Faculté de Sciences Économiques et de Gestion

# Programmation sous Delphi

Maîtrise d'Économétrie
Année 1999-2000

Jérôme Darmont
*jerome.darmont@univ-lyon2.fr*
*http://eric.univ-lyon2.fr/~jdarmont/*

# Table des matières

# I. Introduction

Delphi est un environnement de développement de type RAD (*Rapid Application Development*) basé sur le langage Pascal. Il permet de réaliser rapidement et simplement des applications Windows.

Cette rapidité et cette simplicité de développement sont dues à une conception visuelle de l'application. Delphi propose un ensemble très complet de *composants visuels* prêts à l'emploi incluant la quasi-totalité des composants Windows (boutons, boîtes de dialogue, menus, barres d'outils…) ainsi que des experts permettant de créer facilement divers types d'applications et de librairies.

Pour maîtriser le développement d'une application sous Delphi, il est indispensable d'aborder les trois sujets suivants :

- le langage Pascal et la programmation orientée objet ;
- l'*Environnement de Développement Intégré* (EDI) de Delphi ;
- les objets de Delphi et la hiérarchie de classe de sa bibliothèque.

Nous complèterons cette approche par la connexion aux bases de données avec Delphi.

NB : Ce support de cours ne se veut nullement exhaustif. Ce n'est qu'une référence de base. Le lecteur est encouragé à compléter sa lecture avec les références citées à la fin du document… et à s'entraîner à programmer sous Delphi ! D'autre part, ce support ne traite que de la version 3 de Delphi.

# II. Le langage Pascal


Le langage de programmation Pascal a été conçu en 1968 par Niklaus Wirth.

## 1. Éléments du langage

### *a) Identificateurs et instructions*

Un identificateur est un nom permettant au compilateur d'identifier un objet donné. Les noms de variables, par exemple, sont des identificateurs. Un identificateur doit commencer par une lettre. Les caractères suivants peuvent être des lettres, des chiffres ou le caractère _. Majuscules et minuscules ne sont pas différenciées. Seuls les 63 premiers caractères d'un identificateur sont pris en considération par le compilateur.

Identificateur valide : `Ma_variable01` Identificateur invalide : `9variable`

Outre les variables, constantes, etc., un programme Pascal contient des mots réservés que l'utilisateur ne peut pas employer. Ce groupe d'identificateurs particuliers correspond aux composants du langage Pascal.

| | | | | |
|---|---|---|---|---|
| And | Else | In | Or | To |
| Asm | End | Inherited | Packed | Try |
| Array | Except | Inline | Procedure | Type |
| Begin | Exports | Interface | Program | Unit |
| Case | File | Label | Record | Until |
| Const | Finally | Library | Repeat | Uses |
| Constructor | For | Mod | Set | Var |
| Destructor | Function | Nil | Shl | While |
| Div | Goto* | Not | Shr | With |
| Do | If | Object | String | Xor |
| Downto | Implementation | Of | Then | |

*Liste des mots réservés de Pascal*

### *b) Opérateurs*

Affectation

Ex. `resultat:=100;`

Opérateurs arithmétiques

- Multiplication : `*`
- Division entière : `div`

* Ce n'est pas parce que le goto existe en Pascal qu'il faut l'utiliser !

- Division : `/`
- Modulo : `mod`
- Addition : `+`
- Soustraction : `-`

Opérateurs logiques

- Et logique : `and`
- Ou logique : `or`
- Ou exclusif : `xor`
- Négation : `not`

Opérateurs de relation

- Égal : `=`
- Différent : `<>`
- Supérieur/Supérieur ou égal : `> >=`
- Inférieur/Inférieur ou égal : `< <=`
- Appartenance à un ensemble : `in`

*c) Commentaires*

`(* … *)` ou { … }

NB : Il est possible d'imbriquer ces deux types de commentaires.

## 2. Types de données

*a) Types prédéfinis*

| Types entiers | Domaine |
|---|---|
| `Byte` | 0..255 |
| `Shortint` | -128..127 |
| `Integer` | -32768..32767 |
| `Word` | 0..65535 |
| `Longint` | -2147483648..2147483647 |

| Types réels | Domaine |
|---|---|
| `Single` | $1{,}5.10^{-45}..3{,}4.10^{38}$ |
| `Real` | $2{,}9.10^{-39}..1{,}7.10^{38}$ |
| `Double` | $5{,}0.10^{-324}..1{,}7.10^{308}$ |
| `Extended` | $3{,}4.10^{-4951}..1{,}1.10^{4932}$ |

| Type booléen | Domaine |
|---|---|
| `Boolean` | True \| False |

| Types caractères | Domaine |
|---|---|
| `Char` | Caractère alphanumérique |
| `String[n]` | Chaîne de n caractères (n = 255 au maximum) |
| `String` | Chaîne de 255 caractères |

| Types tableaux |
|---|
| `Array[imin..imax, …] of <type>` |

*b) Types personnalisés*

La déclaration d'un type utilisateur s'effectue dans une clause `type`.

Énumérations

Ex. `Type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);`

Intervalles

Ex.
```
Type chiffre = 0..9;
     jour_ouvrable = lundi..vendredi;
```

Ensembles

Ex.
```
Type des_entiers = set of integer;
{déclaration de la variable ens de type des_entiers}
ens:=[1,2,3];
ens:=[]; {ensemble vide}
```

Opérations ensemblistes : `+` union
`*` intersection
`-` différence

Tableaux

Ex.
```
Type tab_reels = array[1..100] of real; {tableau de 100 réels}
     matrice = array[1..4,1..4] of integer; {matrice 4x4}
{Déclaration des variables t de type tab_reels et de m de type matrice}
t[1]:=1.5;
m[1,3]:=10;
```

Enregistrements

Ex.
```
Type personne = record
       nom: string;
       prenom: string;
       age: byte;
     end;
```

```
{Déclaration de la variable p de type Personne}
p.nom:='Darmont';
with p do
  begin
    prenom:='Jérôme';
    age=:27;
  end;
```

## 3. Squelette d'un programme Pascal

### *a) Squelette*

```
Program Nom_du_programme;

Uses {unités}

Const
    {Déclaration de constantes}

Type
    {Déclaration de types personnalisés}

Var
    {Déclaration de variables}

{Procédures et fonctions}

Begin
    {Bloc principal du programme}
End.
```

### *b) Notion de bloc*

Un bloc est une portion de code délimitée par le couple d'instructions `begin..end`.

### *c) Déclaration des constantes*

Ex.
```
Const PI=3.1416;
      NOM='Université Lyon 2';
```

### *d) Déclaration des variables*

Ex.
```
numero, i: integer; {deux variables de même type}
tab: array[1..10] of real;
phrase: string;
{idem avec des types personnalisés}
jour: jour_ouvrable;
ens: des_entiers;
m: matrice;
```

## 4. Entrées/sorties

*a) Lecture au clavier*

Ex. `Readln (une_variable);`

*b) Écriture à l'écran*

Ex.
```
Write ('Coucou ! ');
Writeln ('La valeur de la variable est : ', une_variable);
```

NB : Ces instructions sont définies dans l'unité Crt de Pascal, qu'il faut appeler en début de programme par la commande `Uses Crt ;`.

## 5. Structures de contrôle

*a) Branchements conditionnels*

If…then…else

```
If {condition} then {Instruction si condition vérifiée}
Else {Instruction si condition non vérifiée};
```

NB : La partie else est optionnelle.

Ex.
```
If a>=0 then writeln('A est positif');
If ville='Lyon' then bool_lyon:=true
Else bool_lyon:=false;
If a>b then
   If a<c then writeln('B < A < C')
   Else writeln('A > B');
```

Case…of

```
Case {variable} of
  {valeur1}: {Instruction};
  {valeur2}: {Instruction};
  …
  else {Instruction par défaut};
end ;
```

NB : La partie else est optionnelle.

Ex.
```
Case a of
     0: writeln('zéro');
     6: writeln('six');
     9: writeln('neuf');
```

```
end ;
Case ch of
  'A'..'Z': writeln('Majuscule');
  'a'..'z': writeln('Minuscule');
  '0'..'9': writeln('Chiffre');
  else writeln('Autre caractère');
end;
```

*b) Boucles*

While…do

Exécute des instructions tant qu'une condition est vérifiée.

```
While {condition} do {instructions}
```

Ex.
```
i:=1;
While i<=10 do
  Begin
    Write(i,' ');
    i:=i+1;
  End;
Writeln('*');
```

Repeat…until

Exécute des instructions jusqu'à ce qu'une condition soit vérifiée.

```
Repeat {instructions} until {condition};
```

Ex.
```
i:=1;
Repeat
  Write(i,' ');
  i:=i+1;
Until i=10;
Writeln('*');
```

NB : La boucle Repeat…until ne nécessite pas de bloc begin…end pour exécuter plusieurs instructions.

