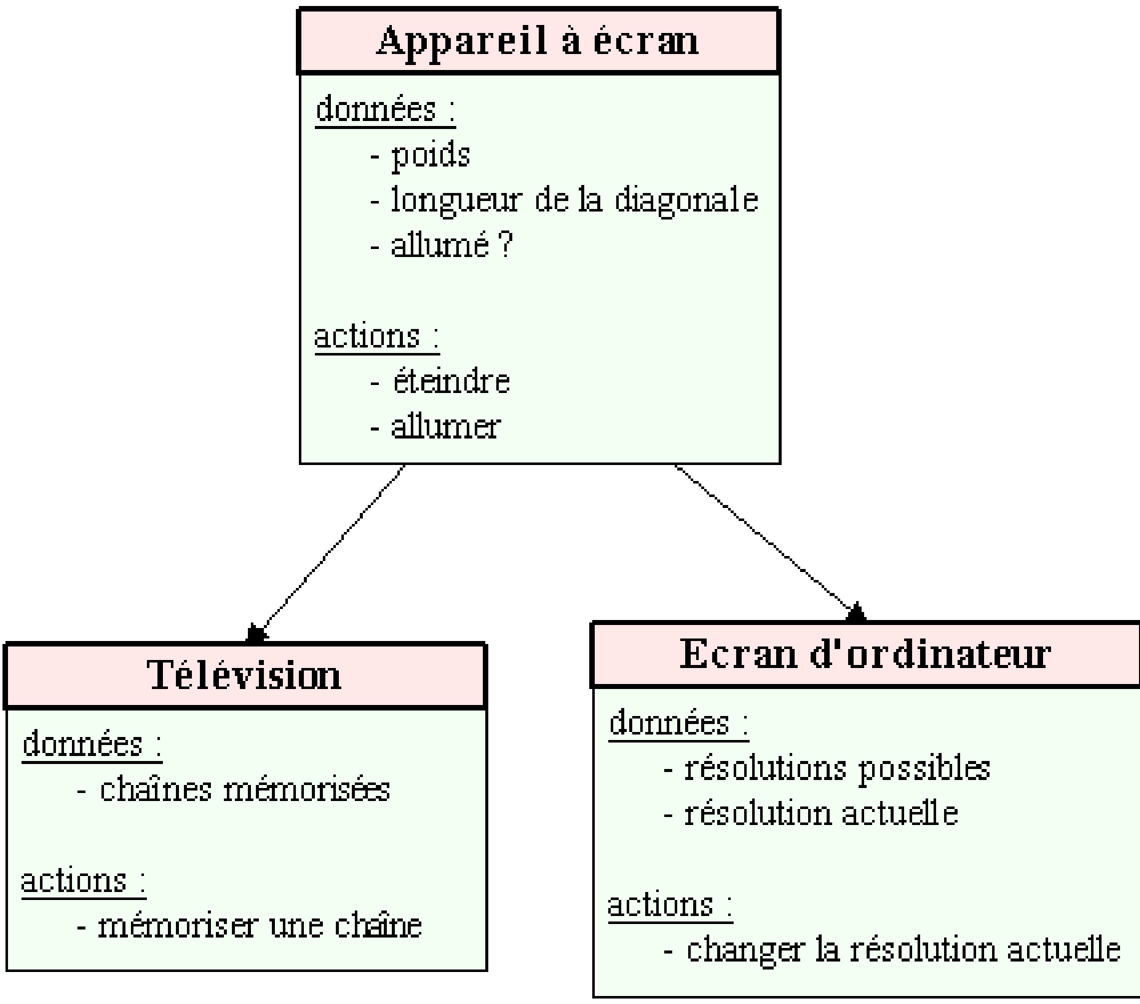
**Ecran d'ordinateur**

données :
- poids
- longueur de la diagonale
- allumé ?
- résolutions possibles
- résolution actuelle

actions :
- éteindre
- allumer
- changer la résolution actuelle


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Comme vous pouvez le constater, les deux diagrammes ci-dessus présentent des similarités flagrantes, comme la longueur de la diagonale, le poids et le fait d'éteindre ou d'allumer l'appareil. Ne serait-il pas intéressant de choisir une approche différente mettant en avant les caractéristiques communes, et de particulariser ensuite ?
Explications : Une télévision et un écran d'ordinateur sont tous deux des appareils à écran. Un appareil à écran possède un poids, une longueur de diagonale et il est possible de l'éteindre et de l'allumer. Une télévision est un tel appareil, possédant également une liste de chaînes mémorisées, une action permettant de mémoriser une chaîne, ainsi que d'autres non précisées ici qui font d'un appareil à écran... une télévision. Un écran d'ordinateur est également un appareil à écran, mais possédant une résolution actuelle, une liste de résolutions possibles, et une action permettant de changer la résolution actuelle. La télévision d'un coté, et l'écran d'ordinateur de l'autre sont deux concepts basés sur celui de l'appareil à écran, et lui ajoutant divers éléments qui le rendent particulier. Si je vous dis que j'ai un appareil à écran devant moi, vous pouvez légitimement penser qu'il peut s'agire d'une télévision, d'un écran d'ordinateur, voire d'autre chose...
Cette manière de voir les choses est l'un des fondements de la programmation objet qui permet (et se base) sur ce genre de regroupements permettant une réutilisation d'éléments déjà programmés dans un cadre général. Voici le diagramme précédent, modifié pour faire apparaître ce que l'on obtient par regroupement des informations et actions redondantes des deux concepts (un nouveau concept intermédiaire, appelé "appareil à écran" a été introduit à cette fin) :




L'intérêt de ce genre de représentation est qu'on a regroupé dans un seul concept plus général ce qui était commun à deux concepts proches l'un de l'autre. On a ainsi un concept de base très général, et deux concepts un peu plus particuliers s'appuyant sur ce concept plus général mais partant chacun d'un coté en le spécialisant. Lorsqu'il s'agira d'écrire du code source à partir des concepts, il ne faudra écrire les éléments communs qu'une seule fois au lieu de deux (ou plus), ce qui facilitera leur mise à jour : Il n'y aura plus de risque de modifier par exemple le code source


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

réalisant l'allumage d'une télévision sans modifier également celui d'un écran d'ordinateur puisque les deux codes source seront en fait un seul et même morceau de code général applicable à n'importe quel appareil à écran.
Au niveau de la programmation objet, nous avons déjà vu qu'un concept peut être représenté par une classe et avons déjà observé la représentation d'une classe. Le fait qu'un concept s'appuie sur un autre plus général et y ajoute des données et des actions fera de ce concept une seconde classe, dont on dira qu'elle *étend* la première classe. Il est maintenant temps de s'attaquer à la traduction des concepts en classes. Voici la représentation du diagramme précédent en terme de classes, sans encore aborder le code source ; notez que les regroupements sont identiques à ce qui est fait au niveau conceptuel, et donc que les deux diagrammes sont très proches l'un de l'autre :

Classe "AppareilAEcran"
champs :
- poids : entier
- longueur_diagonale : entier
- allume : vrai/faux
méthodes :
- eteindre
- allumer

Classe "Television"
champs :
- chaines_memorisees
méthodes :
- memoriser_chaine (chaine)

Classe "EcranOrdinateur"
champs :
- resolutions_possibles
- resolution_actuelle
méthodes :
- changer_resolution (nouvelle_resolution)

La programmation objet accorde une grande importance à ce genre de schémas : lorsqu'il sera question d'écrire un logiciel utilisant les capacités de la programmation objet, il faudra d'abord *modéliser* les concepts à manipuler sous forme de diagramme de concepts, puis traduire les diagrammes de concepts en diagrammes de classes. Lorsque vous aurez plus d'expérience, vous pourrez directement produire les diagrammes de classe, seuls utilisables en informatique.
Résumons ce qu'il est nécessaire d'avoir retenu de ce paragraphe :

- Une classe est la représentation informatique d'un concept
- Un concept possède des données et des actions.
- Une classe possède la représentation informatique des données # des champs # et la représentation informatique des actions : des méthodes.
- Tout comme un concept peut s'appuyer sur un autre concept et le particulariser, une classe peut *étendre* une autre classe en ajoutant des champs (données) et des méthodes (actions).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Voici un tableau vous montrant les 3 facettes d'un même élément dans les 3 mondes que vous connaissez désormais : le vôtre, celui de la programmation objet, et celui de la programmation impérative (sans objets). Vous pouvez y voir la plus forte lacune de la programmation impérative, à savoir qu'un concept ne peut pas y être modélisé directement.

| Monde réel | Programmation objet | Programmation impérative |
|---|---|---|
| Concept | Classe | *(rien)* |
| Donnée | Champ | Variable |
| Action | Méthode | Procédure<br>Fonction |

## XVI-B-4 - Objets

Nous avons vu précédemment que les classes étaient le pendant informatique des concepts. Les objets sont pour leur part le pendant informatique de la représentation réelle des concepts, à savoir des occurrences bien réelles de ce qui ne serait sans cela que des notions sans application possible. Une classe n'étant que la modélisation informatique d'un concept, elle ne se suffit pas à elle-même et ne permet pas de manipuler les occurrences de ce concept. Pour cela, on utilise les objets.
Voici un petit exemple : le concept de télévision est quelque chose que vous connaissez, mais comment ? Au travers de ses occurrences physiques, bien entendu. Il ne vous suffit pas de savoir qu'une télévision est un appareil électronique à écran, ayant un poids et une largeur de diagonale, qu'on peut éteindre, allumer, et dont on peut régler les chaînes : le soir en arrivant chez vous, vous allumez UNE télévision, c'est-à-dire un modèle précis, possédant un poids défini, une taille de diagonale précise. Si vous réglez les chaînes, vous ne les réglez que pour VOTRE télévision et non pour le concept de télévision lui-même !
En informatique, ce sera la même chose. Pour manipuler une occurrence du concept modélisé par une classe, vous utiliserez un *objet*. Un objet est dit d'*une classe* particulière, ou *instance* d'une classe donnée. En programmation objet, un objet ressemble à une variable et on pourrait dire en abusant un petit peu que son type serait une classe définie auparavant. Si cela peut vous aider, vous pouvez garder en tête pour l'instant que les objets sont des formes évoluées de variables, et que les classes sont des formes évoluées de types ; je dis "pour l'instant" car il faudra petit à petit abandonner cette analogie pleine de pièges.
Un objet est en réalité un élément variable possédant un exemplaire personnel de chaque champ défini par sa classe. Ainsi, si on crée deux objets "Tele1" et "Tele2" de classe "Television", chacun aura un champ "poids" et un champ "longueur_diagonale". Il est impossible de lire ou d'écrire la valeur d'un champ d'une classe, mais il devient possible de lire ou d'écrire la valeur d'un champ d'un objet. Le fait de modifier la valeur d'un champ pour un objet ne modifie pas la valeur du même champ d'un autre objet de même classe. Ainsi, le fait de dire que la valeur du champ "poids" de l'objet Tele1 est 23 ne modifie pas la valeur du champ "poids" de l'objet "Tele2", qui peut valoir par exemple 16.
Les méthodes sont partagées par tous les objets d'une même classe, mais une méthode peut être appliquée non pas à la classe qui la définit, mais à un objet de cette classe. Les champs manipulés par la méthode sont ceux de l'objet qui a reçu l'appel de méthode. Ainsi, le fait d'appeler la méthode "eteindre" de l'objet "Tele2" fixe la valeur du champ "allume" de cet objet ("Tele2") à faux, mais n'a pas accès à l'objet "Tele1" à laquelle elle ne s'applique pas. Pour éteindre "Tele1", il faudra appeler la méthode "eteindre" de cet objet ("Tele1"). Ceci permet d'avoir pour chaque objet une copie des champs avec des valeurs personnalisées, et des méthodes s'appliquant à ces champs uniquement. C'est pour cela qu'on introduit souvent la programmation objet en présentant le besoin de regrouper données et instructions dans une même structure, à savoir l'objet.
Il est possible de manipuler un nombre quelconque d'objets d'une classe particulière (c'est-à-dire que vous pouvez par exemple avoir un nombre quelconque de télévisions, mais le concept de télévision reste unique), mais un objet est toujours d'une seule et unique classe (vous ne pouvez pas faire d'une télévision un casse-noisettes, encore que, comme dans la vie réelle, ce genre de chose soit en fait possible (comprenez toléré) mais d'un intérêt fort discutable !). En fait, nous verrons qu'un objet peut être *restreint à une classe*. Par exemple, une télévision pourra être temporairement considérée comme un appareil à écran, mais cela ne change en rien sa nature de télévision.

## XVI-B-5 - Fonctionnement par envoi de messages

Maintenant que vous connaissez les objets, il vous reste à comprendre comment ils vont fonctionner ensembles. En effet, un objet possède des champs et des méthodes, mais reste quelque chose d'inanimé. Pour faire fonctionner un objet, il faut lui envoyer un message. L'envoi d'un message consiste tout simplement à effectuer un appel de méthode sur cet objet. L'objet qui reçoit un tel message exécute la méthode correspondante sur ses champs et


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

retourne éventuellement un résultat. Le fait de modifier directement la valeur d'un champ d'un objet (sans appeler de méthode) ne constitue pas un envoi de message.
Il est intéressant de mettre en lien la notion de message à celle d'événement car ces deux notions sont dépendantes l'une de l'autre. Lorsque dans votre application un événement se produit, par exemple lorsqu'on clique sur un bouton, un message est envoyé à la fiche qui contient ce bouton. La fiche est en fait un objet d'une classe particulière, et le traitement du message consiste à exécuter une méthode de l'objet représentant la fiche, et donc à répondre à l'événement par des actions. Vous voyez au passage que vous utilisez depuis longtemps les envois de messages, les objets et les classes sans toutefois les connaître sous ce nom ni d'une manière profonde.
Dans les langages purement objet, seuls les envois de messages permettent de faire avancer l'exécution d'une application. Par exemple, pour effectuer 1+2, vous envoyez le message "addition" à l'objet 1 avec le paramètre 2, le résultat est le nouevl objet 3. Pascal Objet est heureusement pour vous un langage *orienté* objet, et à ce titre vous pouvez utiliser autre chose que des objets et donc ne pas trop vous attarder sur les messages.
Si je vous ai présenté les messages, c'est à titre d'information uniquement. Je suis persuadé que nombre de programmeurs n'ont jamais entendu parlé de l'envoi de messages en programmation objet, ce qui fait que vous pouvez vous en tenir à la notion d'appel de méthodes en sachant toutefois que cela correspond à l'envoi d'un message.

## XVI-B-6 - Constructeur et Destructeur

Les objets étant des structures complexes, on a recours, dans la plupart des langages, à ce que l'on appelle un *constructeur* et un *destructeur*. Ces deux éléments, qui portent pour une fois très bien leur nom, permettent respectivement de créer et de supprimer un objet.

- Le constructeur est utilisé pour initialiser un objet : non seulement l'objet en lui-même, mais également les valeurs des champs. En Pascal Objet, lorsqu'on construit un objet, ses champs sont toujours initialisés à la valeur 0, false ou nil selon les cas. Le constructeur est une méthode très spéciale souvent nommée create et appelée lors de la création d'un objet et qui permet donc de fixer des valeurs spécifiques ou d'appeler des méthodes pour réaliser cette tâche d'initialisation.
- Le destructeur est utilisé pour détruire un objet déjà créé à l'aide du constructeur. L'objet est entièrement libéré de la mémoire, un peu à la manière d'un élément pointé par un pointeur lorsqu'on applique un **dispose** sur le pointeur. Le destructeur est souvent appelé destroy.

Le cycle de vie d'un objet est le suivant : on le crée à l'aide du constructeur, on l'utilise aussi longtemps qu'on le veut, et on le détruit ensuite à l'aide du destructeur. Notez qu'une classe n'a pas de cycle de vie puisque ce n'est qu'une définition de structure, au même titre qu'une définition de type présente dans un bloc **type**.

## XVI-C - Bases de la programmation objet sous Delphi

Cette partie a pour but de vous faire découvrir les éléments de base de la programmation objet sous Delphi, à savoir la déclaration de classes, de champs, de méthodes, et d'objets. L'utilisation de ces éléments ainsi que du couple constructeur/destructeur est également au programme.

## XVI-C-1 - Préliminaire : les différentes versions de Delphi

Lorsqu'on parle de programmation objets, on touche un domaine de prédilection du langage Pascal Objet. Ce langage, intégré à Delphi depuis la version 1 et descendant de Turbo Pascal, intègre, au fil des versions, de plus en plus de nouveautés au fil de l'évolution des techniques de programmation objet. Ainsi, les interfaces sont apparues récemment, dans la version 2 ou 3 (ma mémoire me joue un tour à ce niveau, désolé). Autre exemple : la directive overload n'est apparue que dans Delphi 5 alors que la notion sous-jacente existe dans d'autres langages depuis longtemps.
Le but de cette petite introduction n'est pas de jeter la pierre à Pascal Objet qui s'en tire bien dans le monde de la programmation objet, mais plutôt de vous mettre en garde : selon la version de Delphi dont vous disposez, il se peut que certaines des notions les plus pointues vues dans la suite de ce chapitre n'y soient pas intégrées. Ce chapitre est actuellement conforme au langage Pascal Objet présent dans la version 6 de Delphi, que je vous suggère d'acquérir si vous souhaitez bénéficier des évolutions les plus récentes du langage.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XVI-C-2 - Définition de classes

La partie que vous venez peut-être de lire étant assez théorique, vous vous sentez peut-être un peu déconcerté. Il est maintenant temps de passer à la pratique, histoire de tordre le cou à vos interrogations. Attaquons donc l'écriture d'un peu de code Pascal Objet.
Une classe, en tant que type de donnée évolué, se déclare dans un bloc **type**. Il n'est pas conseillé, même si c'est possible, de déclarer une classe dans un bloc **type** local à une procédure ou fonction ; les deux endroits idéaux pour les déclarations de classes sont donc dans un bloc **type** dans l'interface ou au début de l'implémentation d'une unité.
La déclaration d'une classe prend la forme suivante :

```
NomDeClasse = class [(ClasseEtendue)]
  déclarations_de_champs_ou_de_methodes
end;
```

Dans la déclaration ci-dessus, le nom de la classe doit être un identificateur, le mot-clé **class** dénote une déclaration de classe. Il peut être suivi du nom d'une autre classe entre parenthèses : dans ce cas la classe que vous déclarez étend la classe en question et donc se base sur tout ce que contient la classe étendue (nous y reviendrons plus tard). Ensuite, vous pouvez déclarer des champs et des méthodes, puis la déclaration de classe se termine par un **end** suivi d'un point-virgule.
Partons de l'exemple pris dans la partie précédente, à savoir celui d'une télévision. Voici la déclaration de base de la classe "Television" :

```
 Television = class
  end;
```

A l'intérieur d'une déclaration de classe, il est possible de déclarer des champs et des méthodes. Je profite de l'occasion pour mentionner un terme que vous rencontrerez certainement dans l'aide de Delphi : celui de *membre*. On appelle *membre* d'une classe un élément déclaré dans la classe, donc un champ ou une méthode (nous verrons plus loin que cela comprend aussi les éléments appelés *propriétés*). Un champ se déclare comme une variable dans une section **var**. Voici la déclaration de la classe Television, complétée avec les champs présentés sur le premier diagramme (celui ne faisant pas encore apparaître l'appareil à écran) :

```
  Television = class
    poids: integer;
    longueur_diagonale: integer;
    allume: boolean;
    chaines_memorisees: array[1..MAX_CHAINES] of integer;
  end;
```

Comme vous pouvez le constater, dans l'état actuel, le morceau de code ci-dessus ressemble furieusement à la déclaration d'un enregistrement (à tel point qu'il suffirait de substituer le mot **record** au mot **class** pour passer à un enregistrement). Nous allons maintenant déclarer les méthodes de la classe Television. Une méthode se déclare comme pour une déclaration dans l'interface d'une unité, à savoir la première ligne donnant le mot **procedure** ou **fonction** (nous parlons toujours de méthodes, même si Pascal Objet, lui, parle encore de procédures ou de fonctions), le nom de la méthode, ses paramètres et son résultat optionnel. Voici la déclaration désormais complète de la classe Television (il est à noter que jusqu'ici vous deviez pouvoir compiler l'application dans laquelle vous aviez inséré le code source, ce qui ne va plus être le cas pendant un moment. Si vous avez des soucis de compilation, consultez le code source complet **ici**) :

```
 Television = class
    poids: integer;
    longueur_diagonale: integer;
    allume: boolean;
    chaines_memorisees: array[1..MAX_CHAINES] of integer;
    procedure allumer;
    procedure eteindre;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    procedure memoriser_chaine (numero, valeur: integer);
  end;
```

Pour l'instant, ceci termine la déclaration de la classe. Si vous êtes très attentif, vous devez vous dire : « mais quid du constructeur et du destructeur ? ». La réponse est que nous n'avons pas à nous en soucier, mais vous le justifier va me prendre un plus de temps. En fait, lorsque vous déclarez une classe en n'indiquant pas de classe étendue, Pascal Objet prend *par défaut* la classe "TObject", qui comporte déjà un constructeur et un destructeur. Comme la classe Television étend la classe TObject, elle récupère tout ce que possédait "TObject" (c'est à peine vrai, mais nous fermerons les yeux là-dessus pour l'instant, si vous le voulez bien) et récupère donc entre autres le constructeur et le destructeur. Il se trouve que ces deux méthodes particulières peuvent être utilisées pour la classe Television, grâce à un mécanisme appelé *héritage* que nous détaillerons plus tard, ce qui fait que nous n'écrirons pas de constructeur ni de destructeur pour l'instant.
Si vous tentez de compiler le code source comprenant cette déclaration de classe, vous devriez recevoir 3 erreurs dont l'intitulé ressemble à « Déclaration forward ou external non satisfaite : Television.????? ». Ces messages signalent tout simplement que vous avez déclaré 3 méthodes, mais que vous n'avez pas écrit le code source des méthodes en question. Nous allons commencer par écrire le code source de la méthode "allumer" de la classe "Television".
Une méthode se déclare dans la section **implementation** de l'unité dans laquelle vous avez déclaré la classe. La syntaxe du code source de la méthode doit être :

```
procedure|function NomDeClasse.NomDeMethode [(liste_de_parametres)] [: type_de_resultat];
[declarations]
begin
  [instructions]
end;
```

Ne vous fiez pas trop à l'aspect impressionnant de ce qui précède. Le seul fait marquant est que le nom de la méthode doit être préfixé par le nom de sa classe et d'un point. Voici le squelette de la méthode "allumer" de la classe "Television" :

```
procedure Television.allumer;
begin

end;
```

Lorsque vous écrivez une méthode d'une classe, vous devez garder en tête que le code que vous écrirez, lorsqu'il sera exécuté, concernera un objet de la classe que vous développez. Ceci signifie que vous avez le droit de modifier les valeurs des champs et d'appeler d'autres méthodes (nous allons voir dans un instant comment faire), mais ce seront les champs et méthodes de l'objet pour lequel la méthode a été appelée. Ainsi, si vous modifiez le champ "allume" depuis la méthode "allumer" qui a été appelée pour l'objet "Tele1", le champ "allume" de l'objet "Tele2" ne sera pas modifié.
Pour modifier la valeur d'un champ depuis une méthode, affectez-lui simplement une valeur comme vous le feriez pour une variable. Voici le code source complet de la méthode "allumer" :

```
procedure Television.allumer;
begin
  allume := true;
end;
```

Comme vous pouvez le constater, c'est relativement simple. Pour vous entraîner, écrivez vous-même le code source de la méthode "éteindre". Nous allons écrire ensemble le code source de la troisième et dernière méthode, "memoriser_chaine". Cette méthode prend deux paramètres, "numero" qui donne le numéro de chaîne et "valeur" qui donne la valeur de réglage de la chaîne (nous nous en tiendrons là pour l'émulation d'un téléviseur !). La partie squelette est toujours écrite suivant le même modèle. Le code source est très simple à comprendre, mis à part le fait qu'il agit sur un champ de type tableau, ce qui ne change pas grand-chose, comme vous allez pouvoir le constater :

```
procedure Television.memoriser_chaine(numero, valeur: integer);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
begin
  if (numero >= 1) and (numero <= MAX_CHAINES) then
    chaines_memorisees[numero] := valeur;
end;
```

Une fois le code source de ces trois méthodes entré dans l'unité, vous devez pouvoir la compiler. Si ce n'est pas le cas, récupérez le code source **ici**. Vous savez maintenant ce qu'il est indispensable de connaître sur l'écriture des classes, à savoir comment les déclarer, y inclure des champs et des déclarations de méthodes, ainsi que comment implémenter ces méthodes. La suite de ce chapitre abordera diverses améliorations et compléments concernant l'écriture des classes, mais la base est vue. Nous allons maintenant passer à l'utilisation des objets, au travers de petits exemples.