For…to|downto…do

Exécute des instructions un nombre fini de fois suivant un ordre croissant ou décroissant.

```
For {variable}:={valeur min} to {valeur max} do {instructions}
For {variable}:={valeur max} downto {valeur min} do {instructions}
```

Ex.
```
For i:=1 to 10 do Write(i,' ');
Writeln('*');
```

Choix d'une boucle

| *Nombre d'itérations connu* | *Au moins une exécution* | *Test avant exécution* |
|---|---|---|
| For | Repeat | While |

## 6. Procédures et fonctions

Les procédures et les fonctions permettent de diviser un programme en unités plus petites (des sous-programmes) pour mieux le structurer. La seule différence entre procédure et fonction est que cette dernière renvoie une valeur au programme ou au sous-programme appelant.

### *a) Squelette de déclaration*

```
Procedure NomP({paramètres});

Type
 {Déclaration de types locaux}

Const
 {Déclaration de constantes locales}

Var
 {Déclaration de variables locales}

{procédures | fonctions locales}

Begin
 {Corps de la procédure}
End;
```

```
Function NomF({paramètres}): {type};

Type
 {Déclaration de types locaux}

Const
 {Déclaration de constantes locales}

Var
 {Déclaration de variables locales}

{procédures | fonctions locales}

Begin
 {Corps de la fonction}
 NomF:={valeur de retour};
End;
```

### *b) Exemples de sous-programmes*

```
Procedure PN_P(nombre: real);

Const N=25;

Var i: integer;
    res: real;

Begin
  res:=1;
  For i:=1 to N do res:=res*nombre;
  Writeln(res);
End;
```

```
Function PN_F(nombre: real):real;

Const N=25;

Var i: integer;
    res: real;

Begin
  res:=1;
  For i:=1 to N do res:=res*nombre;
  PN_F:=res;
End;
```

*c) Mode de passage des paramètres*

Le passage de paramètres fournit des valeurs à un sous-programme lors de son appel (cf. exemples ci-dessus). Ce passage de paramètres peut s'effectuer :

- en faisant une copie des valeurs à passer (*passage par valeur*), ce qui permet de modifier les copies locales sans influencer la valeur initiale (*paramètres en entrée*) ;
- en fournissant l'adresse des valeurs (*passage par adresse* ou *par référence*), ce qui permet de modifier la valeur initiale (*paramètres en sortie ou en entrée/sortie*).

Dans les exemples ci-dessus, les paramètres sont passés par valeur.

Exemple de passage par adresse

```
Procedure PN_P2(var nombre: real);

Const N=25;

Var i: integer;
    res: real;

Begin
  res:=1;
  For i:=1 to N do res:=res*nombre;
  nombre:=res;
End;
```

NB : Un appel de procédure du type `PN_P2(n);` aura le même effet qu'un appel de fonction du type `n:=PN_F(n);`.

*d) Visibilité des variables et des sous-programmes*

Une *variable globale* est utilisable par tous les sous-programmes du programme. Une *variable locale* n'est utilisable que dans le sous-programme où elle est déclarée.

Les procédures et les fonctions sont soumises aux mêmes règles que les variables.

## 7. Les unités

Une unité est un fichier séparé pouvant contenir des constantes, des types, des variables et des sous-programmes disponibles pour la construction d'autres applications. L'utilisation des unités permet de partager des données et des sous-programmes entre plusieurs applications.

*a) Squelette d'une unité*

```
Unit Nom_unite;

Interface
    {Déclarations publiques}

Implementation
    {Déclarations privées}
    {Corps des procédures et des fonctions}

Initialization
    {Code d'initialisation}

End.
```

*b) Visibilité d'une unité*

En-tête

C'est le nom de l'unité (8 caractères maxi). Il figure dans la partie `uses` d'un programme ou d'une unité qui utilise l'unité.

NB : Éviter les références croisées entre unités (ex. l'unité A inclut l'unité B et vice-versa).

Interface

Tout ce qui est placé ici est visible pour toute entité utilisant l'unité.

Implémentation

Cette partie définit les procédures et les fonctions déclarées dans la partie interface. On peut également y définir des types, constantes, variables et sous programmes locaux.

Initialisation

Cette partie permet d'initialiser les variables de l'unité si besoin est et d'effectuer des traitements avant de redonner la main au programme principal.

*c) Exemple d'unité*

```
Unit stats;

Interface
    Const N=100000;
    Type echantillon = array[1..N] of real;
    Function moyenne(ech: echantillon): real;

Implementation
    Function moyenne; {Pas besoin de répéter les paramètres}
        Var i: integer;
            s: real;
```

```
        Begin
            s:=0;
            for i:=1 to N do s:=s+ech[i];
            moyenne:=s/N;
        End ;

End.
```

Programme appelant :

```
Program Calcul_Moyenne;

Uses stats;

Var ech: echantillon;
    i: integer;
    moy: real;

Begin
    For i:=1 to N do readln(ech[i]);
    {Je ne voudrais pas être l'opérateur de saisie !}
    moy:=moyenne(ech);
    Writeln(moy);
End.
```

# 8. La programmation orientée objet

## *a) Objectifs*

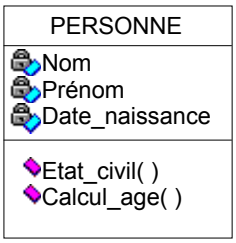
- Lier les données et les fonctions qui les manipulent afin d'éviter des accès aux données par des fonctions non autorisées.
- Obtenir une meilleure abstraction en cachant l'implémentation des techniques utilisées et en ne rendant visible que des points d'entrée. Ainsi, si l'implémentation change, le code utilisateur n'est pas affecté.
- Réutiliser l'existant dans un souci de productivité.
- Traiter les erreurs localement au niveau des objets sans que cela ne perturbe les autres parties du programme.
- Faciliter la maintenance.

## *b) Concepts*

Objet

En langage objet, tout est objet ! Un objet contient des données dites *données membres* ou *attributs* de l'objet et des procédures ou fonctions dites *méthodes* de l'objet.

Ex.



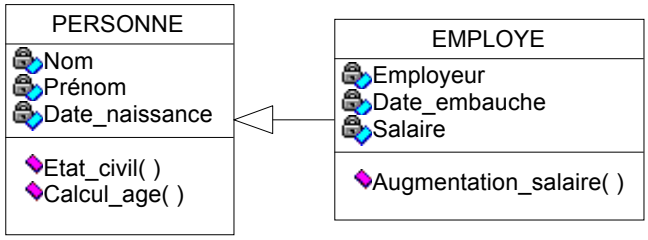

Encapsulation

C'est le mécanisme consistant à lier les attributs et les méthodes au sein d'une même structure.

Héritage

L'héritage permet à un objet de récupérer les caractéristiques d'un autre objet (attributs et méthodes) et de lui ajouter de nouvelles caractéristiques.

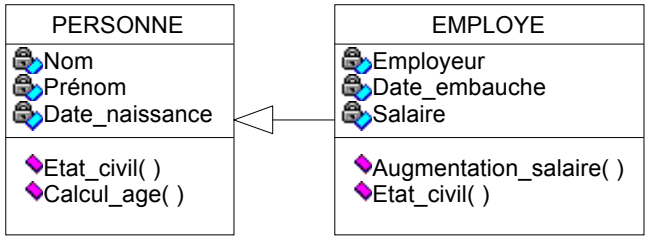
Ex. Un employé *est une* personne.




Polymorphisme

Le polymorphisme permet d'attribuer à différents objets une méthode portant le même nom afin d'exprimer la même action, même si l'implémentation de la méthode diffère complètement.

Ex. Surcharge de la méthode Etat_civil().




*c) Déclaration d'objets en Pascal*

Classe

Un objet est une *instance* d'une *classe* (ex. Jérôme Darmont né le… est une instance de la classe Personne). La classe est un type de données représentant un objet.

Ex.
```
Type Personne = class
       End;
```

Attributs

Ex.
```
Type Personne = class
         nom: string;
         prenom: string;
         date_naissance: string;
       End;
```

Méthodes

Les méthodes sont des procédures et des fonctions définies pour une classe. Elles ont un accès complet à tous les attributs de la classe. Il est recommandé de ne pas laisser un utilisateur de la classe manipuler directement ses attributs, mais d'utiliser une méthode pour cela.

Ex.
```
Type Personne = class
         nom: string;
         prenom: string;
         date_naissance: string;
         procedure etat_civil();
         function calcul_age():byte;
       End;
```

Le corps des méthodes est défini à l'extérieur de la classe. Le nom de chaque méthode doit donc être préfixé du nom de la classe à laquelle il appartient.

Ex.
```
Procedure Personne.etat_civil();
    Begin
      Writeln(nom+' '+prenom);
    End;
  Function Personne.calcul_age():byte;
    Var res: byte;
    Begin
      {Calcul de l'âge, résultat dans la variable res}
      calcul_age:=res;
    End;
```

*d) Protection des données*

Si une classe est utilisée par quelqu'un d'autre que son auteur, rien n'empêche ce dernier d'accéder directement à un attribut de la classe, ce qui la fragilise. En effet, si la méthode associée à l'attribut en question effectue des tests avant de modifier sa valeur (ex. vérification qu'une date de naissance est inférieure à la date du jour), ne pas l'utiliser peut provoquer des erreurs ultérieurement.

Pour se prémunir contre des accès non désirés, il est possible d'associer un niveau de visibilité aux attributs et aux méthodes d'une classe.

| Protection | Description |
|---|---|
| Private | Attributs ou méthodes accessibles uniquement par les méthodes de la classe |
| Protected | Attributs ou méthodes accessibles uniquement par les méthodes de la classe |

| | et de ses sous-classes |
|---|---|
| Public | Attributs ou méthodes accessibles par toute procédure, même externe |

Ex.
```
Type Personne = class
          private
            nom: string;
            prenom: string;
            date_naissance: string;
          public
            procedure etat_civil();
            function calcul_age():byte;
        End;
```

*e) L'héritage*

Déclaration de la classe Employé, sous-classe de Personne

```
Type Employe = class(Personne)
       private
         employeur: string;
         date_embauche: string;
         salaire: real;
       public
         procedure augmentation_salaire();
         procedure etat_civil();
     End;
```

Déclaration d'instances des classes Personne et Employé

```
Var p: Personne;
    e: Employe;

Begin
  p.etat_civil();
  e.etat_civil();
  p:=e; {NB : l'affectation inverse n'est pas possible}
End.
```

NB : La classe Employé est dite *compatible* avec la classe Personne.

*f) Le polymorphisme*

Le polymorphisme permet de manipuler les sous-classes via la classe de base.

Ex.
```
Program Polymorphisme;
   Uses Upers; {Unité contenant les classes Personne et Employé}
   Var p: Personne;
       e: Employe;
   Begin
     p:=e;
     p.etat_civil();
   End.
```

Problème : Quelle est la méthode `etat_civil` appelée ? Celle de la classe Personne ou celle de la classe Employé ?

Pour résoudre ce problème, il faut faire appel au polymorphisme en déclarant cette méthode comme étant *virtuelle* (mot clé `virtual`). Le compilateur rajoute alors du code au programme exécutable afin de pouvoir appeler dynamiquement la méthode appropriée à l'objet manipulé (ici, un employé). De plus, il faut indiquer que la méthode `etat_civil` de la classe Employé *surcharge* celle de la classe Personne (mot clé `override`).

Ex.
```
Type Personne = class
       private
         nom: string;
         prenom: string;
         date_naissance: string;
       public
         procedure etat_civil(); virtual;
         function calcul_age():byte;
     End;

Type Employe = class(Personne)
       private
         employeur: string;
         date_embauche: string;
         salaire: real;
       public
         procedure augmentation_salaire();
         procedure etat_civil(); override;
     End;
```

*g) Constructeurs et destructeurs*

Constructeurs et destructeurs permettent respectivement d'allouer et d'initialiser les attributs d'un objet ou de libérer la mémoire allouée à cet objet.

- Ils doivent être définis dans une section `public`.
- Ils peuvent être plusieurs dans une classe à condition de ne pas porter le même nom.
- Ils peuvent être virtuels.

NB : Par convention, les constructeurs sont nommés *Create* et les destructeurs *Destroy* dans les classes Delphi.

Ex.
```
Type Personne = class
       private
         nom: string;
         prenom: string;
         date_naissance: string;
       public
         constructor Create(); virtual;
         constructor Init(n, p, d: string);
         destructor Destroy; virtual;
         procedure etat_civil(); virtual;
         function calcul_age():byte;
     End;
```

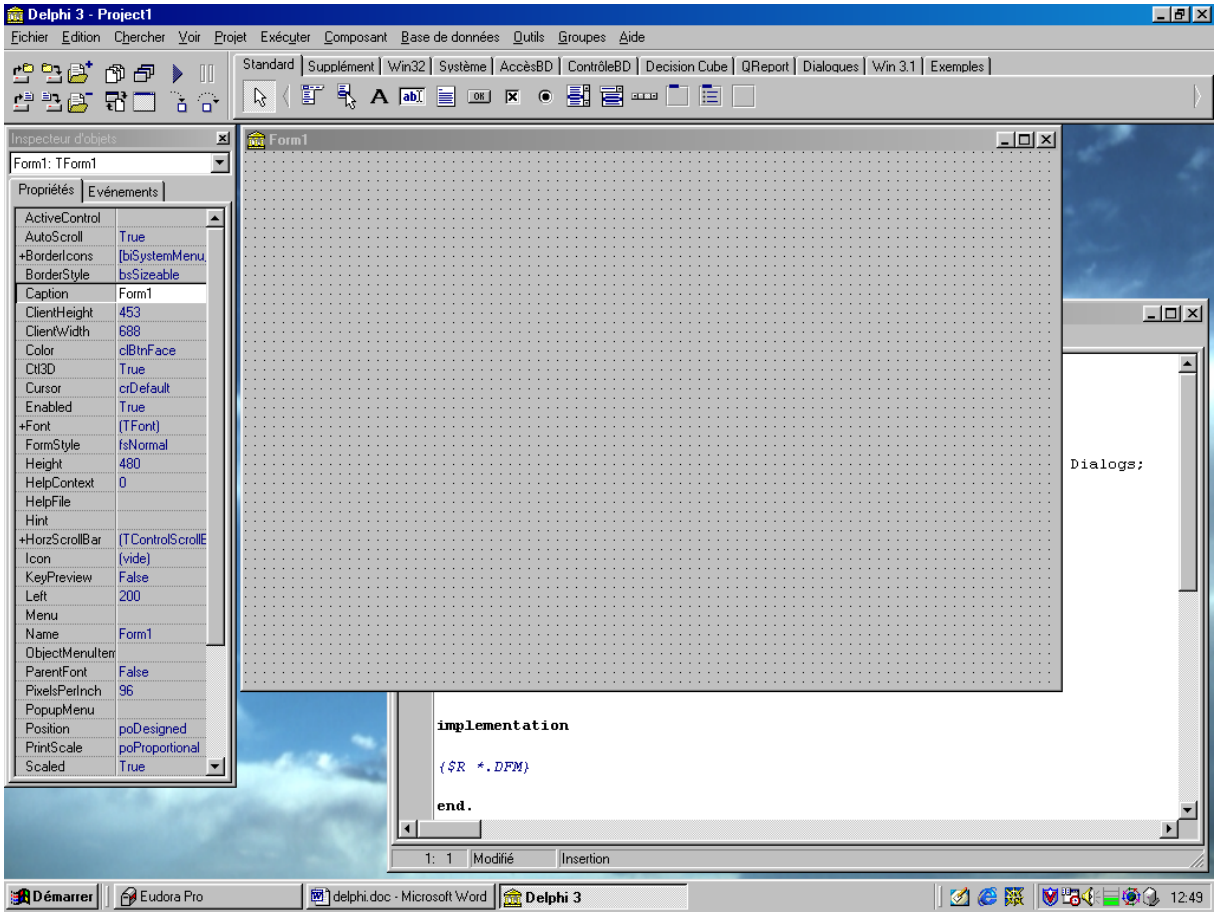
```
Procedure Personne.Init;
Begin
  nom:=n;
  prenom:=p;
  date:=d;
End;

{Utilisation}
p.Init('Darmont','Jérôme','15/01/1972');
p.Destroy;
```

# III. L'EDI de Delphi

## 1. L'interface de Delphi

La figure ci-dessous représente l'interface typique de Delphi. Elle est composée de :

- la barre de menus (en haut),
- la barre d'icônes (à gauche sous la barre de menus),
- la palette de composants (à droite sous la barre de menus),
- le concepteur de fiche (au centre),
- l'éditeur de code (au centre sous le concepteur de fiche),
- l'inspecteur d'objets (à gauche).