## XVI-C-3 - Déclaration et utilisation d'objets

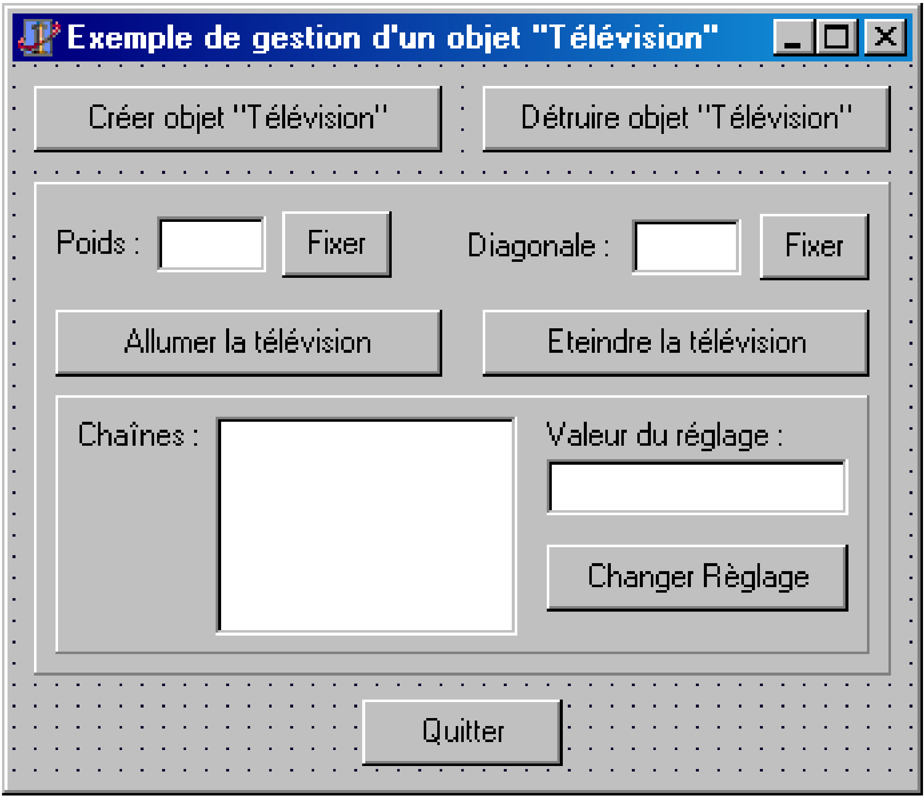
Vous avez vu dans la partie précédente (ou alors vous le saviez déjà) qu'une classe ne s'utilise pas directement : on passe par des objets instances de cette classe. L'utilisation des objets ayant déjà été abordée au chapitre 10, je vais être rapide sur les notions de construction et de destruction ainsi que sur le moyen d'appeler une méthode ou de modifier un champ. Ce paragraphe va surtout consister en l'étude de quelques exemples ciblés qui vous permettront de vous remettre ces opérations en mémoire.
Pour créer un objet, on doit faire appel au constructeur sous une forme particulière :
*Objet := Classe.create [(parametres)];*
Pour appeler une méthode, on utilise la syntaxe :
*Objet.Methode [(paramètres)];*
On peut faire référence à un champ par la construction *Objet.Champ* . Enfin, la destruction consiste à appeler le destructeur, appelé communément destroy, comme on le fait pour une méthode. Un objet se déclare dans une section **var**, avec comme nom de type la classe de cet objet. Voici une petite démo qui effectue quelques petits traitements sur un objet et affiche quelques résultats :

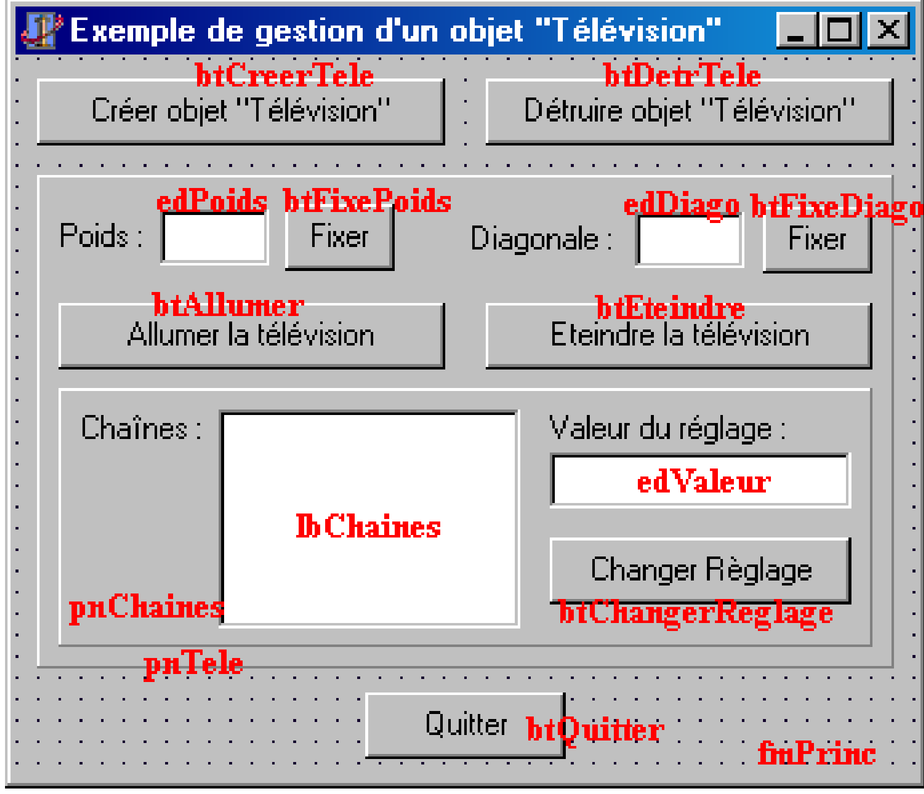
```
procedure TForm1.Button1Click(Sender: TObject);
var
  Tele1: Television;
begin
  // construction de l'objet Tele1
  Tele1 := Television.Create;
  // modification de champs
  Tele1.poids := 35;
  Tele1.longueur_diagonale := 70;
  // appel d'une méthode
  Tele1.allumer;
  // affichage d'infos
  if Tele1.allume then
    ShowMessage('Télévision allumée')
  else
    ShowMessage('Télévision éteinte');
  // encore un petit appel de méthode
  Tele1.eteindre;
  // et réaffichage d'infos.
  if Tele1.allume then
    ShowMessage('Télévision allumée')
  else
    ShowMessage('Télévision éteinte');
  // essai de mémorisation d'une chaîne
  Tele1.memoriser_chaine(1, 2867);
  // affichage du résultat
  ShowMessage('La chaîne n°1 est réglée sur la valeur '+IntToStr(Tele1.chaines_memorisees[1]));
  // destruction de Tele1
  Tele1.Destroy;
end;
```

Nous allons maintenant réaliser une petite application qui va nous permettre de manipuler à souhait un objet de classe "Television". Vous aurez ainsi l'occasion de voir comment on peut maîtriser le cycle de vie d'un objet, à savoir construction-utilisation-destruction. Nous aurons également l'occasion d'ajouter des champs et des méthodes


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

à la classe définissant la fiche principale, ce qui vous fera manipuler beaucoup de notions vues jusqu'à présent. Commencez par créer l'interface de la fiche principale d'une nouvelle application : téléchargez **le projet avec juste l'interface**, ou bien créez-là vous-même à l'aide des deux captures ci-dessus dont l'une donne les noms des composants. Le principe de l'interface est simple : on peut créer ou détruire une télévision, et l'éditer lorsqu'elle est créée. Lorsque la télévision est en plus allumée, on peut obtenir la liste des chaînes et les mémoriser.





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La première chose à faire est d'intégrer la déclaration de la classe "Television" et le code source des 3 méthodes de cette classe. Assurez-vous d'inclure la déclaration de la classe "Television" AVANT celle de la classe "TfmPrinc" (eh bien oui, c'est une déclaration de classe, non ?). Utilisez les fragments de code source donnés dans la section précédente pour cela. Déclarez la constante MAX_CHAINES avec la valeur 10. Fixez les propriétés "Visible" des panels "pnTele" et "pnChaines" à false, ainsi que la propriété "Enabled" des deux boutons "btDetrTele" et "btEteindre" à false. Faites fonctionner le bouton "Quitter" (en mettant un simple Close; dans la procédure de réponse au clic).
Déclarez ensuite un objet "Tele1" de classe "Television" ; vous pourriez le placer dans le bloc **var** de l'interface de l'unité, mais nous pouvons le placer à un endroit plus stratégique. En effet, vous pouvez remarquer que le code source contient la déclaration de la classe "TfmPrinc". Si vous avez lu le chapitre 10, vous savez que cette classe est celle qui permet de définir la fiche. Nous avons tout à fait le droit d'ajouter des éléments à cette classe, et nous allons justement y ajouter un champ... Tele1. Attention de ne pas tout confondre : "Tele1" est un objet de classe "Television", mais ce sera également un champ de la classe "TfmPrinc" (il est tout à fait permis d'utiliser des objets en tant que champs). Voici un morceau de code source pour vous aider :

```
...
    btChangerReglage: TButton;
    btQuitter: TButton;
  private
    { Déclarations privées }
    Tele1: Television;
  public
...
```

L'étape suivante consiste à permettre de construire et détruire l'objet depuis l'interface de l'application. Lorsque l'objet sera créé, l'interface affichera le panneau "pnTele" qui permet le contrôle de la télévision. La procédure de réponse au clic sur le bouton de création va devoir créer l'objet "Tele1" en appelant le constructeur de la classe "Television", afficher le panel "pnTele" et mettre son contenu à jour. Il est également nécessaire de désactiver le bouton de création pour ne pas recréer l'objet une autre fois avant de l'avoir détruit ; pour pouvoir le détruire, on doit également activer le bouton de destruction. Voici le code source de la procédure de réponse au clic :

```
procedure TfmPrinc.btCreerTeleClick(Sender: TObject);
begin
  btCreerTele.Enabled := False;
  btDetrTele.Enabled := True;
  Tele1 := Television.Create;
  pnTele.Visible := true;
  MajTele;
end;
```

Juste une remarque à propos de l'accès au champ "Tele1" : vous le manipulez depuis une méthode (même si pour l'instant, nous parlions de procédures, ce qui ne sera PLUS le cas désormais) de la même classe, ce qui fait que vous y avez accès directement, comme si c'était une variable globale. Il nous reste à écrire une méthode "MajTele" qui mettra à jour le contenu du panel "pnTele". J'ai bien dit méthode car nous allons écrire une procédure faisant partie de la classe TfmPrinc (donc une méthode). Voici sa déclaration dans celle de la classe "TfmPrinc" :

```
...
  private
    { Déclarations privées }
    Tele1: Television;
    procedure MajTele;
  public
...
```

Cette procédure doit mettre à jour les deux zones d'édition de poids et de diagonale, activer ou non les boutons d'allumage/extinction et afficher/masquer le panel pnChaines si la télévision n'est pas allumée. Si la télévision est allumée, nous ferons appel à une autre méthode appelée MajChaines pour mettre à jour le contenu du panneau de visualisation des chaînes. Voici le code source de "MajTele" :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
procedure TfmPrinc.MajTele;
begin
  edPoids.Text := IntToStr(Tele1.poids);
  edDiago.Text := IntToStr(Tele1.longueur_diagonale);
  btAllumer.Enabled := not Tele1.allume;
  btEteindre.Enabled := Tele1.allume;
  pnChaines.Visible := Tele1.allume;
  if Tele1.allume then
    MajChaines;
end;
```

La méthode MajChaines doit être écrite de la même manière que "MajTele" (déclarez-là vous-même dans la déclaration de la classe "TfmPrinc"). Elle vide puis remplit la zone de liste "lbChaines" avec les chaînes mémorisées dans l'objet "Tele1". De plus, elle active le premier élément de la liste afin d'activer son édition (nous nous occuperons de cela plus tard). Voici son code source, assez simple, qui fait appel au champ "chaines_memorisees" de "Tele1" :

```
procedure TfmPrinc.MajChaines;
var
  i: integer;
begin
  lbChaines.ItemIndex := -1;
  lbChaines.Items.Clear;
  for i := 1 to MAX_CHAINES do
    lbChaines.Items.Add('Chaine '+IntToStr(i)+' : '+IntToStr(Tele1.chaines_memorisees[i]));
end;
```

Il est facile de faire fonctionner le bouton de destruction de la télévision. Un clic sur ce bouton doit cacher l'interface d'édition de la télévision et détruire l'objet "Tele1". Voici son code source :

```
procedure TfmPrinc.btDetrTeleClick(Sender: TObject);
begin
  btDetrTele.Enabled := false;
  btCreerTele.Enabled := true;
  pnTele.Visible := false;
  Tele1.Destroy;
end;
```

L'étape suivante consiste maintenant à faire fonctionner les deux boutons "Fixer" qui permettent respectivement de fixer le poids et la diagonale de la télévision. A chaque fois, on regarde le contenu de la zone d'édition associée et on essaie de construire une valeur entière. En cas de réussite cette valeur est affectée au champ correspondant de l'objet "Tele1" et en cas d'échec la valeur en cours est rétablie dans la zone d'édition. les deux méthodes de réponse aux clics sont très similaires, voici l'une des deux, écrivez l'autre vous-même à titre d'exercice :

```
procedure TfmPrinc.btFixePoidsClick(Sender: TObject);
var
  V, E: integer;
begin
  Val(edPoids.Text, V, E);
  if (E = 0) then
    Tele1.poids := V
  else
    begin
      ShowMessage(edPoids.Text + ' n''est pas une valeur entière acceptable');
      edPoids.Text := IntToStr(Tele1.poids);
    end;
end;
```

Il est maintenant temps de faire fonctionner les deux boutons servant à allumer et à éteindre la télévision. Ces deux boutons fonctionnent suivant le même principe que les deux boutons de création/destruction, à savoir qu'ils s'excluent



Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

mutuellement, et que le bouton d'allumage doit afficher et mettre à jour le panel d'édition des chaînes. Voici le code source des 2 méthodes de réponse au clic sur ces deux boutons, qui n'ont rien de bien difficile :

```
procedure TfmPrinc.btAllumerClick(Sender: TObject);
begin
  btAllumer.Enabled := false;
  btEteindre.Enabled := true;
  pnChaines.Visible := true;
  MajChaines;
end;

procedure TfmPrinc.btEteindreClick(Sender: TObject);
begin
  pnChaines.Visible := false;
  btAllumer.Enabled := true;
  btEteindre.Enabled := false;
end;
```

Il ne nous reste plus, pour terminer cette petite application fort inutile (!), qu'à faire fonctionner la mémorisation des chaînes et à régler un ou deux petits détails. La liste des chaînes doit fonctionner selon le principe suivant : l'utilisateur sélectionne une chaîne dans la liste, ce qui affiche le réglage dans la zone d'édition de droite. Le fait de modifier cette valeur puis de cliquer sur "Changer Réglage" met à jour la valeur dans l'objet "Tele1" ainsi que dans la liste des chaînes. Etant donné que l'application que nous développons est très simples, nous irons à l'essentiel sans prendre de détours ni trop de précautions, l'essentiel étant de vous montrer encore une fois comment utiliser l'objet "Tele1".
La première chose à faire est de réagir à un clic sur la liste pour gérer le changement de sélection (lorsque la sélection est modifiée au clavier et non à la souris, l'événement OnClick se produit tout de même, ce qui fait que nous pouvons utiliser cet événement pour gérer les changements de sélection par l'utilisateur). Lors d'un changement de sélection, le numéro de l'élément sélectionné donne un index permettant d'obtenir la valeur du réglage de la chaîne correspondante. Cette valeur doit alors être affichée dans la zone d'édition. Voici le code source de la méthode de réponse au clic sur la zone de liste :

```
procedure TfmPrinc.lbChainesClick(Sender: TObject);
begin
  if (lbChaines.ItemIndex >= 0) then
    edValeur.Text := IntToStr(Tele1.chaines_memorisees[lbChaines.ItemIndex + 1])
  else
    edValeur.Text := '';
end;
```

Juste une remarque sur le code ci-dessus : le numéro d'élément sélectionné (donné par ItemIndex) commence à 0, or les éléments du tableau de chaînes sont indexés à partir de 1, d'où le décalage. Méfiez-vous de ce genre de petit piège très fréquent. Passons enfin à la réaction au clic sur le bouton "btChangerReglage". Ce bouton est toujours actif, ce qui nous contraint à effectuer quelques petits tests. En effet, si l'utilisateur clique sur ce bouton alors qu'aucune chaîne n'est sélectionnée, il faudra lui expliquer qu'il doit d'abord en sélectionner une avant de pouvoir modifier son réglage. Voici le code source (un plus imposant) de la méthode en question :

```
procedure TfmPrinc.btChangerReglageClick(Sender: TObject);
var
  V, E: integer;
begin
  if (lbChaines.ItemIndex >= 0) then
    begin
      Val(edValeur.Text, V, E);
      if (E = 0) then
        begin
          Tele1.memoriser_chaine(lbChaines.ItemIndex + 1, V);
          lbChaines.Items[lbChaines.ItemIndex] :=
            'Chaine ' + IntToStr(lbChaines.ItemIndex + 1) + ' : ' +
            IntToStr(Tele1.chaines_memorisees[lbChaines.ItemIndex + 1]);
        end
      else
        begin
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
          ShowMessage(edValeur.Text + ' n''est pas une valeur entière acceptable');
          edValeur.Text := IntToStr(Tele1.chaines_memorisees[lbChaines.ItemIndex + 1]);
        end;
    end
  else
    begin
      ShowMessage('Vous devez d''abord sélectionner une chaîne avant de modifier son réglage');
      edValeur.Text := '';
    end;
end;
```

Le développement de cette petite application exemple est désormais terminé. Il y aurait encore des améliorations à voir, comme vérifier si "Tele1" a bel et bien été détruit avant de quitter l'application (ce n'est pas trop grave puisque Windows récupèrera la mémoire tout seul). Il serait également intéressant de filtrer les valeurs négatives pour le poids et la longueur de diagonale. Libre à vous d'essayer d'intégrer ces améliorations au projet. Vous pouvez télécharger le code source complet **sans** ou **avec** ces améliorations (attention, à partir de maintenant, les codes source téléchargeables sont créés avec Delphi 6, ce qui ne devrait pas poser de problème avec Delphi 5. Pour les versions antérieures, je suis dans l'incapacité d'effectuer les tests, désolé).
Nous allons maintenant compléter notre implémentation de la classe "Television" en réalisant l'implémentation correspondant au schéma incluant l'appareil à écran. Tout d'abord, il nous faut remarquer les différences avec l'ancien schéma pour faire l'adaptation du code source en douceur.

- Il nous faut une classe "AppareilAEcran", qui va prendre quelques-uns des champs et méthodes de la classe "Television" ;
- "Television" doit étendre la classe "AppareilAEcran"
- Il nous faut une classe "EcranOrdinateur"...

Commençons par écrire la déclaration de la classe "AppareilAEcran". Elle n'étend aucune classe (explicitement, car comme je l'ai déjà dit, elle étendra "TObject"), a comme champs "allume", "poids" et "diagonale" et comme méthodes "allumer" et "eteindre". Essayez d'écrire vous-même la déclaration de cette classe (sans le code source des méthodes), puis regardez la correction ci-dessous :

```
 AppareilAEcran = class
    poids: integer;
    longueur_diagonale: integer;
    allume: boolean;
    procedure allumer;
    procedure eteindre;
  end;
```

L'écriture de cette partie ne doit pas vous avoir posé trop de problèmes, puisqu'il suffisait de calquer la déclaration sur celle de la classe "Television". Ecrivez maintenant le code source des méthodes de la classe "AppareilAEcran" (petit raccourci : placez-vous à l'intérieur de la déclaration de la classe "AppareilAEcran", et utilisez le raccourci clavier Ctrl +Shift+C, qui doit vous générer le squelette de toutes les méthodes). Voici ce que cela doit donner :

```
procedure AppareilAEcran.allumer;
begin
  allume := true;
end;

procedure AppareilAEcran.eteindre;
begin
  allume := false;
end;
```

Dans l'état actuel des choses, nous avons deux classes, "AppareilAEcran" et "Television", complètement indépendantes. Nous voudrions bien faire en sorte que "Television" étende "AppareilAEcran". Pour cela, il va falloir


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

procéder à quelques modifications. La manoeuvre de départ consiste à dire explicitement que "Television" étend "AppareilAEcran". Pour cela, modifiez la première ligne de la déclaration de la classe "Television" en ceci :

```
Television = class(AppareilAEcran)
```

Une fois cette première manipulation effectuée, les deux classes sont bien reliées entre elles comme nous le voulions, cependant, nous avons déclaré deux fois les champs et méthodes communs aux deux classes, ce qui est une perte de temps et de code source, et surtout une erreur de conception. Rassurez-vous, ceci était parfaitement délibéré, histoire de vous montrer que l'on peut retirer une partie du code qui est actuellement écrit.
Pour preuve, si vous compilez actuellement le projet comprenant la déclaration des 2 classes et de leurs méthodes, aucune erreur ne se produira. Cependant, nous allons maintenant profiter d'une des fonctionnalités les plus intéressantes de la programmation objet : l'héritage. Comme "Television" étend "AppareilAEcran", elle *hérite* de tout le contenu de cette classe. Ainsi, le fait de redéfinir les champs "allume", "poids", "longueur_diagonale" et les méthodes "allumer" et "eteindre" dans la classe "Television" est une redondance. Vous pouvez donc supprimer ces déclarations. Faites le et supprimez également le code source des deux méthodes "allumer" et "eteindre" de la classe "Television" (supprimez les deux méthodes dans la partie **implementation**). Voici la nouvelle déclaration de la classe "Television" :

```
Television = class(AppareilAEcran)
   chaines_memorisees: array[1..MAX_CHAINES] of integer;
   procedure memoriser_chaine (numero, valeur: integer);
 end;
```

Si vous compilez à nouveau le code source, vous ne devez avoir aucune erreur de compilation. A titre de vérification, vous pouvez consulter le **code source** dans la version actuelle ici (c'est devenu une unité à part entière). Vous vous demandez peut-être encore comment tout cela peut fonctionner. Eh bien en fait, comme la classe "Television" étend la classe "AppareilAEcran", elle hérite de tous ses éléments, et donc des fonctionnalités implémentées dans cette classe. Ainsi, il est possible d'allumer et d'éteindre une télévision, car une télévision est avant tout un appareil à écran auquel on a ajouté des particularités.
Nous allons maintenant implémenter la classe "EcranOrdinateur" à partir de la classe "AppareilAEcran". Voici la déclaration de base de cette classe :

```
EcranOrdinateur = class(AppareilAEcran)
end;
```

Si vous regardez un peu la liste des éléments introduits dans la classe "EcranOrdinateur", vous constatez qu'il est question de résolutions. Pour définir une résolution, nous allons utiliser un enregistrement dont voici la définition :

```
TResolution = record
  hauteur,
  largeur: word;
end;
```

Les deux champs de la classe sont assez simples, il s'agit d'un élément de type "TResolution" (déclaré ci-dessus), et d'un tableau dynamique d'éléments de type "TResolution". La seule méthode, permettant de fixer la résolution actuelle, accepte un paramètre de type résolution et doit vérifier avant de l'appliquer qu'elle fait partie des résolutions possibles. Voici la déclaration complète de la classe "EcranOrdinateur" :

```
EcranOrdinateur = class(AppareilAEcran)
   resolutions_possibles: array of TResolution;
   resolution_actuelle: TResolution;
   function changer_resolution(NouvRes: TResolution): boolean;
 end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'implémentation de la méthode "changer_resolution" consiste à parcourir le tableau des résolutions possibles, et dés que la résolution a été trouvée, elle est appliquée, sinon, elle est rejetée. le résultat indique si le changement a été oui ou non accepté. Voici le code source de cette méthode, qui manipule les deux champs introduits dans la classe "EcranOrdinateur" :

```
function EcranOrdinateur.changer_resolution(NouvRes: TResolution): boolean;
var
  i: integer;
begin
  i := 0;
  result := false;
  while (i < length(resolutions_possibles)) and not result do
    if resolutions_possibles[i].hauteur = NouvRes.hauteur and
       resolutions_possibles[i].Largeur then
      begin
        resolution_actuelle := NouvRes;
        result := true;
      end;
end;
```

Et c'en est fini des déclarations. La classe "EcranOrdinateur" étendant la classe "AppareilAEcran", elle hérite de tous ses éléments, et chaque objet instance de la classe "EcranOrdinateur" peut donc être allumé ou éteint (en tant qu'appareil à écran particulier).
Ceci termine ce long point sur la création et la manipulation des objets. Je vous engage à réaliser une petite application ressemblant à celle créée pour la classe "Television", et à faire fonctionner un "écran d'ordinateur". Vous pouvez également remplacer la définition de "Television" (dans l'application que nous avons créé un peu avant) par la définition complète des trois classes. Vous verrez que tout fonctionne exactement comme si vous n'aviez effectué aucun changement.

## XVI-C-4 - Utilisation d'un constructeur et d'un destructeur, notions sur l'héritage

Créer un objet peut parfois nécessiter des actions spéciales, comme l'initialisation de valeurs de champs ou l'allocation de mémoire pour certains champs de l'objet. Ainsi, si vous utilisez des champs de type pointeur, vous aurez probablement besoin d'allouer de la mémoire par des appels à new. Vous pouvez également avoir besoin de fixer des valeurs directement à la construction, sans avoir à les respécifier plus tard. Lorsque ce genre de besoin apparaît, il faut systématiquement penser « constructeur ».
Le constructeur, comme vous l'avez peut-être lu dans la première partie du chapitre, est une méthode spéciale qui est exécutée au moment de la construction d'un objet. Pour la petite histoire, ce n'est pas une vraie méthode, puisqu'on ne l'appelle pas à partir d'un objet mais d'une classe. C'est ce qu'on appelle une « méthode de classe ». Cette méthode de classe effectue en fait elle-même la création physique de l'objet en mémoire, et le retourne indirectement (pas en tant que résultat de méthode, puisque le constructeur n'est pas une méthode), ce qui permet une affectation du "résultat" de l'appel du constructeur. La seule chose que vous ayez à retenir ici, c'est que grâce au mécanisme appelé *héritage* dont nous avons vu une partie de la puissance, vous pouvez écrire votre propre constructeur (voire plusieurs), faisant appel au constructeur système pour lui déléguer les tâches difficiles de création de l'objet, et réalisant ensuite autant d'actions que vous le désirez, et en plus des tâches systèmes de construction ayant été réalisées auparavant. L'objet récupéré lors d'un appel à ce constructeur *personnalisé* aura alors déjà subi l'action du constructeur système et l'action de toutes les instructions que vous aurez placé dans votre constructeur personnalisé.
Comme un exemple fait souvent beaucoup de bien après un grand discours comme celui-ci, voici la toute première version du constructeur que nous pourrions écrire pour la classe "MachineAEcran" :

```
constructor AppareilAEcran.Create;
begin
  allume := false;
  poids := 20;
  longueur_diagonale := 55;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Et voici sa déclaration dans celle de la classe "AppareilAEcran" :

```
AppareilAEcran = class
    ...
    procedure eteindre;
    constructor Create;
  end;
```

Un constructeur se déclare non pas à l'aide d'un des mots réservés **procedure** ou **function**, mais à l'aide du mot-clé réservé à cet usage : **constructor**. Vient ensuite le nom du constructeur ; le nom classique à conserver dans la mesure du possible est « Create ». Peut éventuellement suivre une liste de paramètres qui est absente dans le cas présent.
Il est possible de définir un destructeur de la même manière, à une petite difficulté près, nous allons le voir. Le destructeur est une méthode classique déclarée d'une manière spécifique, et qui est appelée afin de détruire l'objet duquel on a appelé le destructeur. Le nom habituel du destructeur est "Destroy", et c'est une méthode sans paramètre. Voici comment on déclare un destructeur dans la déclaration d'une classe :

```
destructor Destroy; override;
```

La partie **destructor** Destroy; permet de dire qu'on définit un destructeur personnalisé pour la classe en cours de déclaration. Ce destructeur est introduit par le mot réservé **destructor** suivi du nom, traditionnellement "Destroy", puis d'un point-virgule puisque le destructeur n'a normalement pas de paramètre. La suite, à savoir « **override** », est une nouveauté et nous allons y consacrer toute notre attention.
Ce mot-clé, suivant la déclaration du destructeur et lui-même suivi immédiatement d'un point-virgule signale que nous appliquons une *surcharge* à une méthode déjà existante (j'ai lâché là un des gros mots de la programmation objet). Ce terme de *surcharge*signifie que la méthode que nous allons définir va s'appuyer sur la méthode de même nom, mais déclarée dans la classe parente. Ainsi, la classe parente de "AppareilAEcran", à savoir "TObject", possède déjà un destructeur appelé "Destroy". Nous allons donc redéfinir une méthode qui existe déjà, mais en étendant ses possibilités. Dans la grande majorité des cas, ceci permet d'adapter le fonctionnement à la nouvelle classe en gérant ce qui a été ajouté à la classe parente : Il est alors possible d'écrire les instructions spécifiques à l'extension de la classe parente, puis de déléguer le reste du travail à réaliser par cette méthode, à savoir le traitement des données de base de la classe parente (en appelant la méthode de même nom de la classe parente afin de permettre le traitement qui y était effectué par cette même méthode).
Je m'explique : TObject possède une méthode (plus exactement un destructeur) appelé Destroy, qui est parfaitement adaptée à TObject. Si nous créons une nouvelle classe dont les objets manipulent des pointeurs, il serait sage qu'à la destruction de chaque objet, la mémoire allouée aux pointeurs soit du même coup libérée. Il serait tout à fait possible d'écrire une méthode autre que le destructeur qui réaliserait cela, mais il n'y aurait aucun moyen de s'assurer que la méthode en question soit appelée quand on détruit un objet (oubli ou mauvaise manipulation par exemple). Le destructeur est l'endroit idéal où placer ces instructions, mais là où le constructeur nous facilitait la tâche en nous permettant d'écrire des instructions sans nous préoccuper du constructeur système, le destructeur est plus vache et on doit explicitement effectuer les tâches manuelles liées à la destruction, mais aussi faire en sorte que le destructeur de la classe parente soit appelé (ce ne sera pas le cas par défaut !).
Voici le code source de base du destructeur déclaré plus haut :