### *a) Conception de fiches : la palette des composants*

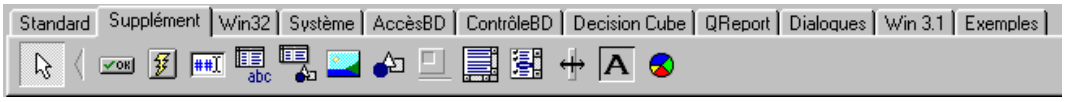
Une *fiche* constitue l'interface (ou une partie de l'interface) d'une application. Pour concevoir une fiche, il suffit d'y insérer des *contrôles* (ressources Windows prêtes à l'emploi : boutons de commande, listes, menus…) listés dans la palette des composants. Un clic sur le contrôle,

puis un autre sur la fiche cible suffisent (un double clic insère le composant au milieu de la fiche active). La palette des composants réunit plusieurs volets. Les principaux sont listés ci-dessous.

Composants standards


| Contrôle | Nom Pascal | Description |
|---|---|---|
| Menu principal | `MainMenu` | Barre de menus |
| Menu surgissant | `PopupMenu` | Menu contextuel |
| Étiquette | `Label` | Zone d'affichage de texte non modifiable |
| Édition | `Edit` | Boîte d'édition permettant la saisie ou la modification d'une ligne de texte |
| Mémo | `Memo` | Boîte d'édition permettant la saisie ou la modification d'un texte de plusieurs lignes |
| Bouton | `Button` | Bouton de commande simple |
| Case à cocher | `CheckBox` | Sélection de choix entre plusieurs |
| Bouton radio | `RadioButton` | Sélection d'un choix entre plusieurs |
| Boîte de liste simple | `ListBox` | Liste [déroulante] d'éléments |
| Boîte de liste combinée | `ComboBox` | Liste déroulante d'éléments avec possibilité d'édition sans une boîte d'édition |
| Barre de défilement | `ScrollBar` | Ascenseur vertical ou horizontal |
| Boîte de groupe | `GroupBox` | Cadre conteneur de contrôles |
| Groupe de boutons radio | `RadioGroup` | Cadre conteneur de boutons radio |
| Volet | `Panel` | Volet conteneur de contrôles |

Composants supplémentaires




| Contrôle | Nom Pascal | Description |
|---|---|---|
| Bouton bitmap | `BitBtn` | Bouton de commande avec image bitmap |
| Turbo bouton | `SpeedButton` | Icône dans une barre d'outils |
| Masque de saisie | `MaskEdit` | Boîte d'édition permettant des saisies formatées |
| Grille de chaînes | `StringGrid` | Tableau d'affichage de chaînes |
| Grille d'affichage | `DrawGrid` | Tableau d'affichage de données |
| Image | `Image` | Zone d'affichage d'une image bitmap, d'une icône ou d'un métafichier Windows |
| Forme | `Shape` | Forme géométrique (ellipse ou rectangle) |
| Biseau | `Bevel` | Ligne ou rectangle 3D |
| Boîte de défilement | `ScrollBox` | Conteneur de composant possédant des barres de défilement |
| Boîte de liste à cocher | `CheckListBox` | Liste [déroulante] d'éléments à cocher |
| Séparateur | `Splitter` | Séparateur mobile |

| | | |
|---|---|---|
| Texte statique | `StaticText` | Étiquette contenant un descripteur de fenêtre |
| Graphique | `Chart` | Graphique type Excel |

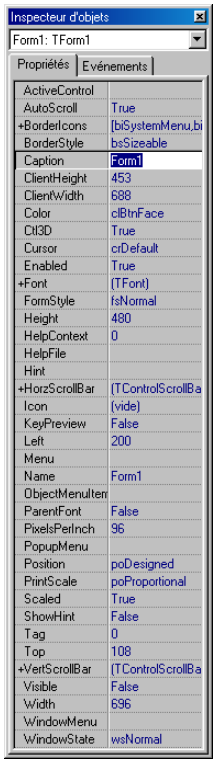
Composants boîtes de dialogue


| Boîte de dialogue | Nom Pascal | Description |
|---|---|---|
| Ouvrir | `OpenDialog` | Ouverture d'un fichier |
| Enregistrer sous | `SaveDialog` | Sauvegarde d'un fichier |
| Ouvrir image | `OpenPicture` | Ouverture d'un fichier image |
| Enregistrer image | `SavePicture` | Sauvegarde d'un fichier image |
| Fontes | `FontDialog` | Sélection d'une police de caractère |
| Couleurs | `ColorDialog` | Sélection d'une couleur dans la palette |
| Imprimer | `PrintDialog` | Impression d'un document |
| Configuration impression | `PrinterSetupDialog` | Paramétrage de l'imprimante |
| Recherche | `FindDialog` | Recherche d'une donnée |
| Remplacer | `ReplaceDialog` | Recherche et remplacement d'une donnée |

### *b) L'inspecteur d'objets*

Cet outil est dédié à la gestion des composants. La fenêtre de l'inspecteur contient deux volets :

- la liste des *propriétés* (attributs) du composant courant,
- la liste des *événements* associés au composant courant.




Propriétés

Les noms des propriétés sont placés dans la colonne de gauche (dans l'ordre alphabétique) et les valeurs sur la ligne correspondante à droite.

Les propriétés dont le nom est précédé d'un + ont plusieurs niveaux imbriqués (ex. *Font*). Lorsqu'une propriété de ce type est « déroulée », le signe – apparaît à la place du signe +.

Pour donner une valeur à une propriété, le plus simple est de remplacer sa valeur par défaut par une nouvelle valeur dans la boîte d'édition prévue à cet effet.

La propriété *Name* est particulièrement importante car elle permet d'accéder au composant depuis les programmes. Par défaut, Delphi lui confère une valeur peu explicite (ex. Form1, Button1…). Il est préférable d'utiliser des noms plus « parlants ».

Les propriétés visibles dans l'inspecteur sont modifiables lors de la phase de conception. D'autres propriétés sont uniquement accessibles lors de l'exécution, grâce à du code source.

Événements

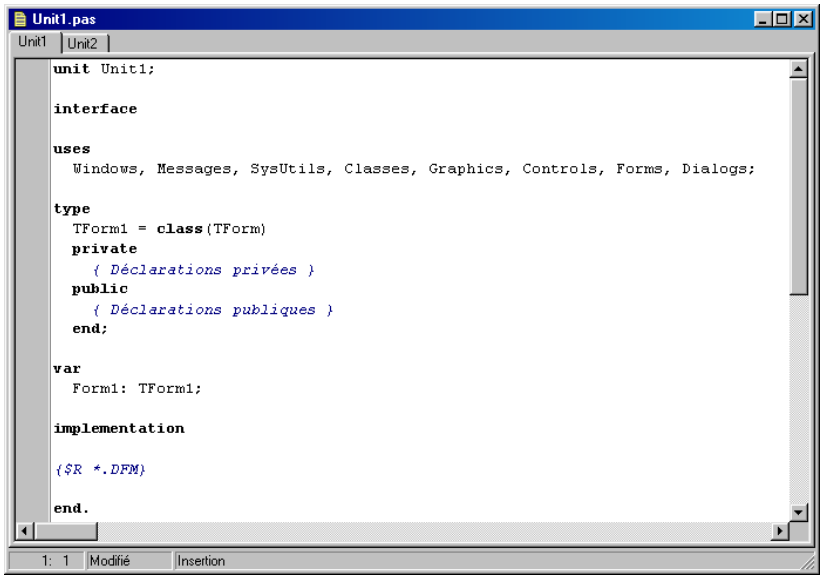

La colonne de gauche contient l'ensemble des événements associés à un composant donné. Ex. *OnClick*, commun à un grand nombre de composants, qui est activé lorsque l'utilisateur clique sur le composant.

La colonne de droite consiste en une série de boîtes de listes combinées permettant d'associer un gestionnaire d'événements à l'événement correspondant (i.e., un ensemble d'instructions Pascal exécutées lorsque l'événement survient).

Sur un double clic sur une de ces boîtes, l'éditeur de code s'ouvre et le curseur se positionne à l'intérieur du gestionnaire d'événements (une procédure).

### *c) L'éditeur de code*

Les fichiers de code source composant un *projet* sont rassemblés dans l'éditeur de code. À chaque fiche est associée une unité identifiée par un onglet situé en haut de la fenêtre de l'éditeur. L'éditeur permet de modifier le code Pascal de ces unités.




### *d) Les menus*

- Menu Fichier : création, ouverture, enregistrement…
- Menu Édition : annulation, copier, coller…
- Menu Chercher : chercher, remplacer…
- Menu Voir : gestionnaire de projets, inspecteur d'objets, déboguage…
- Menu Projet : compilation de code source, options de compilation…
- Menu Exécuter : exécution de programmes, déboguage…

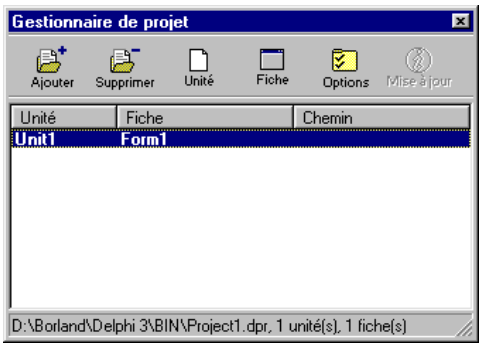
- Menu Composant : création de nouveau composant, configuration de la palette de composants…
- Menu Bases de donnée
- Menu Outils : options, utilitaires…
- Menu Aide : indispensable ! à utiliser sans modération…

## 2. Projets et programmes

L'ensemble des éléments d'un programme exécutable ou d'une librairie DLL (*Dynamic Link Library*) se contrôle à partir d'un *projet*. Concrètement, un projet est un fichier binaire d'extension *.dpr* contenant des liens vers tous les éléments du projet (fiches, unités, etc.), eux-mêmes stockés dans des fichiers séparés. Pour gérer ces fichiers, Delphi met à la disposition des utilisateurs un outil : le *gestionnaire de projets*.

### *a) Le gestionnaire de projets*

Le gestionnaire de projets se présente comme une fenêtre composée de trois colonnes et d'une barre d'outils.




- La colonne de gauche liste les unités du projet.
- La colonne du milieu donne la liste des fiches. Toute fiche est associée à une unité, mais la réciproque est fausse.
- La colonne de droite indique le chemin d'accès au fichier de l'unité quand celle-ci ne se trouve pas dans le même répertoire que le projet lui-même.
- La barre d'outils permet d'accéder rapidement à des fonctionnalités également accessibles par la barre de menus principale.

Fichiers d'un projet

| Fichier | Extension | Contenu |
|---|---|---|
| Projet | `.dpr` | Code d'initialisation du programme |

| | | |
|---|---|---|
| Code source | `.pas` | Code source des procédures, fonctions, fiches, DLL ou composants (explicitement implémenté par l'utilisateur ou automatiquement généré par Delphi) |
| Fiches | `.dfm` | Caractéristiques et propriétés des composants sous forme binaire |
| Options du projet | `.opt` | Fichier texte contenant les paramètres de configuration du projet |
| Ressources | `.res` | Fichier binaire contenant les ressources Windows utilisées par l'application (ex. l'icône du programme) |
| Configuration du bureau | `.dsk` | Configuration du bureau déclarée dans la boîte de dialogue *Options d'environnement* |
| Code objet compilé | `.dcu` | |
| Programme exécutable | `.exe` | |
| DLL | `.dll` | |

*b) Le fichier de projet*

Il n'existe qu'un seul fichier *.dpr* par application. Par défaut, il contient le code suivant.

```
Program Project1;
Uses Forms,
     Unit1 in 'UNIT1.PAS'; {Form1}

{$R *.RES}
Begin
  Application.CreateForm(TForm1, Form1);
  Application.Run(Form1);
End.
```

Delphi génère automatiquement le code correspondant à de nouvelles fiches ou de nouvelles unités. Il est déconseillé d'intervenir manuellement sur le fichier de projet.

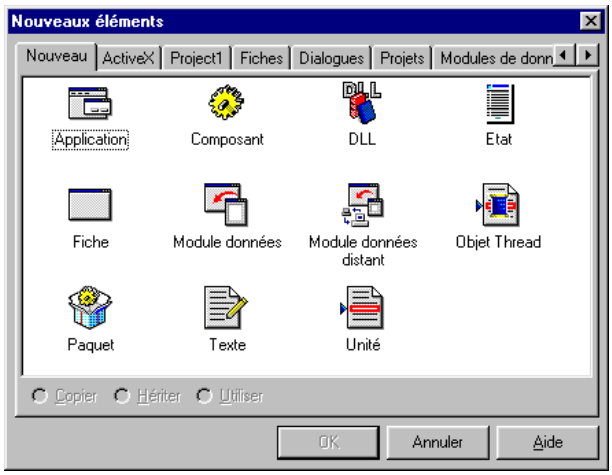
*c) Les options de projet*

Delphi permet le paramétrage d'un projet grâce à la boîte de dialogue « Options de projet » (menu Projet/Options).

- Fiches : sélection de la fiche principale de l'application (celle qui est ouverte au démarrage du programme)…
- Application : choix d'un titre pour le programme, association avec une icône…
- Compilateur : options de compilation (étonnant, non ?)
- Lieur : options d'édition des liens
- Répertoires/Conditions : chemins sur le disque, alias…
- InfoVersion : gestion des versions du projet
- Paquets : inclusion de composants externes

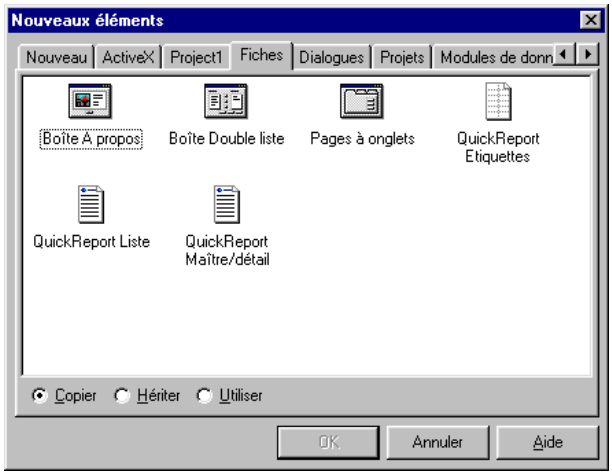
## 3. Experts et modèles

Delphi propose un ensemble de modèles et d'experts qui assistent le développeur dans la conception d'un programme en fournissant des éléments préfabriqués mais totalement remodelables. Les nombreux modèles et experts de Delphi sont accessibles par le menu Fichier/Nouveau. Quelques-uns sont détaillés ci-dessous.



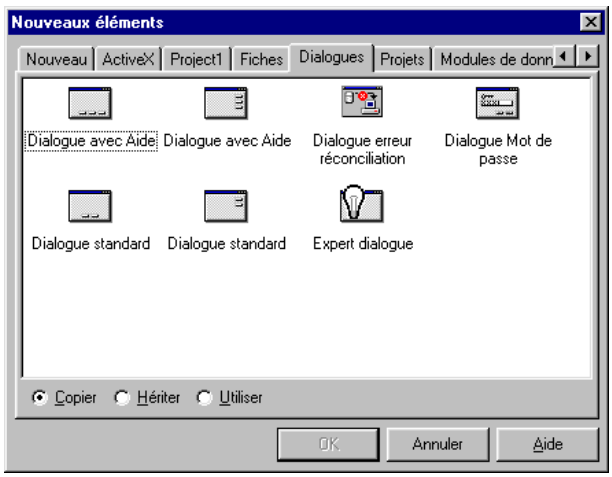

Nouveau

Modèles d'éléments standards : application, fiche, unité, DLL…



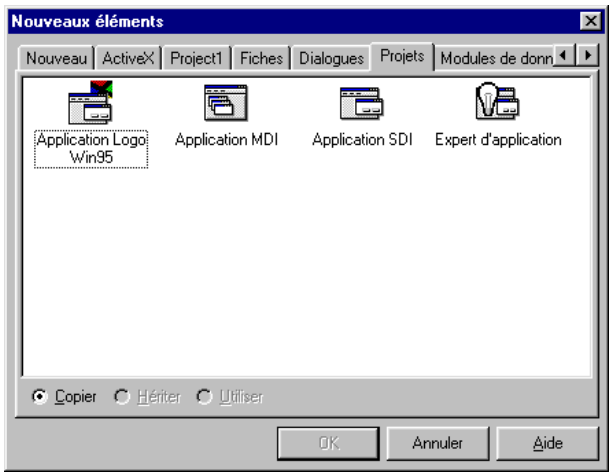

Fiches

Modèles de fiches préfabriquées




Dialogues

Modèles de boîtes de dialogue (fiches) préfabriquées




Projets

Modèles d'application :