```
destructor AppareilAEcran.Destroy;
begin
  inherited;
end;
```

Plusieurs choses sont à remarquer dans cet extrait de code source : on utilise dans la ligne de déclaration non pas **procedure** ou **function** mais **destructor**. Suit ensuite la déclaration classique d'une méthode. Notez l'absence du mot-clé **override** : ce mot-clé ne doit être présent que dans la déclaration de classe. Le corps du destructeur réserve une petite surprise : la seule instruction est un simple « **inherited**; ». Cette instruction signifie : "Appeler la méthode de même nom, déclarée dans la classe parente de celle que je suis en train d'implémenter". **inherited** appelle donc ici le destructeur Destroy de la classe "TObject", ce qui permettra de détruire l'objet en cours lors d'un appel à son


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

destructeur, mais également de réaliser d'autres tâches **avant** (il serait impossible et quelque peu... bête de vouloir appliquer des instructions à un objet dont le coeur aurait déjà été détruit).
La programmation objet ayant parmi ses objectifs la réutilisation de code existant sans avoir à le réécrire, le fait de *surcharger* une méthode permet de reprendre avantageusement toute un lot d'instructions sans les reprogrammer puisqu'elles sont présentes dans la classe parente de la classe qu'on est en train de programmer. Cette *surcharge*, qui est l'un des aspects majeurs d'une notion plus vaste # l'*héritage* # est une manière d'étendre à chaque fois que nécessaire, lorsqu'on crée une classe à partir d'une autre, les méthodes qui le nécessitent, ce qui rend possible l'extension du code source des méthodes au fur et à mesure qu'on passe d'une classe parente à une classe dérivée en augmentant les possibilités des classes en question à chaque étape.
Revenons au code source du destructeur de la classe AppareilAEcran : comme il vaut mieux s'assurer qu'un appareil à écran est éteint avant de le détruire (précaution élémentaire, n'est-il pas ?), nous allons fixer la valeur du champ "allume" d'un objet destiné à être détruit à faux avant de procéder à la destruction proprement dite. Comme il a été mentionné plus tôt, il est impératif de réaliser cette opération **d'abord** et de poursuivre par le destruction faite dans la classe parente **ensuite**. Voici le code source mis à jour :

```
destructor AppareilAEcran.Destroy;
begin
  allume := false;
  inherited;
end;
```

Il y a peu à dire sur ce code source : remarquez encore une fois que les nouvelles instructions, dans le cas du destructeur, doivent être obligatoirement effectuées avant l'appel à la méthode de même nom héritée (inherited...). Notez que pour les autres méthodes que le destructeur, ce sera généralement l'inverse : on effectue d'abord les traitements de base par un appel à la méthode de même nom héritée, et on effectue ensuite les traitements à ajouter pour prendre en compte les éléments ajoutés à la classe parente. Nous allons tout de suite voir un exemple concret, en écrivant un constructeur pour chacune des deux classes descendantes de la classe "AppareilAEcran".
Pour ce qui concerne les télévisions, il serait intéressant d'initialiser l'ensemble des chaînes au réglage 0. Les écrans d'ordinateurs, eux, pourraient définir quelques résolutions autorisées systématiquement, comme le 640 x 480. Le fait d'écrire ces deux constructeurs va non seulement vous montrer l'intérêt de l'héritage, mais va aussi vous permettre de découvrir une nouvelle notion, celle de méthode *virtuelle*, autre notion connexe à l'héritage.
En ce qui concerne la classe "Television", voici la déclaration du nouveau constructeur :

```
constructor Television.Create;
var
  i: integer;
begin
  for i := 1 to MAX_CHAINES do
    chaines_memorisees[i] := 0;
end;
```

Ce constructeur n'est qu'à moitié intéressant : il permet bien d'initialiser les champs de l'objet qui vient d'être créé, mais il ne fait pas appel au constructeur défini dans la classe "AppareilAEcran", ce qui fait que par défaut, vu notre implémentation actuelle de ce constructeur, une télévision sera, par défaut, éteinte, avec un poids de 0 kg (les champs d'un objet sont fixés par défaut à la valeur nulle de leur type (soit 0 pour les entiers, nil pour les pointeurs, '' pour les chaînes...) et une longueur de diagonale de 0 cm, ce qui est regrettable. Nous allons donc modifier le constructeur et sa déclaration pour qu'il fasse appel au constructeur de la classe "AppareilAEcran". Voici la déclaration modifiée :

```
  ...
    procedure memoriser_chaine (numero, valeur: integer);
    constructor Create; override;
  end;
...
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

et le code source, qui fait appel au constructeur hérité (méthode Create de même nom dans la classe parente) (n'essayez pas de compiler le code complet avec l'extrait ci-dessous, un oubli tout à fait délibéré provoquant une erreur a été fait afin de vous expliquer une nouvelle notion) :

```
constructor Television.Create;
var
  i: integer;
begin
  inherited;
  for i := 1 to MAX_CHAINES do
    chaines_memorisees[i] := 0;
end;
```

Comme il est précisé en remarque ci-dessus, si vous tentez de compiler le code source modifié à l'aide du code source ci-dessus (si vous n'avez pas suivi, vous pouvez le **télécharger dans son état actuel ici**), une erreur se produit, vous indiquant qu'il est « impossible de surcharger une méthode statique ». Mais bon sang mais c'est bien sûr !
Lorsqu'une méthode est écrite sans y ajouter le mot-clé **override** (ou un ou deux autres que nous allons voir tout de suite), elle est dite *statique*, c'est-à-dire qu'elle n'est pas candidate à la surcharge. Ainsi, il ne sera pas possible de la surcharger (mais de la *redéfinir*, nous y reviendrons). Bref, notre premier constructeur a été défini sans ce mot-clé, et il est donc statique, et ne peut donc pas être surchargé. Qu'à cela ne tienne, il va falloir faire en sorte qu'il accepte la surcharge. Pour cela, on ne va pas utiliser le mot-clé **override** car le constructeur de AppareilAEcran ne surcharge aucune autre méthode, mais plutôt le mot-clé **virtual** destiné précisément à cet usage. On dira alors que le constructeur de la classe "AppareilAEcran" est virtuel, c'est-à-dire qu'il est susceptible d'être surchargé dans une classe ayant "AppareilAEcran" comme classe parente. Voici la déclaration modifiée du premier constructeur :

```
  ...
    { dans la classe AppareilAEcran }
    procedure eteindre;
    constructor Create; virtual;
    destructor Destroy; override;
  end;
...
```

Si vous compilez le code avec la nouvelle déclaration du constructeur, tout se passe bien, maintenant que le premier constructeur est déclaré virtuel, c'est-à-dire apte à être surchargé, et que le second est déclaré comme surchargeant la méthode héritée, qui peut bien être surchargée. Que se passerait-il si nous définissions une classe ayant "Television" comme classe parente et si nous décidions d'y inclure un constructeur faisant appel à celui de la classe "Television" ? Nous devrions lui adjoindre le mot-clé **override**, signalant que nous surchargeons une méthode héritée. En effet, le mot-clé **override** signifie non seulement que l'on va surcharger une méthode, mais que cette surcharge pourra elle-même être surchargée.
Le fait d'écrire un constructeur pour la classe "Television" fait que la création de tout objet de cette classe se fera en plusieurs étapes. Dans un premier temps, c'est la méthode héritée qui est appelée par **inherited**. Ce constructeur hérité fait lui-même appel implicitement appel au constructeur système de la classe "TObject", puis initialise certains champs. Enfin, de retour dans le constructeur de la classe "Television", les chaînes sont initialisées après que le reste des champs de l'objet aient été initialisés et que l'objet lui-même ait été construit.
Nous aurons l'occasion de voir d'autres exemples d'utilisation des méthodes virtuelles et de la surcharge, et nous reparlerons également du mécanisme d'héritage plus en détail dans un prochain paragraphe. Pour l'instant, je vais passer à un tout autre sujet plus facile, histoire de ne pas trop vous démoraliser.

## XVI-C-5 - Visibilité des membres d'une classe

Tout ce que nous avons déclaré jusqu'à présent comme faisant partie d'une classe, que ce soit des champs, des méthodes, un constructeur ou un destructeur (auxquels s'ajouterons plus tard les propriétés) constitue ce qu'on appelle communément les *membres* de cette classe. On entend par membre un élément déclaré dans une classe, que ce soit un élément normal comme un champ, une méthode... ou même une méthode surchargeant une méthode héritée.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Jusqu'à maintenant, l'ensemble des membres de toutes nos classes avaient une caractéristique dont nous ne nous sommes pas préoccupé vu la petite taille des classes que nous manipulions. Le moment est venu de parler de cette caractéristique que vous avez peut-être remarquée : tous les membres de nos classes sont librement accessibles depuis l'extérieur des objets de ces classes. Ceci ne pose pas trop de problèmes tant qu'on manipule de toutes petites classes, ou qu'on se limite à une utilisation strictement limitée du code source qu'on écrit.
La programmation objet met à votre disposition le moyen de masquer certains éléments afin de ne pas permettre leur accès depuis l'extérieur. Ainsi, on rend souvent la plupart des champs d'une classe inaccessibles de l'extérieur. Pourquoi ? Tout simplement pour éviter une modification hors de tout contrôle, car rappelons-le, une classe peut constituer une petite usine à gaz qui ne demande qu'à exploser lorsqu'un champ prend une valeur imprévue. Il sera possible, en définissant des méthodes dédiées, de permettre la lecture de tel ou tel champ et même d'autoriser l'écriture de manière contrôlée en passant par des méthodes. Ainsi, pour un champ, qu'on a en général intérêt à rendre innaccesslble de l'extérieur, on programme en général deux méthodes appelées *accesseurs*, que l'on laisse visibles de l'extérieur. L'un de ces deux *accesseurs* est en général une fonction (mais qui reste une méthode puisque c'est un membre d'une classe, vous me suivez ?) qui permet de lire la valeur du champ. Le second accesseur est en général une procédure (encore une méthode) qui permet de fixer la valeur du champ, ce qui permet d'effectuer des tests et éventuellement de refuser la nouvelle valeur proposée si elle risque de compromettre le bon fonctionnement de l'objet sur lequel elle a été appelée.
Dès qu'on veut décider de la visibilité des membres vis-à-vis de l'extérieur, il faut préfixer la déclaration de ces membres à l'aide d'un des 4 mots-clé **private**, **protected**, **public** ou **published**. Il est possible de déclarer la visibilité de chaque membre individuellement mais on préfère généralement regrouper les membres par groupes de visibilité en débutant chaque groupe par le mot réservé qui convient. Chaque mot réservé dénote une visibilité différente, définissant les restrictions d'accès aux membres ayant cette visibilité. La visibilité décide en fait quels éléments ont accès à un membre particulier. Si ce membre est un champ, y accéder signifie pouvoir le lire ou le modifier, si c'est une méthode, on peut l'exécuter (nous verrons plus loin le cas des propriétés). En général, le constructeur et le destructeur ont la visibilité **public**. Voici la signification des 4 visibilités possibles dans Pascal Objet (en fait, il en existe une cinquième, **automated**, je n'en dirai pas un mot de plus) :

- **private** (membres privés)
  C'est la visibilité la plus faible, qui permet de restreindre au maximum les accès "sauvages" (non autorisés). Les membres ayant cette visibilité ne sont accessibles que dans l'unité où la classe est déclarée (et PAS comme dans certains langages uniquement dans la classe). Tout élément de la même unité a accès à ces membres. Les membres des classes descendantes déclarées dans une autre unité n'ont pas accès à ces membres. Cette visibilité est idéale pour cacher les détails d'implémentation d'une classe (en ne laissant visible au programmeur que ce qui le concerne et qui n'est pas "dangereux" à utiliser (avez-vous remarqué à quel point l'informatique et plus particulièrement la programmation est une activité où reigne la paranoîa ?)).
- **protected** (membres protégés)
  Visibilité intermédiaire. Elle donne accès à tous les éléments autorisés par la visibilité **private** et y ajoute l'ensemble des membres des classes descendantes de la classe du membre, même et surtout celles qui ne sont pas déclarées dans la même unité. Il est par contre impossible d'accéder à ces éléments depuis un autre endroit d'une application, ce qui permet d'accéder à ces membres lorsqu'on crée une classe descendante de leur classe d'appartenance, sans pour autant les laisser en libre accès.
- **public** (membres publics)
  Visibilité maximale (au même titre que **published**). Les membres ayant cette visibilité sont accessibles de partout, c'est-à-dire depuis tous les endroits autorisés par protected, auxquels viennent s'ajouter l'ensemble des éléments des unités utilisant (au sens de **uses**) l'unité où est déclarée la classe. Lorsqu'aucune visibilité n'est définie pour les membres d'une classe, c'est cette visibilité qui est appliquée par défaut.
- **published** (membres publiés)
  Visibilité spéciale, réservée aux propriétés (nous allons y venir, ne vous inquiétez pas, ce n'est pas très compliqué). Les propriétés déclarées dans ce groupe de visibilité ont la particularité d'apparaître dans l'inspecteur d'objets dans la page des propriétés ou des événements (selon les cas) lorsqu'une instance de cette classe est éditée. Certaines restrictions s'appliquent ici, nous aurons tout le loisir d'y revenir en détail.

Nous allons modifier la visibilité des différents membres des classes que nous avons vues jusqu'à présent. Les éléments qui n'ont pas a être modifiés de l'extérieur vont être placés dans une section privée, et les autres dans une section publique. Nous n'utiliserons pas de section protégée car aucune classe descendante des nôtres n'est au programme dans une autre unité. A l'occasion, nous allons écrire des accesseurs, à savoir, je le rappelle, des méthodes dédiées à l'accès en lecture et éventuellement en écriture de certains champs.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Commençons par la classe "AppareilAEcran" : voici sa nouvelle déclaration, qui ne change rien au niveau de l'implémentation.

```
 AppareilAEcran = class
  private
    // les 3 éléments ci-dessous ont une visibilité "privée"
    poids: integer;
    longueur_diagonale: integer;
    allume: boolean;
  public
    // les 4 éléments ci-dessous ont une visibilité "publique"
    procedure allumer;
    procedure eteindre;
    constructor Create; virtual;
    destructor Destroy; override;
  end;
```

Vous pouvez voir dans l'extrait de code précédent comment on déclare une section privée (mot-clé **private** en général seul sur une ligne, suivi des déclarations des membres dont la visibilité doit être privée) ou publique (de même). Il est à noter que dans chacune de ces sections, les champs doivent être tous déclarés d'abord, et les méthodes ensuite. Plusieurs sections de chaque sorte peuvent être présentes, mais cela a en général peu d'intérêt et il est recommandé d'utiliser le schéma suivant :

```
ClasseDescendante = class(ClasseParente)
private
  { déclarations de membres privés }
protected
  { déclarations de membres protégés }
public
  { déclarations de membres publics }
published
  { déclarations de membres publiés }
end;
```

Le fait de déclarer les champs comme privés pose cependant le souci suivant : le code source qui les manipulait directement auparavant est maintenant caduc, et devra être modifié. Pour accéder aux champs, il va nous falloir pour chaque une méthode de lecture (généralement, on parle de "get", en pur jargon informatique) et éventuellement une autre d'écriture (on parle de "set"). Voici la déclaration de deux méthodes permettant l'accès en lecture et en écriture du champ "poids" d'un appareil à écran :

```
 ...
    { dans la classe AppareilAEcran, section publique }
    destructor Destroy; override;
    function getPoids: integer;
    procedure setPoids(valeur: integer);
  end;
...
```

L'implémentation de la méthode de lecture est des plus simple puisqu'il suffit de retourner en résultat la valeur du champ dont la valeur est demandée. La voici :

```
function AppareilAEcran.getPoids: integer;
begin
  Result := poids;
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

C'est en écrivant le code source de la méthode d'écriture que l'on comprend mieux l'intérêt de ce genre de procédé. Il va être possible de tester la valeur fournie avant d'accepter l'écriture de cette valeur. Les valeurs négatives seront ainsi refusées. Voici le code source de cette méthode :

```
procedure AppareilAEcran.setPoids(valeur: integer);
begin
  if valeur >= 0 then
    poids := valeur;
end;
```

L'avantage d'avoir programmé ces deux méthodes et de les laisser en accès public tandis que le champ qui se cache derrière est en accès privé est que le programmeur, vous ou quelqu'un d'autre, est dans l'incapacité de "tricher" en affectant une valeur de manière sauvage au champ, ce qui pourrait compromettre le bon fonctionnement de l'objet. L'obliger à utiliser les méthodes de lecture et d'écriture *sécurise le programme*, ce qui est très important car c'est autant de bugs évités sans effort.
Ce sera tout pour ce qui est des visibilités. Nous reviendrons si l'occasion se présente sur la visibilité protégée, et surtout sur la visibilité publiée lorsque nous parlerons de la création de composants (qui sera l'objet d'un autre chapitre, vu que le sujet est très vaste...).

## XVI-C-6 - Propriétés

Les propriétés sont une des spécificités de Pascal Objet dans le domaine de la programmation orientée objets. A ma connaissance, on ne les retrouve dans aucun autre langage objet, et c'est bien dommage car elles procurent des fonctionnalités très intéressantes.
Les propriétés en Pascal Objet existent sous un bon nombre de formes et mériteraient un chapitre entier. Pour ne pas vous décourager, je vais plutôt m'attacher ici à décrire d'abord les plus simples, puis à augmenter progressivement la difficulté pour vous montrer tout ce qu'on peut faire à l'aide des propriétés. Nous nous limiterons cependant aux possibilités de base en laissant les plus complexes à un futur chapitre sur la création de composants.

### XVI-C-6-a - Propriétés simples

Une propriété est un membre spécial d'une classe permettant de fournir un accès en lecture et éventuellement en écriture à une donnée en ne révélant pas d'où on la sort lors de la lecture ni où elle va lors de l'écriture. Une propriété a un type qui donne le type de la donnée que l'on peut lire et éventuellement écrire. Ce type peut être de n'importe quel type prédéfini de Pascal Objet, ou un type créé dans un bloc **type**, y compris une classe.
Nous allons continuer l'écriture de nos trois classes en créant une première propriété permettant l'accès au poids. Voici la déclaration de cette propriété, telle qu'elle doit apparaître dans la section publique de la classe "AppareilAEcran" :

```
property Poids: integer
  read getPoids
  write setPoids;
```

Cette déclaration est celle d'une nouevlle propriété dont le nom est "Poids", dont le type est "integer", qu'on lit (read) via la méthode "getPoids" et qu'on écrit (write) via la méthode "setPoids". La propriété "Poids" va nous permettre d'accéder de manière sélective au champ qui porte pour l'instant le même nom (nous corrigerons cela plus tard, c'est une erreur). Lorsqu'on utilisera "Poids" dans un contexte de lecture, la méthode "getPoids" sera appelée de manière transparente et son résultat sera la valeur lue. Lorsqu'on utilisera "Poids" dans un contexte d'écriture (une affectation par exemple), la méthode "setPoids" sera de même appelée de manière transparente et son paramètre sera la valeur qu'on tente d'affecter à la propriété, ce qui fait que l'affectation pourra très bien ne pas être faite, par exemple si la valeur transmise est négative (rappelez-vous que la méthode "setPoids" ne fonctionne qu'avec des valeurs positives).
Il est à noter que les méthodes "getPoids" et "setPoids" doivent respecter une syntaxe particulière. "getPoids" doit être une fonction sans paramètre retournant un entier, ce qui est bien le cas, et "setPoids" doit être une procédure à un seul paramètre de type entier, ce qui est également le cas. Ces deux méthodes étant déjà programmées, il n'y a rien de plus à ajouter pour que la propriété fonctionne (mis à par un changement de nom du champs "poids").


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Il y a effectivemùent un petit souci puisque nous venons de déclarer deux membres avec le même nom, à savoir le champ "poids" et la propriété "Poids". Dans ce genre de situation, on a pour habitude (une habitude piochée dans le code source des composants de Delphi, ce qui prouve son bon sens) de préfixer les champs par un f ou un F (pour "Field", soit "Champ" en anglais). Ainsi, le champ poids sera renommé en "fPoids" et la propriété "Poids" gardera son nom. Afin de clarifier les choses, voici la déclaration complètement mise à jour de la classe "AppareilAEcran" :

```
 AppareilAEcran = class
  private
    fPoids: integer;
    longueur_diagonale: integer;
    allume: boolean;
  public
    procedure allumer;
    procedure eteindre;
    constructor Create; virtual;
    destructor Destroy; override;
    function getPoids: integer;
    procedure setPoids(valeur: integer);
    property Poids: integer
      read getPoids
      write setPoids;
  end;
```

N'oubliez pas de modifier le code source des deux méthodes "getPoids" et "setPoids" pour faire référence au champ "fPoids" et non à la propriété "Poids", car sinon je vous fais remarquer qu'il y aurait certainement des appels des deux méthodes en boucle lors de la lecture ou de l'écriture de "Poids" puisque chaque lecture appelle "getPoids" et chaque écriture appelle "setPoids".
Le fait d'utiliser une propriété a un autre intérêt majeur quand on part du principe qu'un minimum de visibilité pour chaque membre est une bonne idée (et c'est un bon principe !), est que les méthodes employées pour effectuer la lecture et l'écriture de la valeur de la propriété peuvent être "cachées", c'est-à-dire placées dans la section **private** de la classe. Ceci oblige à se servir de la propriété sans se servir explicitement des méthodes, ce qui cache au maximum le détail des opérations de lecture et d'écriture de la valeur : le programmeur n'a que faire de ces détails tant qu'ils sont effectués correctement. Déplacez donc les deux déclarations des méthodes "getPoids" et "setPoids" dans la section privée (**private**) de la déclaration de classe.
Afin de voir les appels implicites des méthodes "getPoids" et "setPoids", nous allons créer une mini-application exemple en modifiant le code des deux méthodes "getPoids" et "setPoids" pour qu'elles signalent leur appel par un simple "ShowMessage". Voici leur nouveau code source temporaire (n'oubliez pas d'ajouter les unité Dialogs (pour ShowMessage) et SysUtils (pour IntToStr) à la clause **uses** de la partie interface ou implémentation de l'unité) :

```
function AppareilAEcran.getPoids: integer;
begin
  ShowMessage('Appel de la méthode getPoids. Valeur retournée : '+IntToStr(fPoids));
  Result := fPoids;
end;

procedure AppareilAEcran.setPoids(valeur: integer);
begin
  if valeur >= 0 then
    begin
 
 ShowMessage('Appel de la méthode setPoids avec une valeur positive (acceptée) : '+IntToStr(valeur));
      fPoids := valeur;
    end
  else
    ShowMessage('Appel de la méthode setPoids avec une valeur négative (refusée) : '+IntToStr(valeur));
end;
```

Il est à noter que du fait de l'héritage et de sa visibilité publique, les deux classes descendantes "Television" et "EcranOrdinateur" ont accès à la propriété "Poids", mais pas aux deux méthodes "getPoids" et "setPoids" ni au champ "fPoids", ce qui interdit de surcharger les méthodes ou l'accès direct au champ sans passer par la propriété. Vous noterez que le fait d'utiliser une propriété laisse l'accès au "Poids" tout en limitant sa modification (poids positif).


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Créez une nouvelle application et ajoutez-y l'unité dans son état actuel. Placez une zone d'édition (edPoids) et deux boutons (btModifPoids et btQuitter) sur la fiche principale. Faites fonctionner le bouton "Quitter". Le principe de l'application va être tout simplement de proposer l'édition du poids d'une télévision. Le poids sera indiqué dans la zone d'édition, et la propriété "Poids" d'un objet de classe "Television" sera affectée lors du clic sur le bouton "Modifier Poids". Suivra une lecture pour connaître la valeur actuelle du poids, au cas où l'affectation précédente aurait été rejetée. Cette valeur sera placée dans la zone d'édition.
A cette occasion, nous allons apprendre à *utiliser* une propriété. Une propriété accessible en lecture/écriture s'utilise en tout point comme une variable ayant le type de la propriété, mis à part que la lecture et l'écriture sont effectuées de manière transparente par les accesseurs ("getPoids" et "setPoids" pour le poids). Une propriété peut être en lecture seule (pas de mot-clé **write** ni de méthode d'écriture dans la déclaration de la propriété) et dans ce cas elle peut être employée comme une *constante* du type de la propriété.
L'application doit manipuler une télévision : il nous faut donc un objet de classe "Television". Le plus simple, et le mieux pour vous faire manipuler des notions encore toutes chaudes et de rajouter un champ à la classe qui définit la fiche. Ajoutons donc un champ de type (classe) "Television" dans la section privée de la classe en question. Afin de clarifier les choses, voici sa déclaration complète faisant apparaître le champ :

```
TfmPrinc = class(TForm)
   pnPoids: TPanel;
   lbPoids: TLabel;
   edPoids: TEdit;
   btModifPoids: TButton;
   btQuitter: TButton;
 private
   { Déclarations privées }
   fTele1: Television;
 public
   { Déclarations publiques }
 end;
```

Remarquez dans le code ci-dessus le "f" qui préfixe le nom du champ. Nous avons déclaré un champ de type objet dans une classe, c'est la première fois mais cela n'a rien de très original et ne constitue pas une révolution puisqu'un champ peut être de n'importe quel type autorisé pour une déclaration de variable.
Afin de respecter le « cycle de vie » de l'objet "fTele1", il nous faut le construire puis l'utiliser puis le détruire. La construction peut se faire au moment de la construction de la fiche et la destruction au moment de sa destruction. Notez qu'il serait possible de surcharger le constructeur et le destructeur de la classe pour inclure ces deux instructions, mais Delphi fournit un moyen plus simple que nous avons employé jusqu'à présent et qui épargne beaucoup de soucis : les événements. Les deux qui nous intéressent sont les événements OnCreate et OnDestroy de la fiche. OnCreate se produit pendant la création de la fiche, tandis que OnDestroy se produit lors de sa destruction. Ce sont deux endroits idéaux pour construire/détruire des champs de type objet. Générez les deux procédures de réponse à ces événements et complètez le code en construisant fTele1 dans OnCreate et en le détruisant dans OnDestroy. Voici ce qu'il faut écrire :

```
procedure TfmPrinc.FormCreate(Sender: TObject);
begin
  fTele1 := Television.Create;
end;

procedure TfmPrinc.FormDestroy(Sender: TObject);
begin
  fTele1.Destroy;
end;
```

C'est le moment idéal pour vous parler d'une petite astuce sur laquelle insiste lourdement l'aide de Delphi : la méthode Free. Lorsque vous appelez Destroy sur un objet qui n'a pas été construit, il se produit une erreur fâcheuse qui ne devrait pas effrayer le programmeur consciencieux (et habitué !) car il vérifie toujours ce genre de chose. Bref, ce genre d'erreur n'est pas agréable et il existe un moyen simple de s'en protéger : appeler, à la place de Destroy,


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

la méthode Free. Cette méthode effectue un test de validité sur l'objet avant de le détruire. Voici la modification a effectuer :

```
procedure TfmPrinc.FormDestroy(Sender: TObject);
begin
  fTele1.Free;
end;
```

Nous allons maintenant faire fonctionner l'essentiel de l'application exemple : le bouton "Modifier Poids". Le principe de son fonctionnement a été détaillé plus haut, voici donc le code source qui illustre comment on utilise une propriété tant en lecture qu'en écriture.

```
procedure TfmPrinc.btModifPoidsClick(Sender: TObject);
var
  v, e: integer;
begin
  val(edPoids.Text, v, e);
  if e = 0 then
    begin
      fTele1.Poids := v;
      if fTele1.Poids <> v then
        // valeur refusée
        edPoids.Text := IntToStr(fTele1.Poids);
    end
  else
    edPoids.Text := IntToStr(fTele1.Poids);
end;
```

Comme vous pouvez le voir, un essai de traduction du contenu de la zone d'édition est réalisé grâce à la procédure "val". Ensuite, suivant le résultat de la conversion, la propriété est modifiée et sa valeur après tentative de modification est testée ou la valeur actuelle est replacée dans la zone d'édition. Voici une capture d'écran montrant l'exécution de l'application, dont vous pouvez obtenir le **code source ici** :

Ce sera tout pour cete petite application ; vous pourrez l'améliorer à souhait lorsque vous aurez complété l'exercice suivant :

Exercice 1 : (voir la **solution**).
Transformez la classe "AppareilAEcran" pour que la longueur de diagonale et l'état allumé/éteint soient des propriétés. En ce qui concerne l'état éteint/allumé, l'accesseur en écriture fera appel à eteindre et à allumer qui deviendront des méthodes privées. Vous ferez particulièrement attention à l'endroit où vous déclarez et placez les propriétés et les méthodes dans la déclaration de la classe "AppareilAEcran".


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Continuons sur les propriétés. Nous venons de voir qu'il est possible de déléguer la lecture et l'écriture de la valeur d'une propriété chacune à une méthode. Cette manière de faire, qui est courante, a cependant un inconvénient : le fait de devoir écrire parfois une méthode de lecture ou d'écriture rien moins que triviale, à savoir l'affectation de la valeur d'un champ à Result dans le cas de la lecture et l'affectation d'un champ dans le cas d'une écriture. Il est possible, pour ces cas particuliers où l'une ou l'autre des méthodes (voire les deux, dans certains cas très précis) n'est pas très originale, de s'en passer purement et simplement.
Le principe est de dire qu'au lieu d'appeler une méthode pour réaliser une des deux opérations de lecture ou d'écriture, on fait directement cette opération sur un champ qu'on spécifie en lieu et place du nom de méthode. On utilise la plupart du temps cette technique pour permettre la lecture directe de la valeur d'un champ tandis que son écriture est « filtrée » pa une méthode d'écriture. Voici un petit exemple reprenant la classe "TAppareilAEcran", où la propriété "Poids" a été modifiée pour ne plus utiliser de méthode de lecture mais le champ "fPoids" à la place (uniquement en lecture, puisque l'écriture passe toujours par la méthode "setPoids") :

```
AppareilAEcran = class
 private
   fPoids: integer;
   fLongueur_Diagonale: integer;
   fAllume: boolean;
   //function getPoids: integer;
   procedure setPoids(valeur: integer);
   ...
 public
   ...
   destructor Destroy; override;
   property Poids: integer
     read fPoids
     write setPoids;
   ...
 end;
```

Si vous modifiez l'unité obtenue à l'issue de l'exercice 1, vous constaterez que la modification ne gène pas le compilateur, à condition toutefois de supprimer ou de mettre en commentaire le code source de la méthode "getPoids". En effet, n'étant plus déclarée, cette méthode ne doit plus non plus être implémentée.
Il est également possible de créer une propriété en lecture seule ou en écriture seule (quoique ce cas-là soit d'un intérêt plus que douteux). Nous allons uniquement parler des propriétés en lecture seule. Certaines valeurs, à ne donner qu'à titre de renseignement, ne doivent pas pouvoir être modifiées. Ainsi, une propriété en lecture seule est le meilleur moyen de permettre la lecture tout en rendant l'écriture impossible.
Pour créer une propriété en lecture seule, il suffit de ne pas indiquer le mot clé **write** ainsi que la méthode ou le champ utilisé pour l'écriture. En ce qui concerne la lecture, vous pouvez aussi bien spécifier un champ qu'une méthode de lecture. Voici la déclaration de deux propriétés en lecture seule permettant d'obtenir, pour un objet de classe "EcranOrdinateur", la résolution horizontale ou verticale actuelle. Comme la résolution ne peut être changée qu'en modifiant à la fois les deux dimensions, on peut lire ces deux dimensions individuellement mais pas les modifier.

```
EcranOrdinateur = class(AppareilAEcran)
 private
   resolutions_possibles: array of TResolution;
   resolution_actuelle: TResolution;
   function changer_resolution(NouvRes: TResolution): boolean;
   function getResolutionHorizontale: word;
   function getResolutionVerticale: word;
 public
   property ResolutionHorizontale: word
     read getResolutionHorizontale;
   property ResolutionVerticale: word
     read getResolutionVerticale;
 end;
```

Comme vous le voyez, il n'y a rien de très transcendant là-dedans. Les propriétés en lecture seule sont assez peu courantes, mais se révèlent fort pratiques à l'occasion pour remplacer avantageusement une fonction de lecture. L'avantage tient en deux mots : l'inspecteur d'objets. Ce dernier sait, depuis la version 5 de Delphi, afficher les propriétés en lecture seule. Comme leur nom l'indique, il ne permet évidemment pas de modifier leur valeur.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XVI-C-6-b - Propriétés tableaux

Pour l'instant, nous n'avons considéré que des propriétés de type simple : des entiers, des chaînes de caractères, et pourquoi pas, même, des tableaux. Rien n'empêche en effet à une propriété d'être de type tableau : l'élément lu et écrit lors d'un accès à la propriété est alors un tableau complet. Rien ne permettait jusqu'à présent de ne manipuler plus d'un élément par propriété, et c'est là qu'un nouveau concept va nous permettre de nous sortir de cette impasse : les propriétés tableaux.
Ces propriétés tableaux, qui tirent leur nom non pas du fait qu'elles ont quelque chose à voir avec les tableaux, mais du fait de la manière de les utiliser (nous verrons cela plus loin) permettent de manipuler un nombre à priori quelconque de couples « clé-valeur ». Si vous ignorez ce que signifie ce terme, prenons deux petits exemples qui devraient suffire à vous faire comprendre de quoi je parle.
Premier exemple : imaginons un tableau de 10 cases indicées de 1 à 10, contenant des chaînes de caractères. Dans ce cas, ce tableau possède 10 couples clé-valeur où les clés sont les indices de 1 à 10, et les valeurs les chaînes de caractères stockées dans le tableau. La valeur de la clé i est la chaîne stockée dans la case n°i.
Second exemple plus subtil : prenons un enregistrement (**record**) "TPersonne" comprenant le nom, le prénom et l'âge d'une personne. Cet enregistrement comporte alors 3 couples clé-valeur où les 3 clés sont le nom, le prénom et l'âge. Les trois valeurs correspondantes sont les valeurs associées à ces trois clés. Vous saisissez le principe ?
Delphi permet, via les propriétés tableau, de gèrer une liste de couples clés-valeurs où les clés sont toutes d'un même type et où les valeurs sont également toutes d'un même type (ce qui peut être contourné et l'est fréquemment par l'utilisation d'objets, mais nous y reviendrons lors du paragraphe sur le polymorphisme). Le cas le plus fréquent est d'utiliser des clés sous forme de nombres entiers. Dans un instant, nous allons déclarer une propriété tableau permettant d'accèder aux chaînes mémorisées dans un objet de classe "Television". Mais voici tout d'abord le format de déclaration d'une propriété tableau

```
property nom_de_propriete[declaration_de_cle]: type_des_valeurs
  read methode_de_lecture
  write methode_d_ecriture;
```

Vous noterez ici que ce qui suit **read** et **write** ne peut PAS être un champ, contrairement aux propriétés classiques. Vous noterez en outre que la partie **write** est facultative.
La partie *nom_de_propriete* est le nom que vous voulez donner à la propriété. C'est, comme pour une propriété non tableau un identificateur. La partie *type_des_valeurs* donne le type d'une valeur individuelle, à savoir le type de la valeur dans chaque couple clé-valeur. Ainsi, si vous utilisez des couples indexant des chaînes de caractères par des entiers, c'est le type **string** qui devra être utilisé. Mais la partie intéressante est certainement celle qui figure entre les crochets (qui doivent être présents dans le code source) : elle déclare le type de la clé utilisée dans les couples clé-valeur. Contrairement à la valeur qui doit être un type unique, la clé peut comporter plusieurs éléments, ce qui permet par exemple d'indexer des couleurs par deux coordonnées X et Y entières afin d'obtenir la couleur un pixel aux coordonnées (X, Y) d'une image. Le plus souvent, on n'utilise qu'une seule valeur, souvent de type entier, mais rien n'empêche d'utiliser une clé farfelue définie par un triplet (trois éléments), par exemple un entier, une chaîne et un objet ! La déclaration d'une clé ressemble à la déclaration des paramètres d'une fonction/procédure :
*nom_de_parametre1: type1[; nom_de_parametre2: type2][; ...]*
(notez ici les crochets italiques qui dénotent des parties facultatives : ces crochets n'ont pas à être placés dans le code source)
Venons-en à un exemple : déclarons une propriété tableau permettant l'accès aux chaînes mémorisées par une télévision. La clé est dans notre cas un entier, et la valeur est également, ce qui est un cas particulier, un entier. Voici la déclaration de la propriété. Nous reviendrons plus loin sur les méthodes de lecture et d'écriture nécessaires :

```
 Television = class(AppareilAEcran)
  private
    ...
  public
    constructor Create; override;
    property Chaines[numero: integer]: integer
      read GetChaines
      write SetChaines;
  end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La seule nouveauté par rapport aux propriétés classiques est la présence d'une déclaration supplémentaire entre crochets, qui fait de cette propriété une propriété tableau. Comme vous le voyez, la propriété s'appuie sur deux accesseurs qui sont obligatoirement des méthodes. Les types de paramètres et l'éventuel type de résultat de ces deux méthodes est imposé par la déclaration de la propriété. Ainsi, la méthode de lecture doit être une fonction qui admet les paramètres définissant la clé, et retournant un résultat du type de la valeur de la propriété. La méthode d'écriture doit être une procédure dont les arguments doivent être ceux de la clé, auquel on ajoute un paramètre du type de la valeur. Ceci permet, lors de la lecture, à la propriété d'appeler la méthode de lecture avec la clé fournie et de retourner la valeur retournée par cette méthode, et à l'écriture, de fournir en paramètre la clé ainsi que la valeur associée à cette clé.
Voici les déclarations des deux accesseurs de la propriété "Chaines" : Voici les déclarations des deux accesseurs de la propriété "Chaines" :

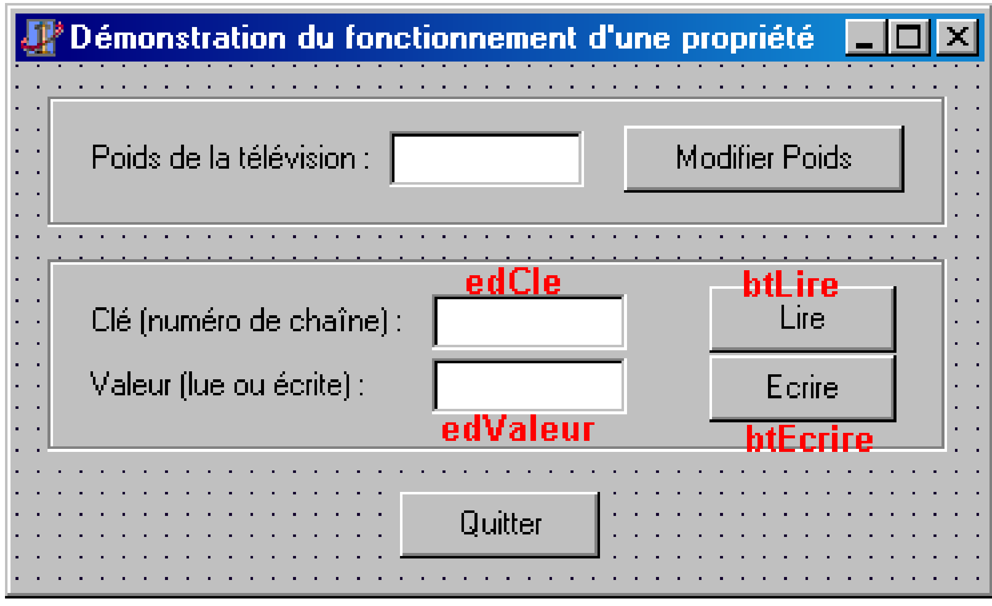
```
  ...
    procedure memoriser_chaine (numero, valeur: integer);
    function GetChaines(numero: integer): integer;
    procedure SetChaines(numero: integer; valeur: integer);
  public
    ...
```

Comme vous pouvez le voir, la méthode de lecture "getChaines" prend les paramètres de la clé, à savoir un entier, et retourne le type de la propriété, à savoir également un entier. La méthode d'écriture "setChaines", quant à elle, prend deux arguments : le premier est celui défini par la clé, et le second est le type de la propriété.
Après la déclaration, passons à l'implémentation de ces méthodes. On rentre ici dans un domaine plus classique où le fait d'être en train d'écrire une méthode de lecture ou d'écriture ne doit cependant pas être pris à la légère. Ainsi, la méthode se doit de réagir dans toutes les situations, c'est-à-dire quelle que soit la clé et la valeur (dans le cas de la méthode d'écriture), même si elles sont incorrectes. La méthode de lecture doit être implémentée de telle sorte qu'en cas de clé valide (c'est à vous de décider quelle clé est valide et quelle autre ne l'est pas) la fonction retourne la valeur qui lui correspond. De même, la méthode d'écriture, si elle existe (si la propriété n'est pas en lecture seule) doit, si la clé et la valeur sont correctes, effectuer les modifications qui s'imposent pour mémoriser le couple en question. Il appartient aux deux méthodes d'effectuer toutes les allocations éventuelles de mémoire lorsque c'est nécessaire : on voit ici l'immense avantage des propriétés qui permettent de lire et d'écrire des valeurs sans se soucier de la manière dont elles sont stockées et représentées.
Voici le code source des deux méthodes "getChaines" et "setChaines" :

```
function Television.GetChaines(numero: integer): integer;
begin
  if (numero >= 1) and (numero <= MAX_CHAINES) then
    Result := chaines_memorisees[numero]
  else
    // c'est une erreur, on peut choisir -1 comme valeur retournée en cas d'erreur
    Result := -1;
end;

procedure Television.SetChaines(numero: integer; valeur: integer);
begin
  if (numero >= 1) and (numero <= MAX_CHAINES) then
    if valeur >= 0 then
      chaines_memorisees[numero] := valeur;
  // sinon rien du tout !
end;
```

Vous pouvez constater qu'on n'autorise plus les valeurs de réglages négatives. C'est un choix arbitraire que nous pouvons faire facilement dans le cas présent mais qui devrait donner lieu à plus de réflexion dans un cas réel. Le fait d'interdire les valeurs négatives permet de distinguer la valeur -1 des autres valeurs, en lui donnant la signification d'erreur. Réalisons maintenant un petit exemple qui nous permettra d'utiliser cette propriété tableau.
Partez de l'application exemple développée pour tester la propriété "Poids" (**téléchargez-là ici** si vous ne l'avez pas sous la main. Attention : cette version inclue les messages indiquant des accès en lecture et en écriture pour la propriété Poids). Améliorez l'interface en la faisant ressembler à ceci :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




Le principe de la partie ajoutée sera le suivant : lors d'un clic sur le bouton "Lire", la valeur de la clé dont le numéro est indiqué dans la zone d'édition "clé" est lue, et écrite dans la zone d'édition "valeur". En cas d'erreur (lecture d'une valeur pour une clé qui n'existe pas, un message d'erreur est affiché. Lorsqu'un clic sera effectué sur le bouton "Ecrire", la valeur dans la zone d'édition "valeur" est affectée à la propriété "Chaines" avec la clé indiquée dans la zone d'édition "clé". En cas d'erreur de clé, un message d'erreur est affiché ; en cas d'erreur de valeur, l'ancienne valeur est rétablie dans la zone d'édition.
Une propriété tableau s'utilise comme une propriété, à savoir qu'on peut l'utiliser comme une constante lorsqu'elle est en lecture seule et comme une variable lorsqu'elle est en lecture/écriture. Une propriété tableau doit cependant être utilisée en spécifiant la clé à laquelle on désire accéder. Ainsi, la propriété doit être suivie d'une paire de crochets à l'intérieur desquels on place la valeur de la clé. Ainsi, par exemple, « Chaines[1] » fera référence à la valeur associée à la clé 1. Comme beaucoup de propriétés sont indexées (la clé est parfois appelée l'index de la propriété) par un simple entier partant de 0, l'utilisation d'une propriété tableau ressemble souvent à l'utilisation d'un simple tableau avec cependant la restriction des indiecs en moins puisque tous les indices sont utilisables (même s'il n'y a pas de valeur correspondante).
Dans notre cas, si on considère un objet "Tele1" de classe "Television", "Tele1.Chaines[1]" fait référence à la chaîne dont le numéro est 1. Le type de cette expression est celui de la valeur de la propriété tableau, à savoir un entier. Comme la propriété est en lecture/écriture, cette expression peut être utilisée comme une variable, à l'exception près que lors d'une lecture, la valeur -1 signale que la valeur n'existe pas (clé incorrecte). Voici le code source de la méthode associée au clic sur le bouton "Lire" :

```
procedure TfmPrinc.btLireClick(Sender: TObject);
var
  v, e, valeur: integer;
begin
  val(edCle.Text, v, e);
  if e = 0 then
    begin
      valeur := fTele1.Chaines[v];
      if valeur <> -1 then
        edValeur.Text := IntToStr(valeur)
      else
        ShowMessage('La clé n°' + IntToStr(v) + ' n''existe pas.');
    end
  else
    ShowMessage('La clé doit être une valeur entière.');
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Remarquez l'endroit surligné où l'on accède à la propriété "Chaines". Vous voyez qu'on utilise la prorpiété comme un tableau (c'est pour cela qu'on appelle ces propriétés des propriétés tableaux). Voici, pour enfoncer le clou et vous montrer que les propriétés tableaux sont simples à utiliser, le code source de la méthode réalisant l'écriture :

```
procedure TfmPrinc.btEcrireClick(Sender: TObject);
var
  cle, e1, valeur, e2: integer;
begin
  val(edCle.Text, cle, e1);
  val(edValeur.Text, valeur, e2);
  if (e1 = 0) and (e2 = 0) then
    begin
      // écriture
      fTele1.Chaines[cle] := valeur;
      // vérification
      valeur := fTele1.Chaines[cle];
      if (valeur <> -1) then
        edValeur.Text := IntToStr(valeur)
      else
        ShowMessage('La clé n°' + IntToStr(cle) + ' n''existe pas.');
    end
  else if (e2 <> 0) then
    ShowMessage('La valeur doit être une valeur entière.')
  else // e1 <> 0
    ShowMessage('La clé doit être une valeur entière.');
end;
```

Les lignes surlignées sont les seules qui nous intéressent ici. La première effectue, et c'est nouveau, une écriture de la propriété Chaines avec une clé donnée justement par la variable "cle". La clé et valeur sont obtenues par conversion des contenus des deux zones d'édition correspondantes, puis il y a tentative d'écriture de la propriété. Afin de vérifier si l'indice utilisé était correct, on effectue une lecture. Dans le cas où cette lecture retourne un résultat valide, on l'affiche dans la zone d'édition (ce qui en fait, si on y réfléchit, ne sert à rien d'autre qu'à vous donner un exemple). En cas d'invalidité, un message d'erreur est affiché.
Ce sera tout pour cet exemple. Nous venons de voir comment utiliser une propriété tableau simple, puisque les types de la clé et de la valeur de la propriété sont des entiers. Nous allons maintenant voir une notion sur les propriétés puis revenir aux exemples sur les propriétés tableau car la notion que nous allons voir est directement liée aux propriétés tableau.

## XVI-C-6-c - Propriété tableau par défaut

Lorsqu'un objet est créé majoritairement pour faire fonctionner une propriété tableau (nous verrons un exemple tout de suite après l'explication), c'est-à-dire quand la propriété tableau est incontesttablement le membre le plus important d'une classe, il peut être intéressant de simplifier son utilisation en en faisant une propriété tableau *par défaut*. Ce terme signifie que la manière de faire référence à cette propriété va être simplifiée dans le code source, cela d'une manière que nous allons voir dans un instant. Une seule propriété tableau peut être définie par défaut, et on peut alors y avoir accès sans mentionner le nom de la propriété : l'objet est simplement suffixé par la clé de la propriété tableau par défaut, et c'est à cette propriété qu'on s'adresse. Voici ce qu'il faut écrire sans propriété tableau par défaut :

```
fTele1.Chaines[cle] := valeur;
```

et avec :

```
fTele1[cle] := valeur;
```

Reconnaissez que c'est mieux, et pas très compliqué. Si vous croyez que c'est compliqué, pensez à une ligne de code du style :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
ListBox1.Items[1] := 'test';
```

Dans cette ligne de code, Items est un objet de classe TStrings, qui comporte une propriété par défaut, que vous utilisez puisque vous y faites référence avec la clé "1". Son nom, vous ne le connaissez pas, c'est "Strings". Votre ignorance de ce nom ne vous a pas empêché d'utiliser la propriété en question de manière implicite. Reste à savoir comment faire d'une propriété tableau une propriété tableau par défaut. C'est on ne peut plus simple et ça tient en un mot. Voici :

```
 ...
    constructor Create; override;
    property Chaines[numero: integer]: integer
      read GetChaines
      write SetChaines; default;
  end;
  ...
```

Le mot-clé **default** précédé et suivi d'un point-virgule (c'est important, car lorsqu'il n'est pas précédé d'un point-virgule, il a une autre signification) signale qu'une propriété tableau devient la propriété tableau par défaut. Il est bien évident que la propriété que vous déclarez par défaut doit être une propriété tableau, vous vous attirerez les foudres du compilateur dans le cas contraire. Mais voyons plutôt ce que cela change par exemple dans la méthode d'écriture de notre petite application exemple. Voyez les endroits surlignés pour les changements :

```
procedure TfmPrinc.btEcrireClick(Sender: TObject);
var
  cle, e1, valeur, e2: integer;
begin
  val(edCle.Text, cle, e1);
  val(edValeur.Text, valeur, e2);
  if (e1 = 0) and (e2 = 0) then
    begin
      // écriture
      fTele1[cle] := valeur;
      // vérification
      valeur := fTele1[cle];
      if (valeur <> -1) then
        edValeur.Text := IntToStr(valeur)
      else
        ShowMessage('La clé n°' + IntToStr(cle) + ' n''existe pas.');
    end
  else if (e2 <> 0) then
    ShowMessage('La valeur doit être une valeur entière.')
  else // e1 <> 0
    ShowMessage('La clé doit être une valeur entière.');
end;
```

Comme vous pouvez le constater, l'utilisation judicieuse de cette fonctionnalité n'est pas très coûteuse en temps de programmation (c'est le moins qu'on puisse dire), mais surtout en fait ensuite gagner non seulement en concision mais également en clarté si la propriété est bien choisie. Pour l'exemple ci-dessus, c'est discutable, mais pour ce qui est de l'exemple ci-dessous, c'est indiscutable.
Supposons un instant que nous ne connaissions pas les propriétés tableau par défaut. Le fait de modifier l'élément n°3 d'une liste défilante (ListBox) s'écrirait ainsi :

```
ListBox1.Items.Strings[2] := 'test';
```

C'est, comment dire... un petit peu lourd. Le fait que "Strings" soit une propriété par défaut est très justifié puisque l'objet "Items" (qui est en fait lui-même une propriété de la classe "TListBox" de classe "TStrings") ne sert qu'à manipuler des chaînes de caractères. AInsi, on met en avant, en faisant de "Strings" une propriété tableau par défaut, l'accès en lecture ou en écriture à ces chaînes de caractères, tout en laissant le reste utilisable classiquement.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Puisque "Strings" est par défaut, voici évidemment ce qu'on utilisera systématiquement (il ne faudra surtout pas s'en priver puisque les deux écritures sont équivalentes) :

```
ListBox1.Items[2] := 'test';
```

Vous en savez maintenant assez sur les propriétés et sur la programmation objet sous Delphi pour être capable de mener à bien un mini-projet. Dans ce mini-projet, vous aurez à manipuler nombre de connaissances apprises au cours des paragraphes précédents.

### XVI-C-7 - Mini-projet n°5 : Tableaux associatifs

Certains langages de programmation comme PHP proposent un type de données particulièrement intéressant qui n'a aucun équivalent en Pascal Objet. Ce genre de tableau utilise en fait n'importe quel indice pour contenir n'importe quelle donnée. Ainsi, on peut trouver dans un tableau associatif une chaîne indexée par un entier et un enregistrement indexé par une chaîne. La taille de ce type de tableau est en outre variable, et dépend des éléments ajoutés et retirés du tableau associatif.
Ce mini-projet consiste à réaliser une classe permettant l'utilisation d'un tableau associatif simple indexé uniquement par des chaînes de caractères et dont chaque élément est également une chaîne de caractères. Pour ce qui est du stockage d'un nombre quelconque d'éléments, vous avez le droit d'utiliser la classe "TList" qui permet de sauvegarder un nombre variable de pointeurs. L'accès aux éléments du tableau associatif doit se faire via une propriété tableau par défaut, pour un maximum de simplicité. Ainsi, les instructions suivantes, où "TA" est un tableau associatif, se doivent de fonctionner :

```
TA['prénom'] := 'Jacques';
TA['nom'] := 'Dupont';
TA['age'] := '53';
ShowMessage(TA['prénom'] + ' ' + TA['nom'] + ' a ' + TA['age'] + ' ans.');
```

Vous pouvez vous lancer seul sans aide mais je vous recommande de suivre les **étapes de progression**, qui vous donnent également les clés pour vous servir des éléments que vous ne connaissez pas encore (comme l'utilisation de la classe "TList"). Vous pourrez également y télécharger le mini-proje résolu.

## XVI-D - Notions avancées de programmation objet

La plupart des gens se contenteront sans sourciller des connaissances abordées dans la partie précédentes. Sachez que c'est largement insuffisant pour prétendre connaître et encore moins maîtriser la programmation objet. Dans cette partie, nous allons aborder quelques notions plus ardues comme l'héritage et le polymorphisme. Nous (re)parlerons également de notions très pratiques et plus accessibles comme la surcharge (plusieurs notions sont à mettre sous cette appelation générale), la redéfinition et les méthodes de classe. Essayez, même si vous aurez peut-être un peu plus de mal que pour la partie précédente, de suivre cette partie car elle contient les clés d'une bien meilleure compréhension de la programmation objet. Il est à noter que les interfaces et les références de classe ne sont pas au programme.