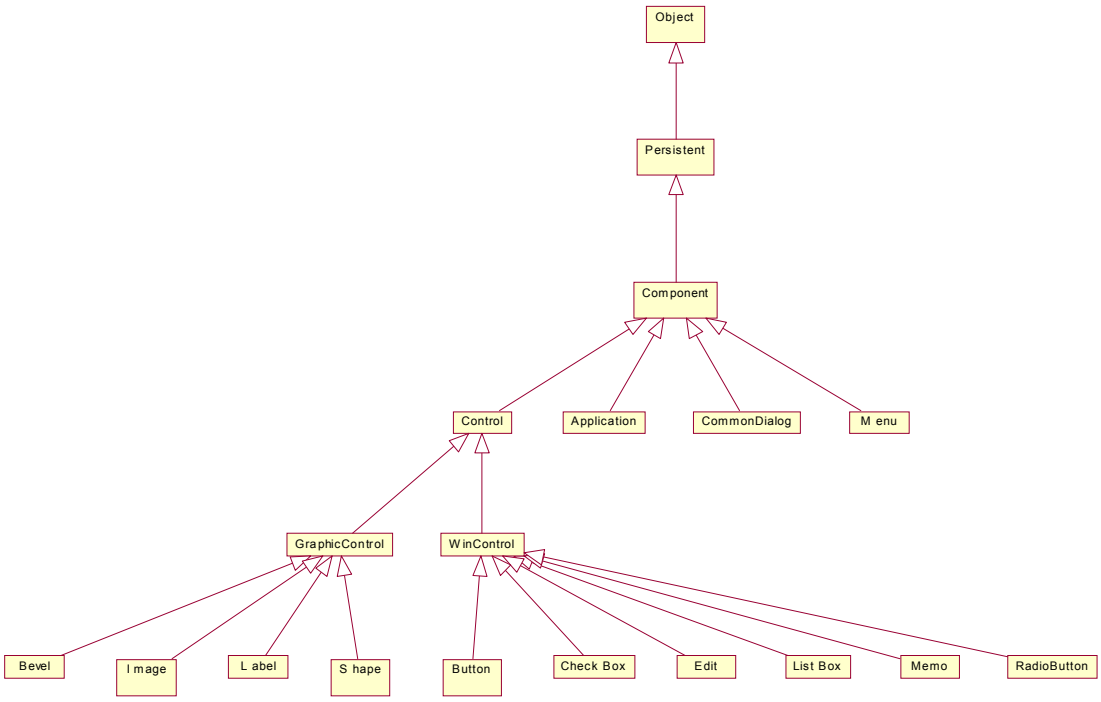
- Application Logo Win95 : modèle le plus simple avec uniquement une boîte « à propos »
- Application MDI (*Multiple Document Interface*)
- Application SDI (*Single Document Interface*), application standard

+ Expert de création d'applications

# IV. La bibliothèque d'objets de Delphi

## 1. Hiérarchie des objets Delphi

La bibliothèque d'objets de Delphi est constituée d'une hiérarchie de classes Pascal dont le sommet est *Object*. Tous les composants et contrôles sont des objets dérivés de cette classe. Une (petite) partie de cette hiérarchie est présentée ci-dessous.




### *a) Objets Delphi*

Le terme d'objet prend, dans le contexte de la bibliothèque de Delphi, un sens particulier : ce sont les objets de base du système, par opposition aux composants visuels présentés dans la partie II. Ces objets, qui constituent la partie supérieure de la hiérarchie de la bibliothèque, ne sont accessibles que par programme.

### *b) Composants et contrôles*

Composants

Ce sont des objets descendant de la classe *Component*, éléments fondamentaux à partir desquels sont construites les applications.

Ex. Application, boîtes de dialogue, barres de menu, groupe de boutons radio…

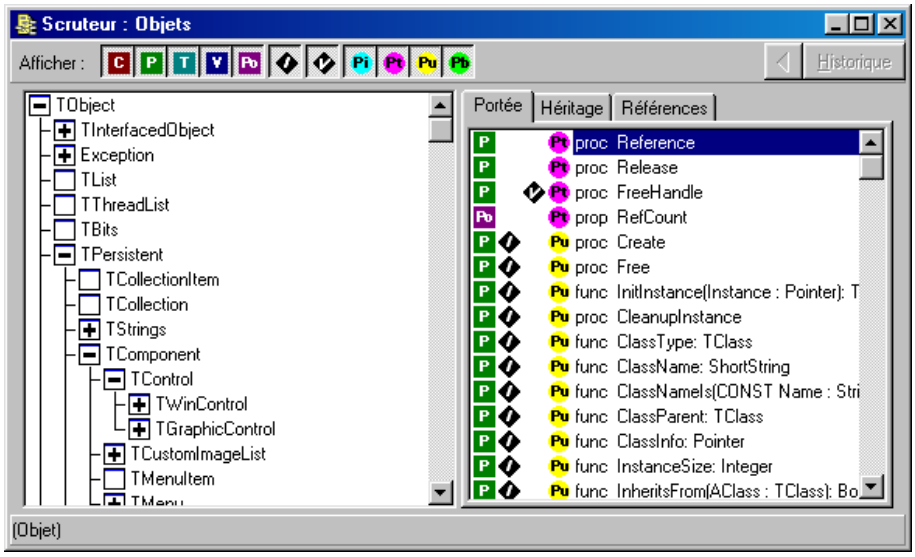
Contrôles

Les contrôles représentent l'ensemble des composants visuels manipulables grâce à la palette des composants. La cohésion des contrôles est basée sur la dérivation de la classe *Control*. Les contrôles se subdivisent en deux groupes :

- les *contrôles fenêtrés*, héritiers de la classe *WinControl*, qui sont capable de recevoir le focus de l'application, disposent d'un descripteur de fenêtre ou *handle* Windows et peuvent contenir d'autres contrôles (ex. boutons de commande, cases à cocher, boîtes de liste, boîtes d'édition…) ;
- les *contrôles graphiques*, héritiers de la classe *GraphicControl*, qui ne possèdent pas de descripteur de fenêtre, ne peuvent pas recevoir le focus de l'application et ne peuvent pas contenir d'autre contrôle (ex. cadres biseautés, images, étiquettes, formes géométriques…).

## 2. Le scruteur

Le scruteur est un outil visuel servant à examiner la hiérarchie des objets d'une application (y compris ceux de la bibliothèque). Il s'ouvre par le menu Voir/Scruteur *une fois le programme compilé*.




La partie gauche de la fenêtre (volet *Inspecteur*) présente l'arborescence des objets. La partie droite (volet *Détails*) est un classeur possédant trois onglets :

- *portée* des symboles : déclarations et éléments accessibles au niveau hiérarchique de l'objet sélectionné dans l'inspecteur (méthodes, fonctions, procédures, types…) ;
- *héritage* : situation de l'objet dans l'arbre hiérarchique ;
- *références* : localisation des occurrences des symboles dans l'application (chemin complet de l'unité + numéro de ligne).

Les informations fournies dans le volet *Détails* peuvent être filtrées à l'aide des icônes de couleur placées en haut de la fenêtre. Dans l'ordre, ces filtres sont :

- constantes,
- fonctions/procédures,
- types,
- variables,
- propriétés,
- attributs hérités,
- attributs virtuels,
- attributs privés,
- attributs protégés,
- attributs publics,
- attributs publiés.

## 3. Hiérarchies des classes et des conteneurs

Les objets Delphi sont organisés en deux hiérarchies :

- celle des classes d'objets, suivant le processus d'héritage (hiérarchie purement langagière que nous avons déjà évoquée) ;
- celle des *conteneurs*, qui concerne l'inclusion de composants les uns dans les autres. Cette hiérarchie est également double. Elle correspond à la présentation visuelle de l'interface où les contrôles peuvent graphiquement s'imbriquer les uns dans les autres et désigne la présence d'un objet comme attribut d'un autre objet.