### XVI-D-1 - Retour sur l'héritage

Cela fait déjà un certain temps que je vous nargue avec ce terme d'héritage sans lui donner plus qu'une brêve définition. Le moment est venu d'aborder la chose.
Comme vous l'avez déjà vu, la programmation objet se base sur la réutilisation intelligente du code au travers des méthodes. A partir du moment où un membre est déclaré dans une classe, il est possible (sous certaines conditions) de le réutiliser dans les classes dérivées. Les conditions sont que le membre soit directement accessible (il doit donc être au plus "protégé" si on y accède depuis une autre unité) soit accessible via une méthode qui l'utilise et à laquelle on a accès. Le mécanisme qui vous permet cet accès, l'héritage, est intimement lié à la notion de *polymorphisme* que nous verrons plus loin. L'héritage permet d'augmenter les possibilité au fur et à mesure que l'on crée une hiérarchie de classes.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La hiérarchie en question, où une classe est une branche d'une classe noeud parent si elle est une sous-classe directe de celle-ci (si elle l'étend, donc), est justement appelé hiérarchie d'héritage des classes. Nous avons vu de petits extraits de telless hiérarchies au chapitre 10 mais les connaissances en programmation objet que je pouvais supposer acquises à l'époque ne me permettaient pas de trop m'étendre sur le sujet. L'héritage permet donc, à condition d'avoir accès à un membre, de l'utiliser comme s'il était déclaré dans la même classe. L'héritage rend possible la constitution étape par étape (en terme d'héritage) d'une vaste gamme d'objets aux comportements personnalisés mais possèdant cependant pour une grande part un tronc commun. Par exemple, tous les composants ont une base commune : la classe TComponent. Cette classe regroupe tous les comportements et les fonctionnalités minimums pour un composant. Il est possible de créer des composants ayant d'un seul coup toutes les fonctionnalités de TComponent grâce au mécanisme d'héritage.
L'héritage permet également une autre fonctionnalité qui pour l'instant est plutôt restée dans l'ombre : la possibilité d'avoir, au fur et à mesure qu'on avance dans une branche d'héritage (plusieurs classes descendantes les unes des autres de manière linéaire), une succession de méthodes portant le même nom mais effectuant des actions différentes. Selon la classe dans laquelle on se trouve, la méthode ne fait pas la même chose (en général, elle en fait de plus en plus au coours des héritages). C'est ici une autre notion qui entre en jeu : le polymorphisme.

## XVI-D-2 - Polymorphisme

Le polymorphisme est aux objets ce que le transtypage est aux autres types de données. C'est une manière un peu minimaliste de voir les choses, mais c'est une bonne entrée en matière. Si le polymorphisme est plus complexe que le transtypage, c'est que nos amies les classes fonctionnent sur le principe de l'héritage, et donc qu'en changeant la classe d'un objet, on peut peut-être utiliser des méthodes et des propriétés existant dans la classe d'origine, mais ayant un comportement différent dans la classe d'arrivée. Le polymorphisme est en fait le mécanisme qui permet à un objet de se comporter temporairement comme un objet d'une autre classe que sa classe d'origine.
Attention, ici, par classe d'origine, on ne veut pas dire la classe qui a servi à *déclarer* l'objet mais celle qui a servi à le *construire*. Jusqu'ici, les deux classes ont toujours été les mêmes, mais il est bon de savoir qu'elles peuvent tout à fait être différentes. En fait, on peut déclarer un objet d'une classe et le construire à partir d'une autre. Dans l'absolu, il faut se limiter à la classe servant à déclarer l'objet et ses sous-classes, faute de quoi les résultats sont imprévisibles.
Prenons de suite un exemple de code. N'essayez pas de le compiler dans l'état actuel :

```
procedure TfmPrinc.Button1Click(Sender: TObject);
var
  Tele: AppareilAEcran;
begin
  Tele := Television.Create;
  Tele.Chaines[1] := 1;
  Tele.Destroy;
end;
```

On a déclaré "Tele" comme étant un objet de classe "AppareilAEcran". En fait, "Tele" peut maintenant être soit un objet de classe "AppareilAEcran", soit être un objet polymorphe d'une classe descendante de "AppareilAEcran". La preuve en est faite par la première instruction qui crée un objet de classe "Television" et l'affecte à "Tele". "Tele" est donc déclaré comme étant un objet de classe "AppareilAEcran et se comportera comme tel en apparence, mais c'est bel et bien un objet de classe "Television". Si vous tentez de compiler le code source présent ci-dessus, le compilateur indique une erreur sur la seconde ligne. En effet, la classe "AppareilAEcran" ne comporte pas de propriété "Chaines". Cette propriété existe pourtant bien pour l'objet "Tele", mais du fait de la déclaration en "AppareilAEcran", la propriété est invisible. Vous venez de voir un premier aspect du polymorphisme.
Le polymorphisme permet en fait, à partir d'un objet déclaré d'une classe, d'y placer un objet de n'importe quelle classe descendante. Ce genre de comportement est très utile par exemple pour donner un objet en paramètre, lorsqu'on sait que l'objet pourra être d'une classe ou de ses sous-classes. C'est le cas d'un paramètre que vous avez rencontré si souvent que vous ne le voyez plus : le voici :

```
Sender: TObject
```

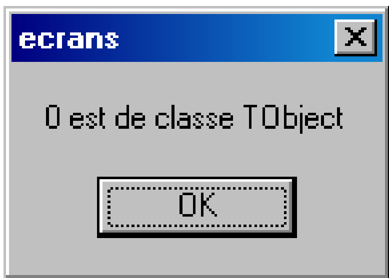
Ca vous dit quelque chose ? C'est tout simplement le paramètre donné à la quasi-totalité des méthodes de réponse aux événements. Ce paramètre qui ne nous a jamais encore servi est déclaré de classe TObject, mais du fait du

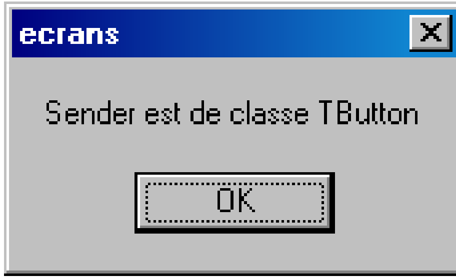

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

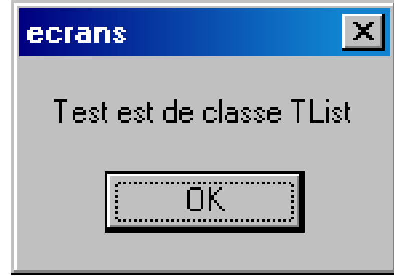
polymorphisme, peut être de n'importe quelle classe descendante de TObject, c'est-à-dire, si vous avez bien appris votre leçon, que ce paramètre peut être de n'importe quelle classe. Nous le voyons (dans la méthode à laquelle il est passé) d'une manière réduite puisque pour nous c'est un objet de classe "TObject", mais il faut savoir qu'en fait le "Sender" est rarement un objet de cette classe. Vrai ? Passons à un exemple :

```
procedure TfmPrinc.Button2Click(Sender: TObject);
var
  O, Test: TObject;
begin
  O := TObject.Create;
  Test := TList.Create;
  ShowMessage('O est de classe ' + O.ClassName);
  ShowMessage('Sender est de classe ' + Sender.ClassName);
  ShowMessage('Test est de classe ' + Test.ClassName);
  Test.Destroy;
  O.Destroy;
end;
```

le membre "ClassName" employé ci-dessus est une méthode de classe : nous y reviendrons, ce n'est pas là le plus important. Elle retourne le nom de la classe véritable de l'objet pour lequel on l'appelle. Ce qui est important, c'est le résultat de l'exécution de cette petite méthode :








Comme vous pouvez le constater, "ClassName" ne retourne pas les résultats auquel on pourrait s'attendre en n'y regardant pas de plus près et en ne connaissant pas le polymorphisme. "O" s'avère être, comme nous pouvions nous y attendre, un objet de classe "TObject". "Sender" est, lui, un objet de classe "TButton". Pour répondre à une question que vous vous posez sans doute, c'est l'objet correspondant au bouton sur lequel vous avez cliqué. Il est transmis sous la forme d'un objet de classe TObject mais c'est en fait un objet de classe "TButton". Enfin, "Test" est, comme sa construction le laisse présager, un objet de classe "TList". Bilan : trois objet apparamment de classe "TObject", dont l'un en est vraiment un, et dont deux autres qui sont en quelque sorte des usurpateurs.
Je pense que ces quelques exemples vous ont permi de cerner le premier principe du polymorphisme : sous l'apparence d'un objet d'une classe peut se cacher un objet d'une de ses classes descendantes.
Lorsqu'on a affaire à un objet polymorphe, on peut avoir besoin de lui redonner temporairement sa classe d'origine. Ainsi, il existe deux mots réservés de Pascal Objet qui vont nous permettre deux opérations basiques : tester si un objet est d'une classe donnée, et changer temporairement la classe d'un objet par polymorphisme. Ainsi, l'opérateur **is** permet de tester un objet pous savoir s'il est d'une classe donnée ou d'une de ses classes ancètres (donc en remontant l'arbre d'héritage et non plus en le descendant). Il s'utilise ainsi, en tant qu'expression booléenne (vraie ou fausse) :
*Objet* **is** *Classe*
Voici de suite un exemple démontrant son utilisation :

```
procedure TfmPrinc.Button2Click(Sender: TObject);
begin
  if Sender is TObject then
    ShowMessage('Sender est de classe TObject');
  if Sender is TButton then
    ShowMessage('Sender est de classe TButton');
  if Sender is TEdit then
    ShowMessage('Sender est de classe TEdit');
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Si vous exécutez cette méthode, vous verrez que "Sender" est considéré par **is** comme un objet de classe "TObject" et de classe "TButton", mais pas de classe "TEdit". En fait, puisque "Sender" est un objet dont la classe véritable est "TButton", il est valable de dire qu'il est de classe "TObject" ou de classe "TButton", mais il n'est certainement pas de classe "TEdit" puisque "TButton" n'est pas une classe descendante de "TEdit".
L'opérateur **is** est en fait pratique pour s'assurer qu'on peut effectuer un changement de classe (pas exactement un transtypage puisqu'on parle de polymorphisme) sur un objet. Ce changement de classe peut s'effectuer via l'opérateur **as**. Ainsi, l'expression :
*Objet* **as** *Classe*
peut être considéré comme un objet de classe "*Classe*". Si la classe n'est pas une classe ancêtre de la classe d'origine de l'objet, une exception se produit. Voici une petite démonstration de l'utilisation de **as** conjointement à **is** :

```
procedure TfmPrinc.Button2Click(Sender: TObject);
begin
  if Sender is TButton then
    (Sender as TButton).Caption := 'Cliqué !';
end;
```

La méthode ci-dessus teste d'abord la classe de "Sender". Si "TButton" est une classe ancêtre (ou la bonne classe) pour "Sender", il y a exécution de l'instruction de la deuxième ligne. Cette instruction se découpe ainsi :

1. "Sender **as** TButton" se comporte comme un objet de classe "TButton", ce qui va permettre l'accès à ses propriétés ;
2. "(Sender **as** TButton).Caption" s'adresse à la propriété "Caption" du bouton.
3. Enfin, l'instruction complète affecte une nouvelle valeur à la propriété.

Lorsque vous exécutez le code en question, en cliquant sur le bouton, vous aurez la surprise (?) de voir le texte du bouton changer lors de ce clic. Remarques que sans utiliser le "Sender", sr une fiche comportant plusieurs boutons, ce genre d'opération pourraît être complexe (étant entendu qu'on ne suppose pas connu le composant auquel la méthode est liée).

## XVI-D-3 - Surcharge et redéfinition des méthodes d'une classe

Nous avons déjà parlé de la possibilité, lors de la création d'une classe héritant d'une autre classe, de *surcharger* certaines de ses méthodes. La surcharge, signalée par le mot-clé **override** suivant la déclaration d'une méthode déjà existante dans la classe parente, n'est possible que sous certaines conditions :

- Vous devez avoir accès à la méthode surchargée. Ainsi, si vous êtes dans la même unité que la classe parente, vous pouvez ignorer cette restriction. Sinon, en dehors de la même unité, la méthode doit être au minimum protégée.
- La méthode de la classe parente doit être soit virtuelle (déclarée par la mot-clé **virtual**) soit déjà surchargée (**override**).

La surcharge possède également une propriété intéressante : elle permet **d'augmenter** la visibilité d'une méthode. Ainsi, si la méthode de la classe parente est protégée, il est possible de la surcharger dans la section protégée ou publique de la classe (la section publiée étant réservée aux propriétés). Depuis la même unité, une méthode surchargée présente dans la section privée de la classe parente peut être placée dans la section privée (pas de changement de visibilité), protégée ou publique (augmentation de visibilité). Il n'est pas possible de diminuer la visibilité d'une méthode surchargée. Cette fonctionnalité vous sera surtout utile au moment de la création de composants et pour rendre certaines méthodes, innaccessibles dans la classe parente, accessibles dans une ou plusieurs classes dérivées.
La surcharge au moyen du couple **virtual-override** permet ce qu'on appelle des appels virtuels de méthodes. En fait, lorsque vous appelez une méthode déclarée d'une classe et construire par une classe descendante de celle-ci, c'est la méthode surchargée dans cette dernière qui sera exécutée, et non la méthode de base de la classe de déclaration


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

de l'objet. Voyons de suite un exemple histoire de clarifier cela. Pour faire fonctionner l'exemple, vous devez déclarer une méthode publique virtuelle appelée "affichage" dans la classe "AppareilAEcran". Voici sa déclaration :

```
  ...
    destructor Destroy; override;
    function affichage: string; virtual;
    property Poids: integer
    ...
```

et son implémentation :

```
function AppareilAEcran.affichage: string;
begin
  Result := 'Appareil à écran';
end;
```

Déclarez maintenant une méthode "affichage" surchargant celle de "AppareilAEcran" dans la classe "Television". Voici sa déclaration :

```
  ...
    constructor Create; override;
    function affichage: string;
    property Chaines[numero: integer]: integer
    ...
```

et son implémentation (Notez ici l'emploi d'une forme étendue de **inherited** que nous n'avons pas encore vue : il s'agit simplement de préciser le nom de la méthode surchargée, ce qui est obligatoire dés que l'on sort de l'appel simple de cette méthode, sans paramètres personnalisés et sans récupération du résultat. Le fait de récupèrer le résultat de la méthode héritée nons oblige à utiliser la forme étendue et non la forme raccourcie) :

```
function Television.affichage: string;
begin
  Result := inherited affichage;
  Result := Result + ' de type Télévision';
end;
```

Entrez maintenant le code ci-dessous dans une méthode associée à un clic sur un bouton. Exécutez-là et regardez ce qu'elle vous annonce.

```
procedure TfmPrinc.Button2Click(Sender: TObject);
var
  Tele1: AppareilAEcran;
  Tele2: AppareilAEcran;
begin
  Tele1 := AppareilAEcran.Create;
  Tele2 := Television.Create;
  ShowMessage('Tele1.affichage : ' + Tele1.affichage);
  ShowMessage('Tele2.affichage : ' + Tele2.affichage);
  ShowMessage('(Tele2 as AppareilAEcran).affichage : ' + (Tele2 as Television).affichage);
  Tele2.Destroy;
  Tele1.Destroy;
end;
```

Le premier appel indique un "appareil à écran". C'est normal puisque nous avons déclaré l'objet et l'avons construit à partir de cette classe. Les deux autres appels, par contre, signalent un "appareil à écren de type télévision". Le résultat du troisième affichage ne devrait pas vous étonner puisque nous demandons explicitement un appel de la méthode "affichage" pour l'objet "Tele2" en tant qu'objet de classe "Television". Le second affichage donne le même résultat, et c'est là que vous voyez le résultat d'un appel virtuel de méthode : l'objet "Tele2", pourtant déclaré de


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

type "AppareilAEcran", lorsqu'on lui applique sa méthode "affichage", redonne bien l'affichage effectué par la classe "Television". C'est non seulement parce que nous avons créé l'objet avec la classe "Television" que cela se produit, mais aussi parce que les deux méthodes "affichage" des deux classes sont liées par une relation de surcharge.
Pour bien vous en rendre compte, supprimez le mot-clé **override** de la déclaration de "affichage" dans la classe "Television". Recompilez en ignorant l'avertissement du compilateur et relancez le test. Le premier affichage reste le même, puisque nous n'avons rien changé à ce niveau. Le second réserve la première surprise, puisqu'il signale maintenant un "appareil a écran" simple. En fait, nous avons appelé la méthode "affichage" dans un contexte où "Tele2" est de classe "AppareilAEcran". Comme le lien entre les deux méthodes "affichage" n'existe plus, il n'y a pas appel virtuel de la méthode "affichage" de "Television" mais seulement de "AppareilAEcran". Le troisième affichage est identique à celui de l'essai précédent. En effet, nous appelons cette fois-ci "affichage" dans un contexte où "Tele2" est de classe "Television". Cette méthode "affichage" fait appel à la méthode héritée, ce qui est en fait légitime même si elle ne la surcharge pas par **override**, puis complète le message obtenu par l'appel de la méthode héritée. C'est pour cela que le message est différent dans le deuxième et le troisème affichage.
Ce que nous venons de faire en supprimant le **override** de la déclaration de "affichage" n'est pas un abus mais une fonctionnalité de la programmation objet. Il s'agit non plus d'une surcharge mais d'une *redéfinition* de la méthode. Pascal Objet dispose même d'un mot-clé permettant de signaler cela et du même coup de supprimer le message d'avertissement du compilateur : **reintroduce**. Corrigez donc la déclaration de "affichage" dans "Television" comme ceci :

```
  ...
    constructor Create; override;
    function affichage: string; reintroduce;
    property Chaines[numero: integer]: integer
    ...
```

Recompilez et désormais le compilateur ne vous avertit plus. En effet, vous venez de lui dire : « OK, je déclare à nouveau dans une classe dérivée une méthode existante dans la classe parente, sans override, mais je ne veux pas la surcharger, je veux la redéfinir complètement ». Dans une méthode redéfinie, vous avez le droit d'appeler **inherited** mais ce n'est pas une obligation. Changez le code source de Television.affichage par ceci afin de vous en convaincre :

```
function Television.affichage: string;
begin
  Result := 'type Télévision';
end;
```

Le troisième affichage, lorsque vous relancez le test utilisé précédemment, signale bien un "type Télévision", ce qui prouve que les deux méthodes "affichage" sont maintenant indépendantes.

## XVI-D-4 - Méthodes abstraites

Les méthodes abstraites sont une notion très facile à mettre en oeuvre mais assez subtile à utiliser. Une méthode abstraite est une méthode virtuelle qui n'a pas d'implémentation. Elle doit pour cela être déclarée avec le mot réservé **abstract**. Le fait qu'une classe comporte une *classe abstraite* fait de cette classe une classe que dans d'autres langages on appelle classe abstraite mais qu'en Pascal Objet on n'appele pas. Une telle classe ne doit pas être instanciée, c'est-à-dire qu'on ne doit pas créer d'objets de cette classe.
L'utilité de ce genre de procédé ne saute pas immédiatement aux yeux, mais imaginez que vous deviez concevoir une classe qui serve uniquement de base à ses classes dérivées, sans avoir un quelconque intérêt en tant que classe isolée. Dans ce cas, vous devriez déclarer certaines méthodes comme abstraites et chaque classe dérivant directement de cette classe se devra d'implémenter la méthode afin de pouvoir être instanciée. C'est un bon moyen, en fait, pour que chaque classe descendante d'une même classe de base choisisse son implémentation indépendante des autres. Cette implémentation n'est pas une option, c'est une obligation pour que la classe dérivée puisse être instanciée.
Imaginez l'unité très rudimentaire suivante, où l'on définit une partie de trois classes :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
unit defformes;

interface

uses
  SysUtils;

type
  TForme = class
  private
    X, Y: integer;
    function AfficherPosition: string;
  end;

  TRectangle = class(TForme)
  private
    Larg, Haut: integer;
  public
    function decrire: string;
  end;

  TCercle = class(TForme)
  private
    Rayon: integer;
  public
    function decrire: string;
  end;

implementation

{ TForme }

function TForme.AfficherPosition: string;
begin
  Result := 'Position : ('+IntToStr(X)+' , '+IntToStr(Y)+')';
end;

{ TRectangle }

function TRectangle.decrire: string;
begin
  Result := AfficherPosition+', Hauteur : '+IntToStr(Haut)+', Largeur : '+IntToStr(Larg);
end;

{ TCercle }

function TCercle.decrire: string;
begin
  Result := AfficherPosition+', Rayon : '+IntToStr(Rayon);
end;

end.
```

Vous constatez que chaque classe descendante de "TForme" doit définir sa propre méthode "decrire", ce qui est ennuyeux car la description est au fond un procédé très standard : on décrit la position, puis les attributs supplémentaires. Seuls les attributs supplémentaires ont besoin d'être fournis pour qu'une description puisse être faite. Nous pourrions envisager une seconde solution, déjà préférable :

```
unit defformes;

interface

uses
  SysUtils;

type
  TForme = class
  private
    X, Y: integer;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    function AfficherPosition: string;
    function AfficherAttributsSupplementaires: string; virtual;
  public
    function decrire: string;
  end;

  TRectangle = class(TForme)
  private
    Larg, Haut: integer;
    function AfficherAttributsSupplementaires: string; override;
  public
  end;

  TCercle = class(TForme)
  private
    Rayon: integer;
    function AfficherAttributsSupplementaires: string; override;
  public
  end;

implementation

{ TForme }

function TForme.AfficherPosition: string;
begin
  Result := 'Position : ('+IntToStr(X)+' , '+IntToStr(Y)+')';
end;

function TForme.AfficherAttributsSupplementaires: string;
begin
  Result := '';
end;

function TForme.decrire: string;
begin
  Result := AfficherPosition + AfficherAttributsSupplementaires;
end;

{ TRectangle }

function TRectangle.AfficherAttributsSupplementaires: string;
begin
  Result := Inherited AfficherAttributsSupplementaires +
            ', Hauteur : '+IntToStr(Haut)+', Largeur : '+IntToStr(Larg);
end;

{ TCercle }

function TCercle.AfficherAttributsSupplementaires: string;
begin
  Result := Inherited AfficherAttributsSupplementaires +
            ', Rayon : '+IntToStr(Rayon);
end;

end.
```

Dans cette nouvelle version, vous voyez que le code servant à la description a été regroupé et qu'il n'est plus nécessaire d'en avoir plusieurs versions qui risqueraient de partir dans tous les sens. Au niveau de "TForme", la description n'affiche que la position puisque "AfficherAttributsSupplementaires" retourne une chaîne vide. Par contre, au niveau des deux autres classes, cette fonction a été surchargée et retourne une chaîne spécifique à chaque classe. La description au niveau de ces classes appelera ces versions surchargées et décrira donc complètement un rectangle ou un cercle.
Le problème, ici, c'est qu'une classe descendante de "TForme" n'a pas du tout l'obligation de s'acquitter de sa description. En effet, elle peut fort bien ne pas surcharger "AfficherAttributsSupplementaires" et cela ne constituera pas une erreur. En outre, le fait d'avoir une implémentation de "AfficherAttributsSupplementaires" dans "TForme" est parfaitement inutile. Nous allons donc la transformer en méthode abstraite, que les classes descendantes DEVRONT surcharger pour avoir la prétention d'être instanciées correctement (une instance d'une classe comportant une méthode abstraite non surchargée plantera si on a le malheur d'appeler la méthode en question !). Voici le code source


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

correspondant : notez que les **inherited** ont disparu et que l'implémentation de "AfficherAttributsSupplementaires" pour la classe "TForme" a également disparu :

```
unit defformes;

interface

uses
  SysUtils;

type
  TForme = class
  private
    X, Y: integer;
    function AfficherPosition: string;
    function AfficherAttributsSupplementaires: string; virtual; abstract;
  public
    function decrire: string;
  end;

  TRectangle = class(TForme)
  private
    Larg, Haut: integer;
    function AfficherAttributsSupplementaires: string; override;
  public
  end;

  TCercle = class(TForme)
  private
    Rayon: integer;
    function AfficherAttributsSupplementaires: string; override;
  public
  end;

implementation

{ TForme }

function TForme.AfficherPosition: string;
begin
  Result := 'Position : ('+IntToStr(X)+' , '+IntToStr(Y)+')';
end;

function TForme.decrire: string;
begin
  Result := AfficherPosition + AfficherAttributsSupplementaires;
end;

{ TRectangle }

function TRectangle.AfficherAttributsSupplementaires: string;
begin
  Result := ', Hauteur : '+IntToStr(Haut)+', Largeur : '+IntToStr(Larg);
end;

{ TCercle }

function TCercle.AfficherAttributsSupplementaires: string;
begin
  Result := ', Rayon : '+IntToStr(Rayon);
end;

end.
```

Cette version est de loin la meilleure. Nous avons cependant perdu une possibilité en gagnant sur la compacité du code source : il n'est plus envisageable d'instancier des objets de classe "TForme". Notez cependant que vous pouvez avoir des objets *déclarés* de classe "TForme" et construits (instanciés) par une des classes "TRectangle" ou


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

"TCercle". Appeler la méthode "AfficherAttributsSupplementaires" pour de tels objets N'EST PAS une erreur, comme en témoigne l'exemple suivant :

```
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TForme;
begin
  F := TRectangle.Create;
  ShowMessage(F.decrire);
  F.Destroy;
end;
```

dont le résultat est :

Une petite explication et j'arrête de vous embêter : dans le code ci-dessus, l'objet F est déclaré de classe "TForme", mais construit de classe "TRectangle". Lorsque vous appelez "F.decrire", c'est d'abord la méthode "decrire" de "TForme" qui est recherchée et trouvée. Cette méthode fait appel à une première méthode classique qui ne pose aucun problème. L'appel de la seconde méthode est un appel d'une méthode abstraite, donc un appel voué à l'échec. La méthode "AfficherAttributsSupplementaires" est cependant virtuelle et c'est là que tout se joue : l'appel virtuel appelle non pas la méthode "AfficherAttributsSupplementaires" de "TForme" mais celle de la classe la plus proche de la classe de construction de l'objet F (soit "TRectangle") qui surcharge "AfficherAttributsSupplementaires" sans la redéfinir, c'est-à-dire dans ce cas précis "TRectangle". L'appel virtuel de "decrire" fait donc appel à la méthode "AfficherPosition" de "TForme" dans un premier temps et à la méthode "AfficherAttributsSupplementaires" de "TRectangle" dans un second temps.
Ne vous inquiètez pas trop si vous n'avez pas tout saisi : les méthodes virtuelles sont peu utilisées. En règle générale, on ne les trouve que dans le code source de certains composants.