### *a) Propriétés Parent / Control*

La propriété *Parent* appartient à tous les contrôles fenêtrés (*WinControl*) et contient la référence du contrôle parent (celui qui le contient). Symétriquement, tous les contrôles contenus dans un objet fenêtré sont ses enfants. Le cas le plus simple et le plus fréquent d'inclusion de ce type est celui d'une boîte de groupe (contrôle parent) contenant des boutons radio (contrôles enfants).

La propriété *Controls* est un tableau d'objets *Control* représentant les enfants d'un objet. La propriété *ControlCount* indique le nombre d'enfants. *Controls* est indexé de 0 à *ControlCount*–1.

### *b) Propriétés Owner / Components*

Le propriétaire d'un objet possède cet objet dans sa liste d'attributs, que cet objet soit un contrôle ou non. Des boutons radios groupés dans une boîte ont cette boîte comme parent, mais appartiennent à la fiche sur laquelle ils sont dessinés. La relation de propriété concerne la structure des objets et non leur imbrication au niveau de l'interface.

La propriété *Owner* permet de connaître le propriétaire d'un objet. Inversement, la propriété *Components* contient la liste des composants déclarés comme attributs. C'est un tableau d'objets *Component*. La propriété *ComponentCount* indique le nombre de composants. *Components* est indexé de 0 à *ComponentCount*–1.

# V. Connexion aux bases de données

Delphi propose en standard des composants pour manipuler des bases de données et des tables relationnelles. Tous ces composants ont une base commune : le *Borland Database Engine* (BDE), un noyau stable, complet et puissant.

## 1. Les composants sources

Les composants sources permettent d'accéder à des bases de données. Ils sont accessibles dans l'onglet *AccèsBD* de la palette des composants.


NB : Ces composants sont « invisibles », i.e., non visuels. Ils apparaissent sur une fiche en mode création mais pas à l'exécution du programme.

### *a) Table*

Le composant *Table* permet d'accéder aux données contenues dans une table relationnelle.

Propriétés principales

| Propriété | Description |
|---|---|
| `Active` | Ouvre ou ferme la table |
| `DataBaseName` | Nom de la base de données contenant la table |
| `Exclusive` | Empêche d'autres utilisateurs d'ouvrir la table si positionnée à true |
| `MasterFields` | Attributs de la table (détail) liés à une table maître |
| `MasterSource` | Source de données d'une table maître |
| `ReadOnly` | Autorise ou non l'utilisateur à modifier la table |
| `TableName` | Nom de la table |

### *b) Query*

Le composant *Query* (requête) permet d'effectuer une sélection sur une base de données. Il est identique au composant *Table*, mis à part la provenance des données.

Propriétés principales

| Propriété | Description |
|---|---|
| `Active` | Exécute ou non la requête |
| `DataBaseName` | Nom de la base de données interrogée |
| `DataSource` | Origine des valeurs à fournir aux paramètres de la requête |
| `Params` | Paramètres de la requête |
| `SQL` | Libellé de la requête SQL |

### *c) DataSource*

Le composant *DataSource* sert à visualiser les enregistrements d'une table ou d'une requête dans des composants visuels de Delphi. Tous ces composants BD visuels utilisent un composant *DataSource* comme source de données.

Propriété principale

| Propriété | Description |
|---|---|
| `DataSet` | Indique le composant (*Table* ou *Query*) source des données |

## 2. Formulaires basés sur des tables

### *a) Composants BD visuels*

Une fiche Delphi sur laquelle apparaissent des données issues d'une base de données est conçue à partir de composants similaires aux composants classiques, les composants BD visuels, réunis dans l'onglet *ContrôleBD* de la palette des composants.


Composants BD principaux

| Contrôle | Nom Pascal | Description |
|---|---|---|
| Texte BD | `DBText` | Texte non modifiable |
| Édition BD | `DBEdit` | Boîte d'édition d'une ligne |
| Mémo BD | `DBMemo` | Boîte d'édition multi-lignes |
| Image BD | `DBImage` | Image |
| Navigateur BD | `DBNavigateur` | Barre d'icônes pour la navigation dans la BD |

### *b) Formulaires simples*

Pour construire un formulaire simple, il suffit de suivre les étapes suivantes.

1. Placer un composant *Table* sur la fiche.
   Donner une valeur à la propriété *DatabaseName*.
   Donner une valeur à la propriété *TableName*.
   Ouvrir la table en positionnant la propriété *Active* à true.
2. Placer un composant *DataSource* sur la fiche.
   Sélectionner la table créée à l'étape 1 dans la propriété *DataSet*.
3. Ajouter sur la fiche autant de composants BD visuels que nécessaire pour afficher les données.
   Sélectionner le composant *DataSource* créé à l'étape 2 dans la propriété *DataSource* de chaque composant.
   Sélectionner un champ de la table dans la propriété *DataField* de chaque composant.

*c) Naviguer dans les données*

La façon la plus simple de naviguer dans les données est d'utiliser le composant navigateur BD (*DBNavigator*). C'est un composant graphique représentant des boutons type magnéto-cassette qui permettent de passer d'un enregistrement à l'autre, de sauter en fin de table, etc.

Pour utiliser un navigateur BD, il suffit de l'ajouter à la fiche contenant les données et de donner une valeur à sa propriété *DataSource*.

*d) Utilisation d'une grille*

Il est possible de visualiser plus d'un enregistrement à la fois à l'aide du composant universel grille BD (*DBGrid*), qui permet d'obtenir une vue des données sous forme tabulaire. Ce composant s'adapte à la structure de la table référencée afin d'en montrer tous les champs.

Pour construire un formulaire basé sur une grille, il suffit de reprendre les étapes 1 et 2 du b), puis d'ajouter un composant grille et de donner une valeur à sa propriété *DataSource*.

*e) Formulaires composés*

Les formulaires composés permettent de visualiser des associations 1, N entre deux tables. La démarche de construction d'un tel formulaire est la suivante.

1. Placer deux composants *Table* et deux composants *DataSource* sur la fiche (cf. étapes 1 et 2 du b)). L'une des tables sera la *table maître* (côté « 1 » de la relation 1, N) et l'autre la *table détail* (côté « N » de la relation 1, N).
2. Sélectionner la table détail.
   Indiquer dans sa propriété *MasterSource* le composant *DataSource* associé à la table maître.
   Éditer la propriété *MasterFields* pour effectuer le lien entre les tables (apparition d'une boîte de dialogue). Cliquer dans les parties *champ détail* et *champ maître* sur-le-

champ permettant de faire la jointure entre les deux tables, ajouter les champs joints (bouton ajouter) et valider.

3. Afficher les informations des deux tables à l'aide des composants BD visuels.

*f) L'expert fiche base de données*

L'expert fiche BD est accessible par le menu Base de données/Expert fiche… Il permet de dessiner facilement, grâce à une série de boîtes de dialogues, des formulaires simples ou composés.

## 3. Requêtes SQL

Le propos de cette section n'est pas de présenter le langage SQL, mais d'indiquer comment formuler et exécuter une requête SQL avec Delphi.

*a) Formulaire basé sur une requête*

La démarche est la même que pour créer un formulaire simple basé sur une table. Il suffit de remplacer le composant *Table* par un composant *Query* et de renseigner la propriété *SQL*.

*b) Requête paramétrée*

Soit la requête SQL suivante : `SELECT * FROM CLIENT WHERE VILLE = 'Paris'.`

Pour accéder aux habitants de Lyon, une autre requête similaire est nécessaire. Et ainsi de suite pour toutes les autres villes. Pour remédier à ce problème, Delphi permet l'insertion de paramètres dans le texte de la requête.

Version paramétrée de la requête initiale : `SELECT * FROM CLIENT WHERE VILLE = :ville.`

Il suffit d'instancier le paramètre « ville » avant l'activation de la requête (propriété *Params*).

## 4. Transfert de données

Lorsque Delphi effectue une requête, il utilise une table virtuelle (en mémoire) pour stocker le résultat. Pour conserver ce résultat sur disque, il faut créer une table réponse et y recopier les données de la table virtuelle. Pour ce faire, il faut disposer sur une fiche les composants suivants :

- un composant *Query* pour effectuer une requête ;

- un composant *Table* pour désigner la table physique (sur disque) à créer ;
- un composant *BatchMove* (onglet *AccèsBD* de la palette des composants) pour effectuer le transfert.