## XVI-D-5 - Méthodes de classe

Nous allons conclure ce chapitre en parlant quelques instants de méthodes très particulières que nous avons déjà eu l'occasion d'utiliser sans les expliquer. Ces méthodes sont appelées *méthodes de classe*. Pour l'instant, nous avons employé deux de ces méthodes : le constructeur, "Create", et "ClassName".
Une méthode de classe est une méthode qui peut s'utiliser dans le contexte d'un objet ou, et c'est nouveau, d'une classe. Par contexte de classe, je veux parler de la forme :

```
... := TRectangle.Create;
```

Nous avions dit jusqu'à présent qu'une méthode s'appliquait à un objet, et manifestement, "Create" est appliquée à une classe. Il y a manifestement un souci. "Create" est en fait une méthode de classe, c'est-à-dire que, comme toutes les méthodes de classe, elle peut s'appliquer à un objet de la classe où elle a été définie, mais son intérêt est de pouvoir s'appliquer à la classe elle-même.
Essayez le code source ci-dessous :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(TEdit.ClassName);
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;
```

qui doit afficher le message "TEdit". Comme vous pouvez le constater, nous n'avons pas construit d'instance de la classe "TEdit" mais nous avons cependant appelé sa méthode de classe ClassName qui retourne le nom de la classe, soit "TEdit".
En Pascal Objet, les méthodes de classe sont très rarement utilisées, mais je tenais simplement à vous montrer à quoi est due la forme bizarre que prend la construction d'un objet. A titre de curiosité, la déclaration de la méthode "ClassName" se fait dans la classe "TObject", et sa déclaration est :

```
  ...
    class function ClassName: ShortString;
    ...
```

Comme vous pouvez le constater, la déclaration d'une méthode de classe est précédée par le mot réservé **class**. Vous pourrez maintenant les repèrer si vous en rencontrer par hasard (et changer de trottoir rapidement ! ;-).

## XVI-E - Conclusion

Ce chapitre est maintenant terminé. L'écriture m'en aura pris en temps incroyable, et pourtant il ne contient pas toutes les notions que j'avais envisagé d'y placer au départ. Au lieu de faire un pavé monolithique, j'ai préféré placer dans ce chapitre les notions les plus importantes à connaître dans le domaine de la programmation orientée objets sous Delphi, et relèguer à plus tard les notions suivantes :

- Interfaces
- Références de classe
- Exceptions
- Création de composants

Ces notions de plus en plus importantes en programmation objet feront l'objet d'un autre chapitre.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XVII - Liaison DDE

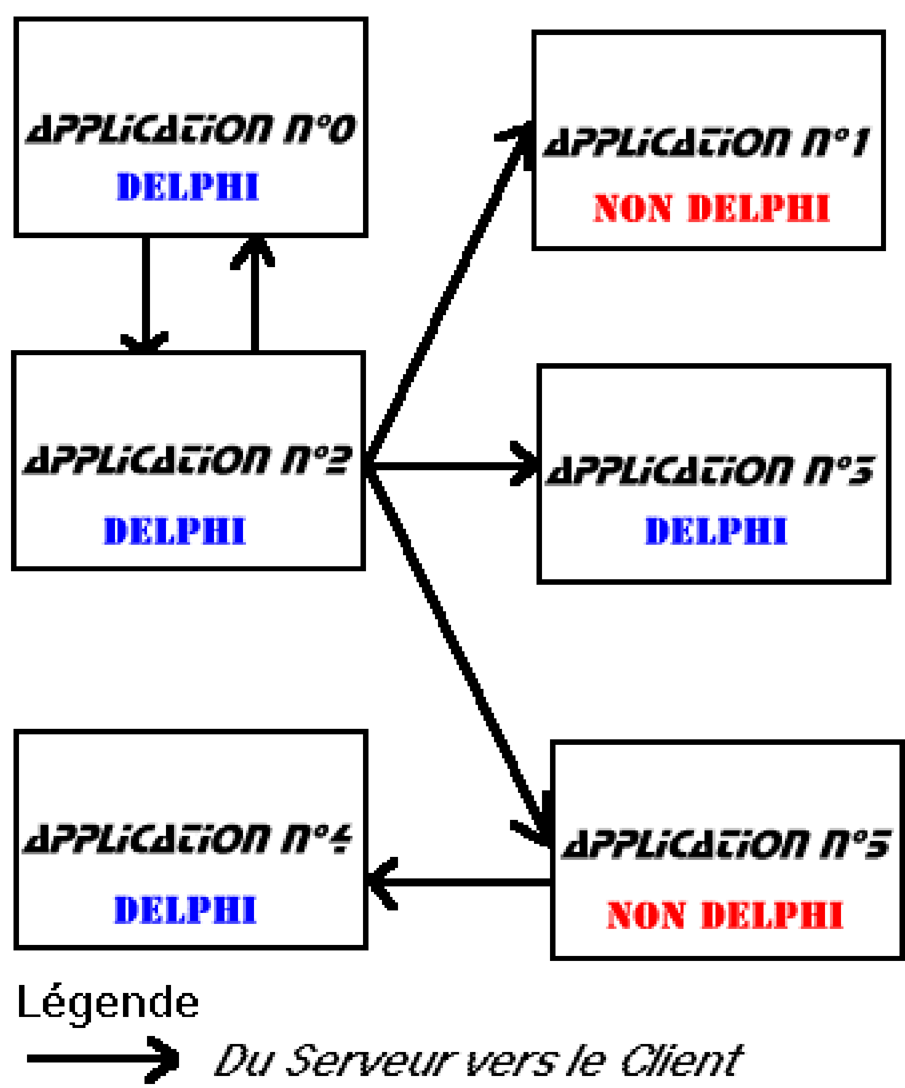
Le DDE (Dynamic Data Exchange soit en Français échange de donnée dynamique) permet à deux applications (voir plus) de communiquer entre elle. Notez qu'il ne faut pas le confondre avec OLE qui est une autre façon de procéder mais cela est une autre histoire. Mon but ici, n'est pas de vous montrer quelle utilisation vous pouvez en faire mais de vous apprendre à la mettre en oeuvre et d'apprendre quelques termes. Ce qui est relativement facile avec delphi.

## XVII-A - Présentation

### XVII-A-1 - Introduction

Le DDE permet à deux applications de communiquer entre elles. Les applications doivent être lancées. L'une des applications est alors serveur tandis que les autres sont clients. Le serveur peut aussi être le client d'autres applications. La façon dont communiquent les applications peut être de deux types. Soit les clients appellent le serveur à chaque fois qu'ils ont besoin d'une information, soit le serveur informe le client des nouvelles informations disponibles.

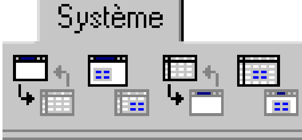

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




*Liaison DDE - Vue d'ensemble des liaisons possibles.*

## XVII-A-2 - Composants

Dans Delphi, vous trouverez les composants dans l'onglet Système. Ils sont au nombre de quatre et permettent de réaliser des applications utilisant le DDE.




Dans l'ordre, on trouve : DdeServerConv, DdeServerItem, DdeClientConv et DdeClientItem.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

### XVII-A-3 - Utilisation

Le DdeServerConv et le DDeClientConv permet d'établir la conversation entre un client et son serveur. Mais il faut en plus ajouter les DdeServerItem et DdeClientItem pour permettre l'échange de donnée. En fonction du nombre de donnée à échanger, on ajoute autant de DdeServerItem que de DdeCLientItem.
Les deux composants DdeServerItem et DdeClientItem proposent les propriétés suivantes : **Lines** de type **TStrings** et **Text** de type **TString**.
L'une ou l'autre des propriétés est à utiliser en fonction de la taille de la donnée à échanger (la propriété **Text** de type **TString** est limitée à 255 caractères).

## XVII-B - Partie Serveur

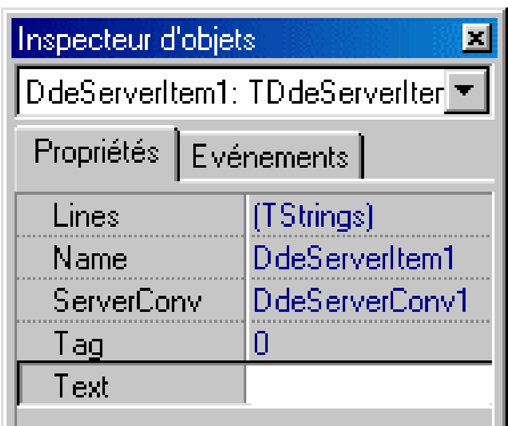
Nous allons commencer par la partie Serveur pour deux choses. La première, une fois le serveur réalisé, nous saurons quelle type de donnée, nous voulons échanger. La deuxième lors de l'exécution ou du débogage du client, le serveur peut être chargé automatiquement.
Dans votre projet, ajouter un composant DdeServerConv et autant de DdeServerItem que de donnée à échanger.

*Vous n'êtes pas obligé d'ajouter de Composant DdeServerConv, dans ce cas c'est la fiche contenant les DdeServerItem qui sert de serveur. Le problème est que c'est le titre de la fiche qui est utilisé pour identifier le serveur. Si par malheur vous changer le titre de la fiche, vous rompez la communication. Ce genre de gag, peut vous faire passer des nuits blanches à chercher l'origine du problème. Donc à moins que vous ne vouliez tenter le diable, utilisez toujours un DdeServerConv pour identifier le serveur.*

Pour mettre les nouvelles données dans les DdeServerItem, une simple affectation suffit.

```
procedure TForm1.Edit1OnChange(Sender : TObject);
begin
  DdeServerItem1.Text := Edit1.Text;
end;
```

Il ne reste plus qu'à changer les propriétés des DdeServerItem et la partie serveur sera terminée. Donc ne perdons pas un instant et modifions la propriété ServerConv, en indiquant soit le nom du composant DdeServerConv, soit le nom du projet sans l'extension.

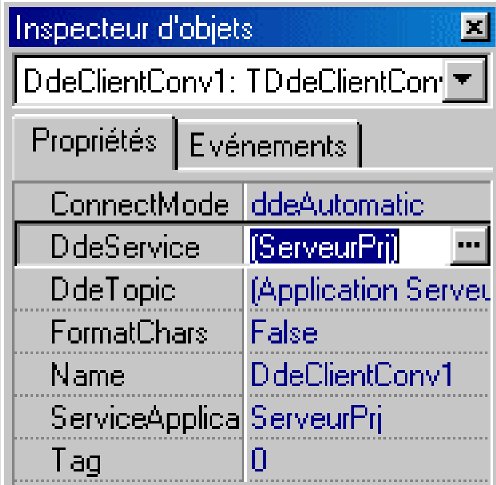



## XVII-C - Partie Client

Passons maintenant à la partie client. Ajoutez un composant DdeClientConv (obligatoire cette fois-ci) et autant de composant DdeClientItem que de DdeServerItem dont vous voulez récupérer la valeur.
Modifiez ensuite les propriétés du DdeClientConv.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

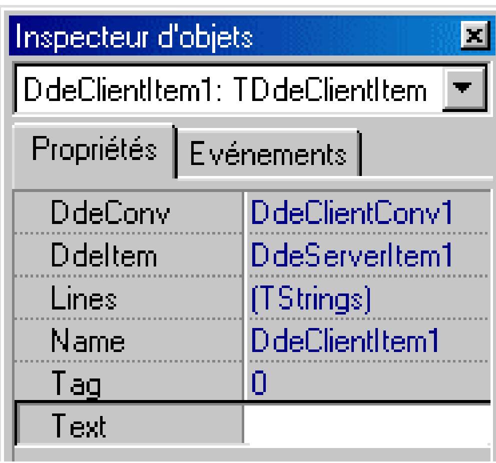
- Modifiez ensuite les propriétés du DdeClientConv.




- Positionnez la propriété ConnectMode à DdeManual.
- Double cliquez sur DdeService.

1. Dans la partie Service indiquer le nom de l'application serveur sans l'extension.
2. Dans la partie Rubrique indiquer le nom du composant DdeServerConv (ou le titre de la fiche si vous n'avez pas suivit mon conseil).

- Dans la rubrique Service application, indiquez de nouveau le nom de l'application serveur sans l'extension. Vous pouvez indiquer le chemin complet si elle n'est pas dans le même répertoire que l'application cliente
- Modifier ensuite les propriétés de Chaque DdeClientItem.





Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- Dans DdeConv indiquez le DdeServerConv ou le nom de la fiche serveur.
- Dans DdeItem indiquez le nom du composant DdeServerItem.
- RePositionner la propriété ConnectMode à DdeAutomatic du DdeClientConv.

Si vous avez correctement fait les choses, l'application serveur doit se charger.

```
procedureTForm1.DdeClientItem1Change(Sender: TObject);
begin
  // Récupère les informations du serveur et les placent dans Edit1
  Edit1.Text := DdeClientItem1.text;
end;
```

Lancez votre application cliente. Normalement l'application serveur est déjà chargée, si ce n'est pas le cas vérifier les propriétés du DdeClientConv.
Arrangez-vous pour voir les deux applications en même temps. Modifiez le texte dans l'application serveur, vous pouvez voir que les modifications sont répercutées dans l'application cliente.

## XVII-D - Cas particulier : Serveur Non-Delphi

Vous me direz c'est bien gentil mais que faire si le serveur n'est pas une application Delphi. Dans ce cas, il reste une solution à condition de 'Bien' connaître l'application serveur. En effet, pour pourvoir communiquer avec le serveur, vous devez :

- Connaître le nom du serveur. C'est la partie facile car il s'agit du nom de l'application sans l'extension.
- Connaître le nom du service. C'est là qu'il faut connaître l'application serveur.
- Connaître le sujet ou *topic*. Correspond aux données que vous voulez récupérer.

### XVII-D-1 - Serveur Non-Delphi

Exemple d'une application Cliente en Delphi communicant avec le gestionnaire de programme donc un programme Windows.
Dans votre application cliente, ajouter un DdeClientConv, DdeClientItem, une ListBox.
Initialiser les propriétés de DdeClientConv,

- à ddeManuel (permet de dire à Delphi d'attendre la fin des changements).
- Double cliquez sur DdeService et indiquez : Progman dans service et sujet.
- Dans service application entrez Progman
- Modifier aussi le composant DdeClientItem,
- Dans DdeConv, indiquez le nom du DdeClientConv,
- Dans DdeItem, indiquez Groups
- Remettez ConnectMode à DdeAutomatic (On a finit les modifications et Delphi peut appliquer les changements).

Dans l'événement OnChange du DdeClientItem, faire une boucle parcourant touts les éléments de DdeCLientItem.Lines et les ajouter dans la ListBox.

```
procedureTForm1.DdeClientItem1Change(Sender: TObject);
var
  i : integer;
begin
  // Récupère les informations du serveur et les placent dans ListBox1
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    for i := 0 to DdeClientItem1.Lines.Count - 1 do
    begin
      ListBox1.Lines.Add(DdeClientItem1.Lines[i]);
    end;
  end;
```

## XVII-D-2 - Client Non-Delphi

L'exemple suivant, utilise HTBasic. Je doute que beaucoup de monde l'utilise mais ainsi vous verrez que vos applications peuvent communiquer avec une application conçue dans n'importe quel autre langage de programmation du moment qu'il gère les liaisons DDE. Et puis il n'y a pas que Delphi et Visual C/C++ dans la vie.

```
INTEGER Result,Ch1,Ch2
DIM Ret$[256],Htbvers$[80]
Htbvers$=SYSTEM$("VERSION:HTB")
IF POS(Htbvers$,"9") THEN
  Htbvers$="9"
ELSE
  Htbvers$="5"
END IF
CLEAR SCREEN
! Charge la librairie DDE correspondant à la version du HTB, seulement si elle n'est pas déjà chargée.
IF NOT INMEM("Ddeinit") THEN
  SELECT Htbvers$
    CASE "5"
      LOADSUB ALL FROM "htbdde.csb"
    CASE "9"
      LOADSUB ALL FROM "htb8dde.csb"
  END SELECT
  PRINT
  PRINT " BASIC DDE Client CSUB (R"&Htbvers$&") est chargé.."
END IF
CALL Ddeinit(Result)
IF Result=0 THEN
  DISP "Doing some DDE conversation.."
  !WAIT 2
  ! Connexion à l'application, retourne un numéro de canal dans Ch
  CALL Ddeconnect(Ch1,"ServeurPrj","DdeServerConv1")
  IF Ch1<>0 THEN
    ! Se reconnecte avec un autre canal
    CALL Ddeconnect(Ch2,"ServeurPrj","DDeServerConv1")
    CALL Dderequest(Ch2,"DDeServerItem1",Ret$,Result) ! récupère la valeur
    CALL Ddeterminate(Ch2,Result) ! Termine la connexion sur le second canal
    CALL Ddeterminate(Ch1,Result) ! Sur le premier
  ELSE
    BEEP
    Buf$="ServeurPrj DDE connexion échouée"
  END IF
  CALL Ddeclean(Result) ! Libère les ressources DDE
ELSE
  BEEP
  Buf$="DDE initialisation échouée"
END IF
CLEAR SCREEN
PRINT Ddeinfo$
PRINT "Valeur du serveur: ";Ret$;
PRINT
PRINT Buf$
PRINT "DDE Test terminé"
DELSUB Ddeinit TO END
END
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

## XVII-E - Conclusion

Pour réaliser une connexion Manuel, c'est à dire qui demande les informations. Il suffit de placer ConnectMode à DdeManual pour le DdeClientConv. Ensuite, lors que l'on veut récupérer les données, on invoque l'instruction RequestData. Notez qu'il n'est plus nécessaire d'avoir des DdeClientItems dans ce cas.
Avec peu d'effort, vous pouvez rendre vos applications communicantes que se soit des serveurs ou des clients d'autre application et ce quels que soient leurs langages.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XVIII - DLL

DLL, Dynamic Library Link ou en français lien dynamique vers une librairie. Le fichier DLL est cette librairie. Le but étant au départ de permettre aux développeurs de bénéficier de fonction déjà existante et aussi de décharger la mémoire. Les DLL contiennent en effet des ressources qu'elle peuvent partager avec plusieurs programmes. C'est ressources permettent d'avoir accès à des fonctions, des composants...

## XVIII-A - Introduction

Qu'est ce qu'une DLL me direz-vous ? Une dll est un fichier contenant des ressources et la capacité de mettre à disposition ces ressources pour les programmeurs et donc pour les applications. Une DLL peut contenir des fonctions (exemple des fonctions mathématiques), une classe, des composants ou d'autre chose comme des icônes.
Pour ce qui est de son utilité, on peut en distinguer trois.

- La principale est qu'elle permet de partager ses ressources avec tout le monde. Vous pouvez distribuer ou récupérer une dll et ensuite l'utiliser ses ressources. Exemple si vous créez ou récupérer une DLL contenant des fonctions mathématiques, vous n'avez plus besoin de les réaliser dans votre application, il suffit de les importer. Leur utilisation se fait alors comme celle que l'on trouve dans Delphi comme **sqrt**.
- La seconde est qu'elle permet de découper son application en morceau afin d'améliorer la maintenance et le débogage. Si vous modifier une partie du code dans une dll, il vous suffit de redistribuer votre dll pour mettre l'application à jour. Cela permet aussi de séparer par fonctionnalité votre code. Je dois dire que l'intérêt de ce point est un peu flou. Il est quasiment inexistant pour les petites applications, il suffit de faire plusieurs unités et on peut aussi utiliser l'héritage, je ne parle pas ici de l'héritage des classes quoique mais de celui des unités(Pour Rappel : on peut inclure des unités dans son projet en spécifiant qu'ils ne sont qu'une copie de l'original ou un lien, ce qui implique qu'elles se mettent à jour si l'original est modifié). Reste le problème de la compilation qui concerne tout le projet au contraire de la DLL où seul la dll concernée est à recompiler.
- Le troisième est que l'on peut charger dynamiquement une DLL. Quand une DLL est chargée dynamiquement dans une application, elle ne réside en mémoire que lorsqu'elle est utilisée (appelé) et ensuite la mémoire est libérée (la dll est déchargée).

## XVIII-B - Ecriture des DLLs & Utilisation

Dans cette partie, nous allons voir comment écrire et utiliser une DLL de fonction, de Classe et pour terminer de composant.

### XVIII-B-1 - DLL de fonction

Pour ceux qui préfèrent la théorie avant la pratique, regardez les chapitres suivant.

Pour faire simple, utilisons l'expert DLL, Fichier -> nouveau -> Expert DLL.
Vous obtiendrez le prototype d'une Dll avec Delphi

Nous allons commencer une DLL de fonction Mathématique, libre à vous de rajouter d'autre fonction par la suite. Enregistrer le projet sous le nom LibMaths.
Notre première fonction sera la fonction factorielle. Bref rappel de Math, la factorielle de 1 vaut 0 et la factorielle de 0 vaut 1. On note l'opération factorielle : n! où n est un nombre entier positif et ! l'opérateur. Le produit factoriel de n vaut n multiplier par le produit factoriel de n-1 (pour les esprits vifs que vous êtes, cette définition vous fait penser à la récursivité). Exemple : 3! = 3*2! = 3*2*1! soit 3*2*1 donc 6. et 4! = 4*3! = 4*6 soit 24. Pour ce qui ne connaissent pas la récursivité, pas d'inquiétude car ce n'est pas le sujet du cours, l'essentiel est de comprendre le principe.

- Pour partir sur de bonne base, votre projet doit ressembler à ceci :

```
library LibMaths;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
{ Remarque importante concernant la gestion de mémoire de DLL : ShareMem doit
  être la première unité de la clause USES de votre bibliothèque ET de votre projet
  (sélectionnez Projet-Voir source) si votre DLL exporte des procédures ou des
  fonctions qui passent des chaînes en tant que paramètres ou résultats de fonction.
  Cela s'applique à toutes les chaînes passées de et vers votre DLL --même celles
  qui sont imbriquées dans des enregistrements et classes. ShareMem est l'unité
  d'interface pour le gestionnaire de mémoire partagée BORLNDMM.DLL, qui doit
  être déployé avec vos DLL. Pour éviter d'utiliser BORLNDMM.DLL, passez les
  informations de chaînes avec des paramètres PChar ou ShortString. }

uses
  SysUtils,
  Classes;

{$R *.res}

begin
end.
```

- Entre le windows et le begin, nous allons coder notre fonction factorielle soit :

```
function Factorielle(n : integer): integer;
begin
  // On n'aurait pu traité le cas 0 et 1 séparement mais cela fait un test de plus
  if n = 0 then Result := 1
  else Result := n * Factorielle(n-1);
end;
```

*Vérifiez bien que la fonction puisse s'arrêter quand vous faites de la récursivité sinon vous créez une boucle infinie.*

- Maintenant, nous allons exporter notre fonction pour qu'elle puisse être utilisée par d'autre application. Entre le end de la fonction et le begin du projet, tapez :

```
exports
  Factorielle;
```

Le code source complet doit ressembler à ceci. Mais attention, il ne s'agit pas d'un exécutable, vous ne pouvez que le compiler. Pour ce faire, Tapez CTRL+F9 ou Projet -> Compiler LibMaths.

```
library LibMaths;

uses
  SysUtils,
  Classes;

function Factorielle(n : integer): integer;
begin
  if n = 0 then Result := 1
  else Result := n * Factorielle(n-1);
end;

exports
  Factorielle;

begin

end.
```

Nous allons maintenant nous attaquer à la programmation d'une application utilisant cette dll. Pour cela enregistrer votre projet si ce n'est déjà fait et commencez un nouveau projet.
Dans la Form ajouter un EditBox pour la saisie de l'utilisateur et un Label ou EditBox pour afficher le résultat plus d'autres labels pour documenter votre application. Ajoutez aussi deux boutons l'un pour calculer et l'autre pour


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

fermer. Pour le bouton Fermer vous pouvez utiliser un BitBt (onglet : Supplément) et positionner sa propriété Kind à bkClose.

- Entre Implémentation et les directives de compilation, importez la fonction. L'importation se fait en reprenant la déclaration de la fonction soit ici : function Factorielle(n : integer): integer; Attention à la casse (il faut le f majuscule) ajouter le mot reservé external 'nom du fichier dll'; sans l'extension.

```
implementation

function Factorielle(n : integer): integer; external 'LibMaths';

{$R *.DFM}
```

- On peut maintenant utiliser cette fonction, dans l'événement onclick du bouton Calculer, ajouter le code suivant : Remarque : Il n'y a pas de test sur la saisie de l'utilisateur dans un souci de clarté !

```
procedure TForm1.btTestClick(Sender: TObject);
var
  n : integer;
begin
  n := StrToInt(Edit1.Text);
  lbResultat.Caption := IntToStr(Factorielle(n));
end;
```

- Exécutez et testez votre application. Notez que le fichier dll doit se trouver dans le même répertoire que l'application l'utilisant ou dans le répertoire window ou window/systeme (à éviter cependant sauf cas particulier comme les dll de windows :) ).

Le code source complet ci dessous.

```
unit PrincipalFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, Mask;

type
  TForm1 = class(TForm)
    BitBtn1: TBitBtn;
    Label1: TLabel;
    btTest: TButton;
    lbResultat: TLabel;
    Edit1: TEdit;
    procedure btTestClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

function Factorielle(n : integer): integer; external 'LibMaths';

{$R *.DFM}

procedure TForm1.btTestClick(Sender: TObject);
var
  n : integer;
begin
  n := StrToInt(Edit1.Text);
  lbResultat.Caption := IntToStr(Factorielle(n));
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
end;

end.
```

## XVIII-B-2 - Théorie & Subtilité

Si vous avez regardé la partie au-dessus, le premier mot nouveau fut **library**. Ce mot indique au compilateur qu'il ne s'agit pas d'un programme principal (contenant main) et donc sans point d'entrée de type winmain. Ensuite la partie entre **begin** et **end**. correspond comme pour les projets à la partie initialisation. Dans la partie initialisation, vous pouvez initialiser les variables globales. Voilà pour les détails.
**exports** : permet de lister les fonctions à rendre visible/accessible depuis l'extérieur.
On peut rajouter y ajouter des options :

- name : renomme la fonction exportée, par exemple quand le nom de la fonction existe déjà ailleurs.
  exporte la fonction toto sous le nom titi, c'est titi qu'il faudra utiliser dans les applications utilisant cette dll. Attention à la casse.

  ```
  exports
    Toto name 'titi';
  ```

  exporte la fonction toto sous le nom titi, c'est titi qu'il faudra utiliser dans les applications utilisant cette dll. Attention à la casse.
- index : il spécifie un index pour la fonction exporter. Toutes les fonctions sont indexées, si on change l'index d'une fonction les index des fonctions suivantes reprennent à partir de celui-ci. N'est plus utilisé en environnement 32 bits (win98 et +) car le gain de performance est négligeable.

  ```
  exports
    Toto index 5;
  ```

  La fonction Toto sera accessible par l'indice 5.

```
exports
  Toto index 5;
```

Ces options sont cumulables, on peut exporter une fonction par nom et par index (Toto name 'titi' index 5;).
Les noms des fonctions exportées sont sensibles à la casse (il y a une différence entre majuscule et minuscule).
Convention : La convention permet de spécifier la gestion du passage de paramètre autrement dit comment les paramètres vont être gérés. Elles sont pour l'instant au nombre de quatre.

- **stdcall** : qui est la convention par défaut mais il est bon de le préciser quand même (en cas d'évolution du langage). Cette convention permet d'exporter ces fonctions dans une grande majorité des langages. Elle est apparue avec les systèmes 32 bits.
- **register** : permet de placer les paramètres dans les registres donc d'optimiser leur vitesse d'accès.
- **pascal** : utilise une convention propre au Pascal.
- **cdecl** : Convention du c.

Par défaut nous vous conseillons d'utiliser surtout **stdcall**, c'est le format le plus reconnu par tous les langages.
Si votre dll ne sera utilisée que par des programme avec Delphi, utilisez **register** qui est plus performant.