Attribuer à la propriété *Source* du composant *BatchMove* le composant requête, à la propriété *Destination* le composant table et à la propriété *Mode* la valeur `batCopy` (ce qui permet de créer la table si elle n'existe pas).

Il suffit ensuite de programmer un bouton de commande pour appeler la méthode *Execute* du composant *BatchMove*.

Pour transférer des enregistrements d'une table à une autre, il suffit de calquer l'opération précédente en remplaçant le composant requête par un composant table.

## 5. Manipulation de données par programme

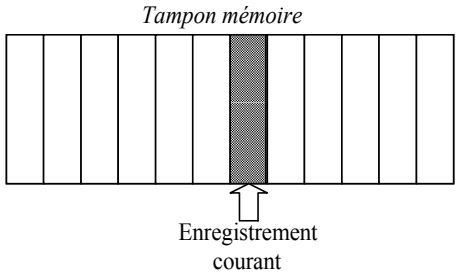
### *a) Opérations de base*

Classe *DataSet*

Les composants *Table* et *Query* héritent indirectement de la classe *DataSet*, qui permet de représenter un ensemble de données.

Enregistrement courant

Lorsqu'un *DataSet* est ouvert, un certain nombre d'enregistrements sont lus sur disque et placés dans un tampon mémoire. Cependant, un seul enregistrement est considéré comme l'enregistrement courant. La lecture d'un champ ou la suppression d'un enregistrement se font toujours par rapport à l'enregistrement courant




Lecture d'un champ

L'accès au contenu d'un champ se fait par la propriété *Fields* du *DataSet*, qui est un tableau de champs. L'accès à la valeur d'un champ peut se faire par sa position dans le tableau ou par le nom de ce champ, grâce à la méthode *FieldByName*. Puisqu'un champ peut être de nature très variée, son contenu est stocké en mémoire sous forme *physique*. Lorsque cette donnée est manipulée, elle doit l'être sous forme *logique*. Par conséquent, il faut indiquer dans quel format logique on désire accéder à la donnée.

Ex.
```
var a: integer;
Begin
    a:=Table1.Fields[0].asInteger;
    a:=Table1.FieldByName('Numero').asInteger;
    { Conversion physique-logique :
        asBoolean
        asFloat
        asInteger
        asString                    }
```

Édition d'une table

Par défaut, le *DataSet* est en mode édition, c'est-à-dire que ses données peuvent être modifiées via des contrôles. Pour empêcher cela, il faut passer la propriété *AutoEdit* du composant *DataSource* à `false`. Pour passer le *DataSet* en mode édition, il faut appeler sa méthode *Edit*.

Ex.
```
Table1.Edit;
```

Ajout/insertion d'enregistrements

- Ajout : méthode *Append* ; un enregistrement vierge est ajouté à la fin de la table.
- Insertion : méthode *Insert* ; l'enregistrement inséré se place avant l'enregistrement courant

Il faut ensuite remplir chaque champ.

Ex.
```
Table1.Insert;
Table1.FieldByName('Numero').asInteger:=100;
Table1.Append;
Table1.FieldByName('Numero').asInteger:=500;
```

Il est également possible d'ajouter ou d'insérer un enregistrement tout en remplissant mes champs grâce aux méthodes *AppendRecord* et *InsertRecord*.

Ex.
```
Table1.InsertRecord(900,'Champ 2','Champ3',3.1416);
```

Validation/annulation des mises à jour

| **Méthode du *DataSet*** | **Description** |
|---|---|
| `Post` | Valide la saisie et quitte le mode édition |
| `Refresh` | Valide la saisie, quitte le mode édition et rafraîchit les données visibles à l'écran |
| `Cancel` | Annule les modifications apportées à un enregistrement tant qu'elles n'ont pas été validées |

Ex.
```
Table1.Refresh;
```

*b) Navigation dans la base de données*

Enregistrement précédent, suivant, premier et dernier

| Méthode du *DataSet* | Description |
|---|---|
| `Prior` | L'enregistrement courant devient l'enregistrement précédent |
| `Next` | L'enregistrement courant devient l'enregistrement suivant |
| `First` | L'enregistrement courant devient le premier enregistrement |
| `Last` | L'enregistrement courant devient le dernier enregistrement |

Ex.
```
Table1.First;
Table1.Next;
Table1.Next;
```

Début et fin de table

| Méthode du *DataSet* | Description |
|---|---|
| `BOF` | Début de table (*avant* le premier enregistrement) |
| `EOF` | Fin de table (*après* le dernier enregistrement) |

Ex.
```
Table1.First;
While not Table1.EOF do
    Begin
        Writeln(Table1.FieldByName('Numero').asInteger);
        Table1.Next;
    End;
```

*c) Requêtes SQL*

Requête simple

La propriété SQL d'un composants *Query* est de type *Strings*, c'est-à-dire un tableau de chaînes de caractères. Il est donc possible de modifier cette propriété en utilisant les méthodes du type *Strings* et ainsi de créer dynamiquement une requête.

Ex.
```
Query1.Database:='DBDEMOS';
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('select * from clients');
Query1.SQL.Open;
```

Requête paramétrée

Il est également possible d'accéder à la propriété *Params* de façon dynamique.

Ex.
```
Query1.Close;
Query1.Params[0].asString:='Lyon';
Query1.Open;
```

*d) Recherche d'enregistrements*

Recherche exacte : *SetKey*/*GotoKey*, *FindKey*

*SetKey* positionne le *DataSet* en mode recherche. La propriété *Fields* permet alors de fournir les critères de recherche pour chaque champ indexé. Puis, il suffit d'appeler la méthode *GotoKey* pour activer la recherche. Si aucune correspondance n'est trouvée, *GotoKey* renvoie `false`.

Ex.
```
Procedure TForm1:RchNomPrenom(n, p: string);
Begin
    Table1.SetKey;
    Table1.FieldByName('Nom').asString:=n;
    Table1.FieldByName('Prenom').asString:=p;
    Table1.GotoKey;
End;
```

*FindKey* permet la même opération en une instruction, en précisant le ou les critères de sélection dans un tableau ouvert.

Ex.
```
Procedure TForm1:RchNomPrenom2(n, p: string);
Begin
    If Table1.FindKey([n, p]) then ShowMessage('Trouvé !');
End;
```

Recherche approchante : *FindNearest*

Ex.
```
Procedure TForm1:RchNom(n: string);
Begin
    If Table1.FindNearest([n]) then ShowMessage('Trouvé !');
End;
```

Filtres : *SetRangeStart*/*SetRangeEnd*/*ApplyRange*, *SetRange*

Les filtres permettent de limiter la vue de la table à certains enregistrements seulement. Ils sont plus particulièrement adaptés aux champs numériques, mais peuvent également être employés sur des champs alphanumériques.

Ex.
```
Table1.SetRangeStart;
Table1.FieldByName('Numero').asInteger:=50;
Table1.SetRangeEnd;
Table1.FieldByName('Numero').asInteger:=100;
Table1.ApplyRange;
```

ou
```
Table1.SetRange([100],[300]);
```

Marqueurs

Les marqueurs sont des objets de type *TBookMark*. Ils permettent de conserver un pointeur sur un enregistrement pour pouvoir y revenir rapidement par la suite.

| Méthode du *DataSet* | Description |
|---|---|
| `FreeBookMark` | Détruit un marqueur |
| `GetBookmark` | Pose un marqueur |
| `GotoBookMark` | Va au marqueur |

Ex.
```
Var marqueur: TbookMark;
Begin
    { … }
    Marqueur:=Table1.GetBookMark;
    { … }
    Table1.GotoBookMark(marqueur);
    { … }
    Table1.FreeBookMark(marqueur);
End;
```

# Références


- *Guide de l'utilisateur Delphi 3*, Borland, 1997.
- *Guide du développeur Delphi 3*, Borland, 1997.
- P. Spoljar, *Mode d'emploi Delphi*, Sybex, 1995.
- D. Lantim, *Delphi programmation avancée*, Eyrolles, 1996.
- J.-C. Armici, *Cours d'introduction à Delphi*, http://www.ellipse.ch/cours/DelphiHome.htm, 1999.
- M. Bardou, *Delphi*, http://perso.wanadoo.fr/bardou/michel/delphi.htm, 1999.