*Si vous voulez connaitre plus en détail les différences entre les conventions, allez ici :* ***Conventions d'appel***

Pour déclarer une convention :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

fonction toto(liste de paramètre et leur type)[:retour si function]; **stdcall**;

Exemple avec notre dll :

- Dans le projet dll, on ajoute la convention à la fin de la fonction. Remarquez le mot export (sans s) qui se place après la convention et notez qu'il n'est pas obligatoire.

```
function Factorielle(n : integer): integer; stdcall; export;
begin
  if n = 0 then Result := 1
  else Result := n * Factorielle(n-1);
end;
```

- Dans le projet utilisant la dll, on rajoute la même convention

```
implementation

function Factorielle(n : integer): integer; stdcall; external 'LibMaths';

{$R *.DFM}
```

Importation :

- Importation par nom : on reprend la déclaration de la Dll et on ajoute **external** 'nom du fichier dll sans l'extension'
  **function** Toto(n : integer): integer; **stdcall**; **external** 'NomDLL';
  On peut aussi changer le nom de la fonction sous lequel on veut la manipuler dans l'application ex :
  **function** Machin(n : integer): integer; **stdcall**; **external** 'NomDLL' name 'Toto'
  Où Toto est le nom de la fonction dans la DLL et Machin le nom de la fonction dans l'application. Attention, pas de ';' entre external et name.
- Importation par index : par grand chose à dire donc un exemple :
  **function** Machin(n : integer): integer; **stdcall**; **external** 'NomDLL' index 10;

*Manipulation des chaînes Longues :*
*Si vous avez utilisé l'Expert Dll, vous avez pu remarquer un commentaire vous mettant en garde sur l'utilisation des chaînes longues. Cette mise garde ne concerne que la manipulation des chaînes comme paramètre, c'est à dire entrant ou sortant de la dll (en interne, pas de problème). En effet, Delphi gère les chaînes longues d'une façon qui est incompatible avec les autres langages. Vous avez lors deux solutions soit utiliser ShareMem et inclure la dll : BORLNDMM.DLL avec votre application, soit utiliser des PChar ou ShortString (voir le chapitre XVIII.B.5. DLL & Chaîne de caractère). Attention même les chaînes qui sont dans des enregistrements ou des classes sont concernées !*
*Voir le chapitre **Chargement Statique/Dynamique** pour les importations avancées.*

## XVIII-B-3 - DLL de Classe

Les dll permettent aussi d'exporter des classes à condition de comprendre quelques subtilités liées au classe. Commençons par définir une classe dans un projet dll (Nouveau-> expert dll). ajouter une nouvelle unité (Nouveau->Unité) qui contiendra notre classe et commencons sa définitions.

```
unit MaClassDLLUnt;

interface

type
  TMaClass = class
  private
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    bidon : integer;
  public
    function GetBidon() : integer; virtual; stdcall; export;
    procedure SetBidon(NewValeur : integer); virtual; stdcall; export;
  end;

implementation

function TMaClass.GetBidon():integer; stdcall; export;
begin
  // Renvoie la valeur de bidon
  Result := Bidon;
end;

procedure TMaClass.SetBidon(NewValeur : integer); stdcall; export;
begin
  // Change la valeur de bidon en NewValeur
  Bidon := NewValeur;
end;

end.
```

- J'ai déjà noté les fonctions que je voulais exporter mais les déclarer dans la partie **exports** ne serait pas suffisant. Nous manipulons ici une classe et non un ensemble de fonction, c'est donc la classe qu'il nous faut exporter. Voici comment procéder :

```
function CreeInstanceMaClass() : TMaClass; export; // Pas de virtual ici !
begin
  Result := TMaClass.Create;
end;
```

- Dans la partie **exports**, on exporte seulement cette fonction. On exporte la classe depuis la page principal soit celle avec le mot **library**
Le code complet de la page principal puis de l'unité contenant la définition de la classe :

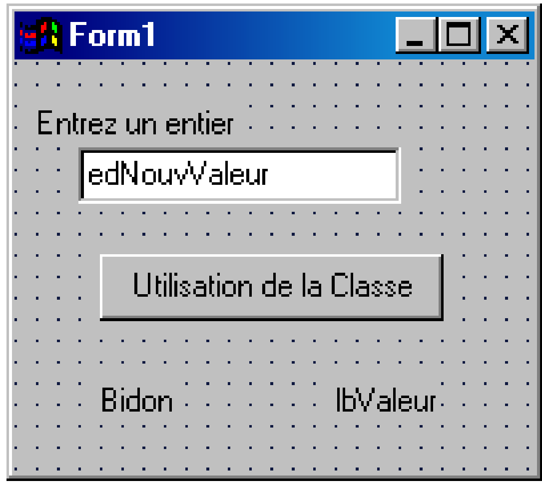
```
// Page principal
library Dll_ClassPrj;

uses
  SysUtils,
  Classes,
  MaClassDLLUnt in 'MaClassDLLUnt.pas';

{$R *.res}

exports

 CreeInstanceMaClass; // Seul la fonction permettant d'instancier(créer) un objet de la classe est expo

begin

end.

// Unité contenant la définition de la classe
unit MaClassDLLUnt;

interface

type
  TMaClass = class
  private
    bidon : integer;
  public
    function GetBidon() : integer; virtual; stdcall; export;
    procedure SetBidon(NewValeur : integer); virtual; stdcall; export;
  end;

  function CreeInstanceMaClass() : TMaClass; stdcall; export; // Pas de virtual ici !
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
implementation

function TMaClass.GetBidon():integer; stdcall; export;
begin
  // Renvoie la valeur de bidon
  Result := Bidon;
end;

procedure TMaClass.SetBidon(NewValeur : integer); stdcall; export;
begin
  // Change la valeur de bidon en NewValeur
  Bidon := NewValeur;
end;

function CreeInstanceMaClass() : TMaClass; stdcall; export; // Pas de virtual ici !
begin
  Result := TMaClass.Create;
end;

end.
```

### Téléchargez le code source de la Dll

Voilà pour la dll, nous allons créer une application l'utilisant. Contrairement à ce qui avait été dit auparavant export ne sera pas la dernière déclaration dans l'importation des fonctions. Il s'agit d'abstract qui terminera nos déclarations. Pourquoi et bien parce que mais plus sérieusement, pour pouvoir utiliser une classe il faut que l'application connaisse sa définition. C'est pourquoi nous avons déclaré toutes nos méthodes (les fonctions quand elles sont dans une classe) virtuel et que nous utilisons abstract ce qui permet de mettre la définition sans implémenter les méthodes (voir les classes pour plus de précision).
Dans un nouveau projet, voici pour l'apparence :




Ensuite, repérez la fin de la déclaration de Form et ajouter la définition de la classe dll. Il suffit de recopier la déclaration de la classe et d'ajouter **abstrac** à la fin de chaque méthode exportée. Après les directives de compilation ({$R *.DFM}), importez la fonction qui permet de crée une instance de la classe. voir ci dessous le code complet.

```
unit UtilDll_ClassUnt;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    edNouvValeur: TEdit;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

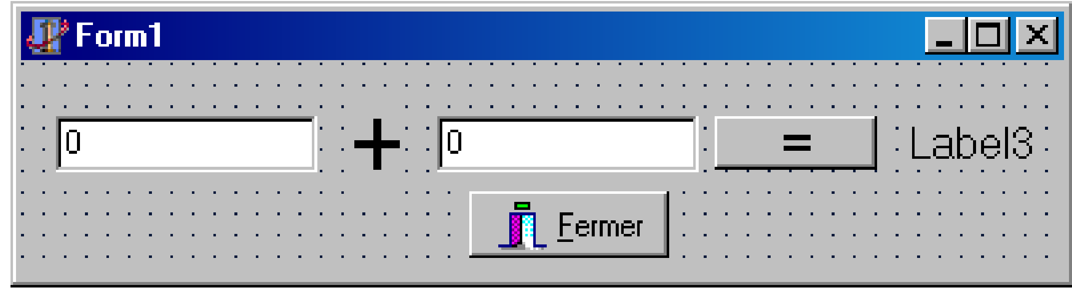
```
    Label1: TLabel;
    lbValeur: TLabel;
    btManipClasse: TButton;
    Label2: TLabel;
    procedure btManipClasseClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

type
  TMaClass = class
  private
    // On n'importe pas Bidon car l'utilisateur ne devrait jamais
    // Manipuler directement cet valeur, voir les classes
    // pour comprendre la philosophie sous jascente.
    // pas de : bidon : integer;
  public
    function GetBidon() : integer; virtual; stdcall; export; abstract;
    procedure SetBidon(NewValeur : integer); virtual; stdcall; export; abstract;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function CreeInstanceMaClass() :
 TMaClass; stdcall; external 'Dll_ClassPrj'; // Pas d'abstract ici, ce n'est pas une méthode !

procedure TForm1.btManipClasseClick(Sender: TObject);
var
  MaClass : TMaClass;
begin
  // La classe ne fait pas grand chose
  // mais permet de rester concentrer sur le sujet : les dll
  // Crée une instance de la classe
  MaClass := CreeInstanceMaClass();
  // Affecte une nouvelle valeur à bidon
  MaClass.SetBidon(StrToInt(edNouvValeur.Text));
  // recupère la valeur bidon et l'affiche
  lbValeur.Caption := IntToStr(MaClass.GetBidon);
  // libère la classe
  MaClass.Free;
end;

end.
```

**Télécharger le code source de la partie utilisation ici**

## XVIII-B-4 - DLL de Composant

Nous allons réaliser une petite calculette sans prétention. Le but est d'apprendre à utiliser des composants Delphi dans une DLL et de voir quelques écueils.
Commencez un nouveau projet DLL et ajoutez une fiche. Enregistrer le projet (PMadll) et l'unité (UMadll). Dans la fiche ajouter les composants et changer la propriété BorderStyle à bsDialog de la fiche, de façon à obtenir ce résultat :


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.




*Le bouton égal est obtenu en mettant le signe '=' dans Caption et le plus par un label avec une fonte différente*

Ajoutez dans la clause Uses : Forms avant UMadll.

- Dans l'événement OnClick du bouton égal :

```
procedure TMaForm.Button1Click(Sender: TObject);
begin
  Label3.Caption := FloatToStr(StrToFloat(Edit1.Text) + StrToFloat(Edit2.Text));
end;
```

- J'ai renommé Form1 en MaForm pour la suite !
- Retournez sur l'unit Projet, Il nous faut donner le moyen à l'application de créer la Form car vous remarquerez qu'il n'y a pas de création automatique contrairement aux exécutables.

```
procedure Creer_Form; stdcall; export;
begin
  // Crée une instance de Form
  DecimalSeparator := '.';  // change le séparateur décimal
  Application.CreateForm(TMaForm, MaForm);
  MaForm.Label3.Caption := '0';
end;
```

- Comme nous créons la Form, nous allons nous donner le moyen de la libérer

```
procedure Free_Form; stdcall; export;
begin
  // Libère la Form
  MaForm.Free;
end;
```

- Nous allons maintenant ajouter une fonction retournant la somme calculée.

```
function fncsomme():Double; stdcall; export;
begin
  with MaForm do
  begin
    // Montre la forme et attend sa fermeture
    ShowModal;
    // Renvoi le résultat du Calcul
    Result := StrToFloat(Label3.Caption);
  end;
end;
```

- Et une version procédure qui ne sera utilisé que dans le chapitre autre langages.

```
procedure proSomme(var R: Double); stdcall; export;
begin
  with MaForm do
  begin
    // Montre la forme et attend sa fermeture
    ShowModal;
    // Modifie la variable R (passage par adresse)
```

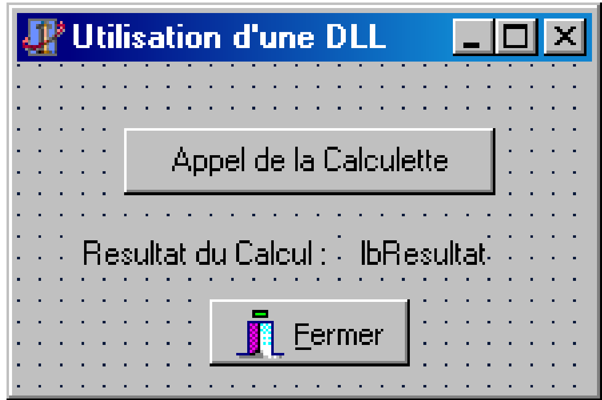

Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        R := StrToFloat(Label3.Caption);
      end;
    end;
```

- Exportons nos fonctions et notre DLL sera terminée

```
    exports
    Creer_Form, Free_Form,
    fncsomme, proSomme;
```

Réalisons maintenant une application qui utilisera cette dll. Enregistrer votre projet dll si ce n'est déjà fait et commencez un nouveau projet application. Nous allons faire simple et nous contenter d'un bouton pour appeler la calculette et d'un label pour afficher le résultat, nous mettrons aussi la fiche en bsDialog.




L'apparence étant faite, passons au codage. Il nous faut importer les fonctions permettant de manipuler la DLL. Donc nous allons importer les fonctions qui permettent de créer la form, de la libérer et bien sûr d'utiliser la calculette. Juste après les directives de compilation ( {$R *.DFM} ) ajoutez :

```
procedure Creer_Form; stdcall; external 'pMadll.dll';
procedure Free_Form; stdcall; external 'pMadll.dll';
function Total():Double; stdcall; external 'pMadll.dll' name 'fncsomme';
```

- A niveau de la **Function**, vous aurez remarqué que j'ai changé le nom de la fonction pour Total.
- Dans l'événement OnClick du Bouton 'Appel de la Calculette', Tapez le code suivant (fmUseDll est le nom de ma Form) :

```
    procedure TfmUseDLL.btAppelCalculetteClick(Sender: TObject);
    var
      R : Double;
    begin
      Creer_Form;
      R := Total();
      lbResultat.Caption := FloatToStr(R);
      // Ne pas oublier de libérer la form à la fin
      Free_Form;
    end;
```

- Enregistrer votre projet et vérifiez que la DLL se trouve dans le même répertoire que la casse des fonctions importés est correct.
  Lancez votre appli, vous disposez maintenant d'une dll proposant une calculette :)

## XVIII-B-5 - DLL & Chaîne de caractère

Si votre dll doit utiliser des chaînes longues comme paramètre, vous allez vous heurter à un problème. Heureusement deux solutions s'offrent à vous.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- La plus simple utiliser l'unité ShareMem en le déclarant dans la clause uses avant toutes les autres déclarations. Vous pouvez alors utiliser les chaînes longues comme bon vous semble. Cette méthode bien que simple à un gros inconvénient, pour que votre application puisse fonctionner, il lui faut alors une autre dll. Le problème et qu'il ne faut pas oublier de fournir cette dll sous peine d'empêcher l'exécution de votre application. le fichier dll à inclure : BORLNDMM.DLL.
- La plus indépendante, utilisez des PChars ou des ShortStrings. En ce qui concerne les ShortStrings, ils se manipulent comme les strings si ce n'est le nombre limit de caractère qu'ils peuvent contenir.

*Attention même les chaînes qui sont dans des enregistrements ou des classes sont concernées !*

Je ne parlerai ici que des PChars et de leur utilisation avec les dll, le reste ne posant pas de difficulté. Tout d'abord un PChar est une chaîne un peu spéciale (rien que le nom est bizarre mais très révélateur). Les Pchars sont des pointeurs sur une chaîne de Char terminé par un indicateur de fin. Une bonne nouvelle lorsqu'on les manipule sous delphi, ils sont compatibles avec les string.
MonPChar := MonString;
Par contre, l'indicateur de fin est le caractère #0. Du coup ce caractère ne doit pas se retrouver dans la chaîne (la chaîne s'arrête dés qu'elle rencontre celui-ci). Cet indicateur de fin est aussi appelé null ou Zéro Terminal.
Le point le plus délicat est l'espace mémoire occupé par ces PChars, contrairement au string, il est fixe. Mais avant d'entrer dans le vif du sujet un exemple d'utilisation vous permettra de fixer les choses :

- Soit MaFonction : une fonction qui attend une chaîne de caractère de type PChar qui la manipule et la renvoie. Le détail de la fonction n'est pas ce qui nous intéresse.

```
function MaFonction(S :PChar; Taille : integer);
```

  On ne met pas **var** devant S car c'est un pointeur donc une adresse (passer une adresse d'une adresse :o )
- Dans une partie du code nous voulons utiliser cette fonction, soit :

```
var
  U : array[0..51] of Char; // Déclaration d'une chaîne de caractère
begin
  U := 'affectation d''une valeur';
  MaFonction(U, sizeof(U));
  ... Suite des instructions...
end;
```

Dans le passage de paramètre, on passe un pointeur sur la chaîne de Caractère soit U par adresse et la taille de la chaîne. Quoique fasse la fonction MaFonction, elle ne devra pas dépasser cette taille-1 (pensez au caractère de fin de chaîne).
Pour copier une chaîne dans un Pchar en limitant la taille d'arriver, vous pouvez utiliser la fonction **StrPLCopy**

```
StrPLCopy(VariablePChar, MonString, TailleMaxi);
```

On peut trés bien se passer des tableaux de caractère en utilisant tout de suite un PChar. Dans ce cas, il faut allouer de l'espace mémoire avant de les utiliser. Le code précedent deviendrait :

```
var
  U : PChar; // Déclaration d'un pointeur sur chaîne de caractère
begin
  U := StrAlloc(50);
  StrLPCopy(U, 'affectation d''une valeur', 49);
  MaFonction(U, 50); // Pas de Size de U surtout !
  ... Suite des instructions...
end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Les notions vue ici sont aussi valables pour transmettre des tableaux n'étant pas des tableaux de caractère. Il suffit de remplacer les pointeurs PChar par un pointeur sur votre tableaux.
Exemple avec un tableau de Double :

- Dans l'application, on déclare le tableau normalement met on ne transmet que le premier élement par adresse. En transmettant le premier élément par adresse, on passe en fait un pointeur sur le tableau. On n'aurait pu utilisé un vrai pointeur.

```
var
  Mat : array[0..5] of double;
begin
  for i:=0 to 5 do
  Mat[i] := 0;
TransmettreMat(Mat[0], 5); // On transmet aussi la Taille !
end;
```

- Dans la DLL, la fonction attend un paramètre par adresse (le pointeur sur le tableau) et la taille.

```
procedure transTableau(var T : Double;taille : integer);stdcall;export;
type
  TTabMat = array[0..6] of double;
  PTabMat = ^TTabMat;
var
  i : integer;
  pMat : PTabMat;
begin
  // fixe le séparateur décimal
  DecimalSeparator := '.';
  // fait pointer pMat sur l'adresse de T donc sur le tableau transmit
  pMat := @T;

  for i:=0 to taille do
    pMat^[i] := pMat^[i] + 1.33;
end;
```

## XVIII-C - Chargement statique/dynamique

Nous avons jusqu'à présent charger les dll de façon statique. Ceci présente plusieurs avantage, vous savez tout de suite si votre application n'arrive pas à charger votre dll, le code est plus compact et vous ne vous posez pas de question sur le déchargement de la dll. Par contre la dll est toute de suite chargée en mémoire et elle ne libèrera cet espace que lorsque l'application sera terminée, si les ressources de la dll ne sont utilisées que ponctuellement quel gaspillage. Nous allons apprendre ici à charger les dll de façon dynamique, c'est à dire les chargé en mémoire que pour le temps de leur utilisation effective. Un autre avantage est la création de plugin (mais je n'en sais pour l'instant pas plus).
Voyons tout de suite les nouveaux mots clé :
D'abord, il faut charger la dll pour cela, on utilise **LoadLibrary** :

```
Handle := loadlibrary('MonfichierDll.dll');
```

Le handle permet de pointer sur cette dll (pensez aux objets d'une classe), si le chargement échoue le handle à une valeur nulle.
Ensuite, il faut charger chaque fonction dont on n'aura besoin grâce à **GetProcAddress** :

```
@MaFonction := GetProcAddress(Handle, 'nomdelafonctiondanslaDLL');
```

Où nomdelafonctiondanslaDLL est le nom de la fonction que l'on veut importer et MaFonction un pointeur sur processus, ici on récupère son adresse ! Si le chargement de la fonction échoue le pointeur renvoie nil.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Lorsque la dll n'est plus requise, il faut la décharger par l'appel à **FreeLibrary**.

```
FreeLibrary(Handle);
```

Reprenez l'exemple sur la Factorielle (le projet et la dll associé). Nous allons modifier l'application pour qu'elle charge dynamiquement la dll libMaths.

```
unit PrincipalFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, Mask;

type
  TForm1 = class(TForm)
    BitBtn1: TBitBtn;
    Label1: TLabel;
    btTest: TButton;
    lbResultat: TLabel;
    Edit1: TEdit;
    procedure btTestClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btTestClick(Sender: TObject);
var
  n : integer;
  Handle: THandle;
  MaFactorielle: TMyProc;
begin
  n := StrToInt(Edit1.Text);
  // Charge une dll dynamiquement et la libère ensuite
  Handle := loadlibrary('LibMaths.dll'); // Charge la dll

  if Handle <> 0 then
  begin
    try
      // Charge dynamiquement une fonction de la dll
      @MaFactorielle := GetProcAddress(Handle, 'Factorielle');
      if @MyProc <> nil then
      begin
        lbResultat.Caption := IntToStr(MaFactorielle(n));
      end;
    Finally
      FreeLibrary(Handle); //Assure le déchargement de la dll
    end; // Try..Finally
  end
  else
    ShowMessage('Impossible de charger la DLL');
end;

end.
```

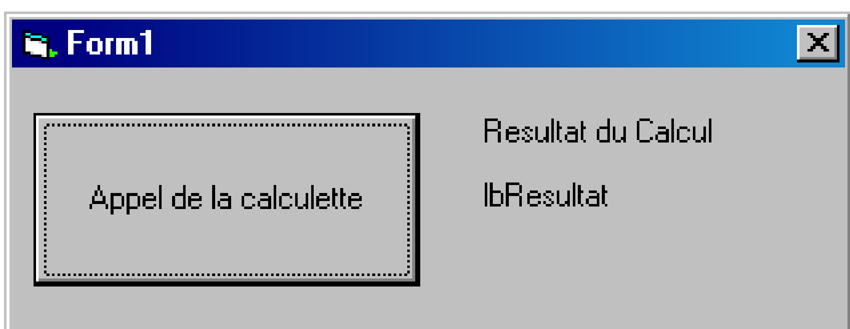
La déclaration de fonction a disparu et dans l'événement onclic du bouton, on charge la dll, la fonction Factorielle et une fois le travail accomplit, on libère la DLL.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

***Libération des ressources :***
*Remarquez l'imbrication du try..finally : On essaie d'abord de charger la dll si l'opération réussit (ici traité par un if mais un try..except est tout à fait valable), on rentre dans un bloc protégé (try) même si une exception arrive la ressource sera libérée car elle est dans la partie finally. Par contre, ce serait une erreur de mettre loadlibrary dans le code protégé.*

## XVIII-D - DLL et Autres Langages

Dans l'intro, vous avez pu lire que la DLL pouvait être écrite dans un langage et utiliser dans un autre. Je ne vous présenterais pas ici des dll écrite dans un autre langage et utilisé dans Delphi car cela à peut d'intérêt. Il vous suffit de reprendre ce que l'on a déjà vu. Parlons plutôt du cas où la Dll a été écrite en Delphi et l'application dans un autre langage. Une chose importante, les conventions d'appel dans la DLL doivent être compatibles avec le langage utilisé pour l'application. En général stdcall.
Un autre point très délicat concerne les paramètres manipuler par la dll, notamment les chaînes longues voir **XVIII-B-5 DLL & Chaîne de caractère**. Mais aussi le composant Menu qui pour une raison qui m'échappe ne marche qu'avec des applications Delphi.
Les précautions d'usage étant établies, essayons de réaliser deux exemples. Le premier utilisera le langage VB et le second HT Basic avec un but identique manipuler la Calculette que nous avons créée dans le chapitre **102.2.4. DLL de composant**. Que vous n'ayez ni l'un ni l'autre de ces langages n'a pas vraiment d'importance, c'est exemple n'étant que didactique. L'important est de savoir comment importer les ressources contenues d'une dll dans le langage que vous utilisez.
Reprenons notre dll Calculette et copions la dans le répertoire où se retrouvera l'application en VB.
Crée un nouveau projet VB et occupons-nous de son apparence en réalisant une fiche dans ce style :




Attaquons ensuite le codage, comme pour Delphi, nous allons importer les fonctions nécessaires à la manipulation de la DLL. Ajoutez les lignes suivantes au début de la page de code de la fiche :

```
Private Declare Sub CreerFormDll Lib "PMadll.dll" Alias "Creer_Form" ()
Private Declare Sub FreeFormDLL Lib "PMadll.dll" Alias "Free_Form" ()
Private Declare Function DllTotal Lib "PMadll.dll" Alias "fncsomme" () As Double
```

Idem pour le bouton 'appel de la calculette' :

```
Private Sub btAppelCalculette_Click()
  CreerFormDll
  lbResultat.Caption = DllTotal
  ' N'oublions pas de libérer la Form
  FreeFormDLL
End Sub
```

Exécuter votre application, et voilà une calculette faite en Delphi utilisée par VB.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Ci-dessous le code source complet d'une application écrite en HT Basic utilisant la calculette écrite en Delphi. Comme je l'avais déjà dit auparavant, peu importe que vous utilisiez ce langage ou non. Notez par contre la similitude du codage dans les différents langages.

```
DLL UNLOAD ALL
! Change le répertoire en cours
MASS STORAGE IS "D:\delphi\Utilisation_de_la_DLL_par_HTB\dll"
! Charge la DLL
DLL LOAD "pMaDll"
! Importe les fonctions et les renomme.
DLL GET "STDCALL VOID pMaDll:Creer_Form" AS "Creerformdll"
DLL GET "STDCALL VOID pMaDll:Free_Form" AS "Freeformdll"
DLL GET "STDCALL VOID pMaDll:proSomme" AS "Dlltotal"

! rechange de répertoire
MASS STORAGE IS "D:\delphi\Utilisation_de_la_DLL_par_HTB"
! Un peu de ménage
CLEAR SCREEN

! Affiche à
 l'écran toutes les fonctions disponibles dans la dll (pas seulement celle qui sont importées)
LIST DLL

REAL R

! Crée la Form
Creerformdll
! Affiche la Form en Modal (Mécanisme interne de la fonction DllTotal)
Dlltotal(R)
! Libère la form
Freeformdll
! Affiche le résultat à l'écran
PRINT R
! Décharge la DLL et oui en HTB, il faut tout faire soi même ;)
DLL UNLOAD ALL
! Fin
END
```

## XVIII-E - Conclusion

***Petite astuce de débogage :***

*Pour explorer le code de votre dll pendant l'exécution, ouvrez votre projet dll dans exécuter choisissez paramètre et indique une application exploitant votre dll dans Application hôte. la dll se comporte alors comme si elle était exécutable, vous pouvez placer des points d'arrêt, faire du pas à pas ...*


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XIX - Gestion des exceptions

Le présent tutoriel n'a pas la prétention de tout vous apprendre sur la gestion des exceptions (comme tous les tutoriaux que je fais mais celui-la plus que les autres). Mais vous devriez vous sentir à l'aise avec la bête et ne plus rechigner à les utiliser. Pour ceux qui ont déjà pris l'habitude d'utiliser leurs services, je leur conseille de le parcourir quand même, il se pourrait qu'ils apprennent quelque chose. Si tel n'était pas le cas, toutes mes excuses (et mieux encore vous en connaissez plus sur le sujet, je vous invite à compléter/modifer ce tutorial).

## XIX-A - Introduction

Tout d'abord qu'est qu'une exception ? Comme son nom l'indique une exception est un événement imprévu par le programmeur (ou programme). Vous me direz comment peut-il gérer quelques chose qu'il n'a pas prévu ? Très simple, le programmeur sachant que le monde n'est pas parfait aura pris le soin de protéger des blocs d'instructions sensibles.
Concrètement, lorsque vous développez votre application, vous vous attendez à ce que telle ou telle chose soit disponible (un fichier dll par exemple) ou que l'utilisateur entre tel type de donnée dans telle variable (champ de saisie où il faut entrer des nombres) (par utilisateur, je considère une personne ou le programme lui-même) or il arrive que les choses ne se passent pas tout à fait comme vous les avez souhaitées, fichier absent, caractère vers une variable numérique etc...
Pour protéger votre application et plus précisément un bloc de code, une solution est d'utiliser les exceptions et ainsi éviter à votre application de planter lamentablement. Une autre solution serait de faire des tests mais si pour des cas simples, les tests sont faciles à mettre en place (l'exemple de la division par zéro est assez parlant), il n'en est pas de même pour des tâches plus complexes ou sujettes à différents types d'erreur (comme l'accès à un fichier). D'autres diront que gérer par exception, les cas à problème, allège le code (à vous de voir).
Les exceptions peuvent être de natures différentes suivant l'opération où elles se produisent. Ces différentes exceptions sont appelées type d'exception ou classe d'exception. Par exemple lors d'une tentative de conversion d'un string en float, si le string contient des lettres, on aura une exception de classe conversion (EConverError).
La protection d'un bloc de code se fait par l'utilisation d'un couple de mot clé. Suivant la façon dont l'on veut gérer l'exception, deux couples existent, **try**..**finally** et **try**..**except**. Les deux couples protégent le code situé entre **try** et l'autre mot clé, si une exception arrive entre ces deux mots clés , le programme arrête de traiter le bloc protégé et passe tout de suite aux instructions comprises entre le second mot clé du couple et poursuit ensuite les instructions suivant ce second bloc de code. La différence entre le couple **try**..**finally** et **try**..**except** se situe dans l'exécution du second bloc d'instruction.

- Avec **try**..**finally**, les instructions du second bloc d'instruction sont toujours exécutées.

```
instructions1
try
   // Bloc de code à protéger
   instructions protégées
finally
   // Second bloc de code
   // Ce bloc sera exécuté à la fin des instructions protégées
   // ou dès qu'une erreur survient dans le bloc de code protégé
   instructions2
end;
// suite des instructions qui seront exécutées si elles ne provoquent pas d'erreur
instructions3
```

- Avec **try**..**except**, les instructions du second bloc d'instruction ne sont exécutées que s'il y a eut une exception dans le bloc protégé. Le couple **try**..**except** propose en plus de traiter, l'exception en fonction de la classe d'exception.

```
instructions1
try
   // Bloc de code à protéger
   instructions protégées
except
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
            // Second bloc de code
            // Ce bloc ne sera exécuté que si une erreur survient dans la partie protégée
            instructions2
        end;
        // suite des instructions qui seront exécutées si elles ne provoquent pas d'erreur
        instructions3
```

Vous l'aurez remarqué, le bloc dit protégé est celui qui le paraît le moins. En fait par bloc protégé, je pense que l'on désigne la protection du programme contre un code sensible. Votre programme ne va plus planter lamentablement lorsqu'il rencontrera une erreur dans un bloc protégé. Par contre vous continuerez d'avoir les messages d'erreur en l'exécutant depuis Delphi, cela est destiné à vous aider à vérifier que vous interceptez bien la bonne classe d'exception.

## XIX-B - Try..Finally

La syntaxe de try..finally est :

```
try
   instruction1
finally
   instruction2
end;
```

Instruction1 est une partie sensible du code (manipulation d'un fichier, création d'un objet, etc...) et instruction2 une partie du code qui doit être exécuté quoiqu'il arrive (libération de ressource, fermeture d'une connexion etc...). Si une erreur survient dans les instructions du bloc instruction1, l'exécution passe immédiatement à l'exécution d'instruction2 sinon l'exécution termine les instructions et passe ensuite au instruction2.
Vous l'aurez compris, son utilisation est fortement recommandée pour libérer une ressource même si le programme rencontre une erreur. Mais attention, l'instruction demandant la ressource doit se trouver à l'extérieur du try..finally. Dans le cas contraire, s'il arrivait que le programme ne puisse pas allouer la ressource, tenter de la libérer peut provoquer une erreur.
Exemple : Création d'un objet

```
var
   Ob : TObjetExemple;
begin
   Ob := TObjetExemple.Create; // Création de l'objet
   try
      {instruction}
   finally
      Ob.Free; // Libération
   end;
end;
```

Exemple : assignation d'un fichier

```
function OuvrirF(Nom : TFileName) : boolean;
var
   F : Textfile;
   S : string;
   i, j, valeur :integer;
begin
   AssignFile(F,Nom);
   try
      Reset(F);
      readln(F,S);

      {instruction}

      Result := True;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
   finally
      CloseFile(F);
   end;
end;
```

Notez la position de demande d'allocation de ressource par rapport au **try**..**finally**.

***Attention :***
*Tant que vous n'avez pas appelé CloseFile(F), vous ne pouvez pas manipuler le fichier (renommer, détruire, déplacer etc...). Ne l'oubliez pas ! Ceci est valable pour les fichiers mais aussi pour d'autres ressources (base de donnée, périphérique...).*

***Précision :***
*La protection d'un bloc de code permet d'éviter la propagation du message d'erreur mais dans certain cas, il peut être nécessaire de relancer sa diffusion. La commande **raise** peut être utilisée à cet effet voir son **chapitre** pour plus de précision.*

# XIX-C - Try..Except

## XIX-C-1 - Grammaire

Examinons une application fournissant un champ de saisie n'attendant que des nombres comme saisie. Deux solutions s'offrent à vous, la première empêcher l'utilisateur d'entrer autre chose que des caractères numériques (assez compliqué à mettre en place mais mieux au niveau de l'ergonomie) et la seconde utiliser les exceptions. Laissons la première de côté et intéressons-nous à la seconde. Ce que nous devons protéger est le moment où la saisie de l'utilisateur doit être affectée à une variable de type numérique. Vous pouvez tester en réalisant une application faisant une telle opération. Lors de l'exécution, un message d'erreur se produira dès que vous affecterez des lettres à la variables, plantant le plus souvent votre application. Par contre en protègeant votre bloc de code, non seulement, vous limitez l'erreur à cette portion de code et vous pouvez en plus réaliser un traitement spécifique au problème (réinitialiser des variables, informer l'utilisateur...).
La syntaxe est la suivante :

```
try
   instruction1
except
   instruction2
end;
instruction3
```

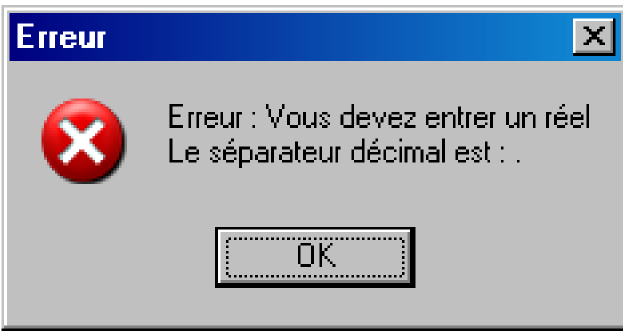
Instruction1 est comme pour le **try**..**finally** la partie sensible du code tandis qu'instruction2 le code qui sera exécuté si instruction1 provoque une erreur. Si une erreur survient dans les instructions du bloc instruction1, l'exécution passe immédiatement à l'exécution d'instruction2 sinon l'exécution termine les instructions et passe ensuite au instruction3.
Exemple : Gestion des erreurs liées à une conversion de Type.

```
procedure TForm1.Button1Click(Sender: TObject);
var
   param1 : Double;
begin
   try
      param1 := StrToFloat(Edit1.Text);

      {suite des instructions}

   except
      on EconvertError do
         MessageDlg('Erreur : Vous devez entrer un réel'
            +#10#13+'Le séparateur décimal est : '+DecimalSeparator, mtError, [mbOk], 0);
   end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    {Autre instruction non sensible}
end;
```

Essayez cet exemple, en cas d'erreur de saisie vous aurez droit à un message d'erreur un peu plus clair que ceux distillés par Windows. Pour le vérifier, tapez votre chiffre en vous trompant dans le séparateur décimal (le point au lieu de la virgule et vice versa), Sans la gestion d'erreur vous saurez seulement que votre saisie n'est pas valide sans comprendre pourquoi car vous avez bien entré un nombre alors que grâce à la gestion des exceptions, vous aurez le droit à :




En plus, vous pouvez ajouter des instructions remettant votre programme dans un état stable (réinitialisation de variable par exemple).
L'exemple ci dessous est une des façons d'écrire la gestion des exceptions par **try**..**except**. Dans ce cas précis, nous savions ce qui pouvait provoquer une erreur dans le code protégé (une erreur de conversion) et nous n'avons traité que ce cas.
D'une manière plus générale, on peut considérer que la gestion d'exception peut intercepter des erreurs prévisibles et d'autre plus aléatoires (non prévues) et que l'on peut soit traiter les erreurs prévisibles soit les autres ou les deux (ce qui quand même préférable).
Quand on veut traiter une erreur prévisible, il faut savoir à quelle classe elle appartient, par exemple une erreur de conversion appartient à la classe EConvertError (on peut savoir ceci en consultant dans l'aide, le type d'erreur soulevée par une fonction particulière).
Le **try**..**except** pourra se présenter ainsi :

```
try
   {instructions}
except
    {instruction éventuelle commun à tous les cas possibles d'erreur}

    // Gestion des cas prévisibles d'erreur
    on Exception1 do InstructionTraitantErr1;
    on Exception2 do InstructionTraitantErr2;
    ....
    on Exception(n) do InstructionTraitantErr(n);
else
    InstructionTraitantLesCasNonPrevue;
end;
```

## XIX-C-2 - Listes non exhaustive de classe d'exception

Ceci est une liste incomplète des classes d'exception que vous pouvez être amené à utiliser.

- **EconvertError** : Erreur de conversion, vous essayez de convertir un type en un autre alors qu'ils sont incompatibles.
  Exemple : un string en integer (StrToInt) avec un string ne correspondant pas à un nombre entier.
- **EDivByZero** : Le programme a tenté de faire une division par zéro (pas Cool).
- **EFOpenError** : Le programme ne peut ouvrir un fichier spécifié (le fichier n'existe pas par exemple)
- **EInOutError** : Erreur d'entrée-sortie, sur le fichier spécifié


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

- **EReadError** : le programme tente de lire des données dans un flux mais ne peut lire le nombre spécifié d'octets.
- **ERangeError** : Débordement de taille. Le programme dépasse les borne d'un type entier ou les limites d'un tableau.
- **EAbort** : Exception spéciale car elle n'affiche pas de message d'erreur. On peut s'en servir pour annuler une tache en cours si une condition arrive (une erreur par exemple). Pour la déclencher un simple appel à **Abort**; suffit, une exception EAbort est alors générée. C'est un moyen simple de créer une exception personnalisée, il suffit alors de traiter l'exception **on** EAbort **do**.

Si vous n'arrivez pas à trouver la classe d'exception correspondant à votre code, tenter de provoquer l'erreur. Dans le message d'erreur, Delphi vous indiquera la classe d'exception (si elle existe), ce message permet aussi de tester si on a intercepté la bonne classe d'exception.

## XIX-D - Exception personnalisée

Toutes les exceptions dérivent de la Class exception, vous pouvez donc créer vos propres classes d'exception en héritant de cette classe. Ceci peut vous permettre de gérer votre programme par exception en supprimant tous les test des cas qui ne vous intéresse pas (exemple : la vérification qu'un diviseur est non nul).
En disant que toutes les exceptions dérivent de la classe Exception, je ne suis pas tout à fait exact. En fait n'importe quel objet peut être déclenché en tant qu'exception. Cependant, les gestionnaires d'exception standard ne gèrent que les exceptions dérivant de la classe exception.
Dans votre programme, si vous voulez déclarer une nouvelle classe d'exception, vous aurez à entrer le code suivant :

```
type
  MonException = class(Exception)
    [HelpContext : THelpContext; // Context dans l'aide]
    [Message : string; // Message d'erreur]
  public
    FonctionGerantErreur(); // Fonction à appeler en cas d'erreur
  end;

FonctionGerantErreur();
begin
  {instructions}
end;
```

Seule la première ligne est obligatoire. Si vous ne précisez pas le reste, la seule information disponible lors du déclenchement de votre exception sera son nom. FonctionGerantErreur est le nouveau gestionnaire de l'exception, dans cette fonction vous mettrez le code assurant la stabilité de votre application.

***Remarque :***
*Pour accéder au message ou à la Méthode d'une Exception (FonctionGerantErreur) tapez E.Message ou E.MaFonction. Dans ce cas le try..except doit s'écrire ainsi :*

```
try
  instruction
except
  on E : Exception do
    ShowMessage('Message : ' + E.Message);
    // Et/Ou
    E.MaFonction; // Permet de Centraliser le code gérant un type d'erreur
end;
```

Exemple : Exception personnalisée

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


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

et dans le code

```
if Valeur <> ValeurCorrect then
    raise EValeurIncorrect.Create('Valeur ne fait pas partie des valeurs autorisées');
```

La propriété Message de la classe Exception (y compris les classes dérivées) et l'affichage de message personnalisé dans le bloc except/end sont équivalents. Pour les classes d'exception déjà existantes, on préférera sans doute rendre le message plus explicite tandis que pour les classes personnalisées on aura recours à la propriété Message plutôt que d'indiquer à chaque fois le texte.
Il en est de même pour les instructions gérant l'erreur. On n'utilisera la Méthode de la classe que pour ceux personnalisées, évitant d'utiliser une fonction orpheline ou pire de réécrire à chaque fois le code.
Reportez-vous sur les classes pour plus de renseignement sur l'héritage.

## XIX-E - Raise

Protéger ainsi votre code, vous permet d'intercepter les messages d'erreur. Toutefois dans certaine situation, vous souhaiterez que le message d'erreur soit propagé pour qu'il soit intercepté par une autre gestion des exceptions.
Prenons l'exemple de l'assignation de fichier, plutôt que d'utiliser une variable de retour pour indiquer le résultat de l'opération, on pourrait transmettre le message d'erreur éventuel. A cet effet, la commande **raise** est à votre disposition.
**Raise** ne sert pas uniquement à propager un message d'erreur, on peut aussi s'en servir pour déclencher une exception (en générale pour déclencher une exception personnalisée).
Ci-dessous, un exemple de déclenchement d'exception personnalisée.

```
if Valeur <> ValeurCorrect then
    raise EValeurIncorrect.Create('Valeur ne fait pas partie des valeurs autorisées');
```

Exemple complet de l'utilisation de **raise** :

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

  MonException = class(Exception)
  public
     function GestErr():string;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function MonException.GestErr():string;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
begin
   if MessageDlg('La variable transmise est incorrect continuer avec la valeur par défaut',
 mtInformation, [mbYes, mbNo], 0) =
       mrYes then
   begin
      Result := 'très petit';
   end;
end;

function DoQuelqueChose(valeur : double):string;
begin
   // La fonction ne travaille que sur des nombres réels positifs
   if valeur < 0 then
   begin
      raise MonException.Create('Erreur: Travaille impossible !');
   end;

   // Travaille sur valeur complètement sans intérêt
   if valeur < 10 then
      result := 'très petit'
   else if valeur <= 50 then
      result := 'moitié de cent'
   else if valeur <= 100 then
      result := 'égal à cent'
   else
      result := 'très grand';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
   r : double;
   tmpstr : string;
begin
   try
      r := StrToFloat(Edit1.Text);
      tmpstr := DoQuelqueChose(r);
   except
    on E : MonException do
      tmpstr := E.GestErr;
    on EconvertError do
    begin
      ShowMessage('Erreur de Saisie : nombre attendu'
         +#10#13+'Séparteur décimal : '+DecimalSeparator);
      tmpstr := 'Mauvaise saisie';
    end;
    else
    begin
       ShowMessage('Erreur Inconnue');
       tmpstr := 'Invalid';
    end;
   end;
   ShowMessage('Résultat : '+tmpStr);
end;

end.
```

## XIX-F - Conclusion

Grâce aux exceptions apparues avec la programmation objet, le développeur a maintenant à sa disposition un outil efficace pour protéger son programme des aléas de l'informatique. J'espère que le présent tutorial a été pour vous une mine d'information et que désormais vous aborderez la gestion des exceptions avec sérénité.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

# XX - TFileStream

## XX-A - Introduction

Ce tutorial a pour but de vous présenter sommairement la classe TFileStream et son utilisation dans la lecture et l'écriture de fichier à travers des exemples. Avant de commencer, il est fortement conseillé de (re)lire les fichiers séquentiels dans le guide Delphi de Frédéric Beaulieu. En effet, la classe TFileStream permet de simplifier la manipulation de fichier séquentiel.

## XX-B - Lecture

L'exemple ci-dessous manipule un record contenant des champs de type différent pour montrer comment procéder. En fait, le seul type qui pose problème est le type (String, tableaux dynamiques) dont la taille n'est pas fixe. Le type énumération bien que ne posant pas de problème passe quand même par une subtilité, il est impossible d'écrire directement une variable de type énuméré. C'est pourquoi nous manipulons un entier issu de la conversion du type énuméré.
La première chose à faire, est d'ouvrir un flux sur le fichier. On utilise la même commande pour la lecture ou l'écriture en spécifiant les paramètres adéquats. La commande ci dessous, ouvre le fichier *Nom* en lecture et en mode partagé pour la lecture.

```
F := TFileStream.Create(Nom, fmOpenRead or fmShareDenyWrite);
```

En mode partagé en lecture seule, l'écriture par d'autre application est impossible tant que le flux est ouvert d'où l'obligation de fermer le flux une fois le travail fini. Par contre, il est accessible en lecture. Ceci est fait en libérant la ressource :

```
F.Free;
```

Remarquez que je n'ai pas utilisé de **Try**..**Finally** et que c'est un tort (mais vous aurez corrigé de vous-même). Voir la chapitre sur la gestion des exceptions si vous ne savez pas comment faire.
Pour commencer la lecture depuis le début (ce qui est préférable), un simple appel à

```
F.Position := 0;
```

Mettra les choses en ordre.
La lecture se fait par l'intermédiaire de la syntaxe suivante :

```
F.ReadBuffer(Mavariable, SizeOf(TMavariable));
```

Ou TMavariable indique la taille de MaVariable. En général, on transmet le type de la variable, ex integer s'il s'agit d'un entier. Mais pour certains types comme les strings il faut indiquer la taille de la donnée. Dans le cas des strings, il faut transmettre un pointeur, on utilise alors MaVariable[1].
Etudiez l'exemple, je pense qu'il est assez parlant.

```
type
    TEnumTest = (etPos1, etPos2, etPos3, etPos4);

    TMonRecord = record
       entier : integer;
       reel : double;
       enum : TEnumTest;
       chainelimit : string[100];
       chaine : string;
    end;
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
function OuvrirFichier(Nom : TFileName;var mrec : TMonRecord) : boolean;
var
   F : TFileStream;
   i : integer;
begin
    // Ouvre un fichier test
    F := nil;
    try

  // Ouverture d'un flux sur le fichier en lecture. Le fichier reste accessible en lecture seule pour d'autres ap
      F := TFileStream.Create(Nom, fmOpenRead or fmShareDenyWrite);
      F.Position := 0; // Début du flux
      if (pos('.zio',ExtractFileName(Nom))>0) then // Vérification du type du Fichier
      begin
         // Lecture du fichier
         while (F.position < F.Size) do // Tant que la fin du fichier n'est pas atteinte faire :
         begin
            with mrec do // voir TMonRecord pour savoir quelle type de donnée nous avons.
            begin
               // Lit la valeur et déplace la position en cours
               // La première valeur lue est un entier (le fichier a été enregistré ainsi)
               F.ReadBuffer(entier, SizeOf(integer));
               // Ensuite nous avons un réel
               F.ReadBuffer(reel, SizeOf(Double));
               // De nouveau un entier mais sa valeur n'est pas directement exploitable dans mrec.
               // On le convertit, ici il s'agit d'un type énuméré.
               F.ReadBuffer(i, SizeOf(integer));
               enum := TEnumTest(i);
               // On lit ensuite une Chaine de caractère dont la taille est limitée (string[taille]).
               F.ReadBuffer(chainelimit, SizeOf(Chainelimit));
               // On lit un entier correspondant à la taille de la chaine qui suit
               F.ReadBuffer(i, SizeOf(i));
               // Allocation d'assez d'espace dans la chaine pour la lecture
               SetLength(chaine, i);
               // Lecture de la chaine, on transmet un pointeur sur la chaine soit Chaine[1].
               F.ReadBuffer(chaine[1], i);
            end;
         end;
         Result := true;
      end
      else
      begin
         MessageDlg('Erreur ce n''est pas un fichier test.', mtError, [mbOk], 0);
         Result := false;
      end;
      F.Free;
    except
      on EInOutError do
      begin
        Result := False;
        F.Free;
        MessageDlg('Erreur d''E-S fichier.', mtError, [mbOk], 0);
      end;
      on EReadError do
      begin
         Result := False;
         F.Free;
         MessageDlg('Erreur de lecture sur le fichier.', mtError, [mbOk], 0);
      end;
      else
      begin
         Result := False;
         F.Free;
         MessageDlg('Erreur sur le fichier.', mtError, [mbOk], 0);
      end;
    end; // try
end;
```

Cet exemple lit le fichier et récupère tous les enregistrements contenue dans le fichier mais ne garde que le dernier car une seule variable est utilisé pour stocker le résultat. Je sais, c'est nul mais bon j'ai la flemme de mettre un tableau dynamique (ou alors le nombre d'enregistrement contenus dans le fichier doit être fixe) stockant des enregistrements


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

et d'ajouter chaque enregistrement à ce tableau en incrémentant l'index en cours. Ce qui d'ailleurs vous ferez un bon tutorial, à ce propos vous pouvez télécharger l'ébauche du programme pour l'étudier et le compléter (voir à la **XX-D Conclusion**).
Un point important maintenant, même si cet exemple ne manipule qu'un enregistrement, rien ne vous interdit de stocker d'autres valeurs dans le type de fichier. Vous pourriez par exemple enregistrer le nombre d'enregistrement qu'il contient, un commentaire ou l'âge du capitaine. L'essentiel est de savoir dans quel ordre les données sont agencées et leur taille. Passons donc à l'écriture.

## XX-C - Ecriture

Comme vous l'avez vu plus haut, le point délicat mis à part la manipulation de certains type de donné est l'agencement des données. Vous ne devez pas perdre de vue l'ordre dans lequel vous placez vos données dans le fichier, sinon il sera illisible. Le modus operandi de l'écriture est strictement le même que la lecture (si ça, c'est pas une bonne nouvelle). La seule différence est dans les paramètres d'ouverture du fichier et l'utilisation de la commande d'écriture à la place de lecture.

```
F := TFileStream.Create(Filetmp, fmCreate or fmShareExclusive);
```

Ouverture en écriture et en mode non partagé (lecture et écriture impossible par d'autre application)

```
.WriteBuffer(MaVariable, SizeOf(TMaVariable));
```

Ecriture de MaVariable en indiquant sa taille, comme pour la lecture en transmet en général le type de la variable.
N'oubliez pas de libérer la ressource une fois le travail terminé et avant d'essayer de manipuler le fichier (et oui, vous l'avez verrouillé).
Voir l'exemple ci-dessous.

```
function EnregistreFichier(Nom : TFileName;mrec : TMonRecord):boolean;
var
   F: TFileStream;
   Filetmp : TFileName;
   Chem_tmp, Nom_tmp : string;
   i : integer;
begin
   // Enregistrement d'un fichier
   Chem_tmp := ExtractFilePath(Nom);
   Nom_tmp := '~temp.zio';
   Filetmp := TFileName(Chem_tmp+'\'+Nom_tmp);
   F := nil;
   try
      DeleteFile(Filetmp);
      // Ouverture d'un flux sur le fichier, en création et de façon exclusive
      F := TFileStream.Create(Filetmp, fmCreate or fmShareExclusive);
      F.Position := 0; // Début du flux
      // Ecriture du fichier
      with mrec do
      begin
        // On écrit le champ entier en premier
         F.WriteBuffer(entier, SizeOf(integer));
        // Puis le champ Réel
         F.WriteBuffer(reel, SizeOf( Double));

  // On convertit le champ de type énuméré en entier, on ne peut pas écrire directement une variable de type enum
         i := Ord(enum);
        // On écrit l'entier correspondant au champ énuméré
         F.WriteBuffer(i, SizeOf(integer));
        // On écrit la chaîne à taille fixe, aucune difficulté.
         F.WriteBuffer(chainelimit, SizeOf(chainelimit));

  // Pour la chaîne de type string, il nous faut indiquer la taille de la chaîne sinon nous ne pourrions plus la 
         i := Length(chaine);
        // On écrit donc la taille de la chaîne
         F.WriteBuffer(i, SizeOf(i));
```


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
         // Puis la chaîne en indiquant sa taille et en transmettant un pointeur.
          F.WriteBuffer(chaine[1], i);
       end;
       F.Free;
       // detruit Nom et renome temp.zio en Nom
       DeleteFile(Nom);
       if RenameFile(Filetmp, ExtractFileName(Nom)) then
           Result := true
       else
           Result := false;
   except
       on EInOutError do
       begin
         Result := False;
         F.Free;
         MessageDlg('Erreur d''E-S fichier : Fichier non enregistré.', mtError, [mbOk], 0);
       end;
       on EWriteError do
       begin
          Result := False;
          F.Free;
          MessageDlg('Erreur d''ecriture dans le fichier. Fichier non enregistré', mtError, [mbOk], 0);
       end;
       else
       begin
          Result := False;
          F.Free;
          MessageDlg('Erreur sur le fichier. Fichier non enregistré.', mtError, [mbOk], 0);
       end;
   end; // try
 end;
```

Comme pour la lecture, je me suis limité à l'écriture d'un seul enregistrement. Si vous avez réalisé une fonction de lecture lisant (essayant de lire, le fichier peut très bien ne contenir qu'un seul enregistrement) plusieurs enregistrements dans le fichier, votre fonction d'écriture devrait permettent d'écrire plusieurs enregistrements également. En utilisant un tableau d'enregistrement, il suffit de le parcourir et d'écrire chaque enregistrement, les uns à la suite des autres. Vous pouvez également indiquer d'autres informations dans le fichier.

## XX-D - Conclusion

Deux points sont capitaux dans les fichiers séquentiels. Le premier si une variable est de taille dynamique, alors la taille devra faire partie des informations enregistrées dans le fichier. Le second, les fonctions de lecture et d'écriture doivent être symétriques, les données attendues par ses fonctions doivent l'être dans le même ordre. Si vous écrivez le nom d'une personne, lors de la lecture vous lirez le nom de cette personne au même moment.
Pour ceux qui voudront compléter le programme pour qu'il puisse lire et écrire plusieurs enregistrements ou pour voir le code 'complet', vous pouvez **le télécharger en cliquant ici**.


Copyright © 2008 - Frédéric Beaulieu. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.