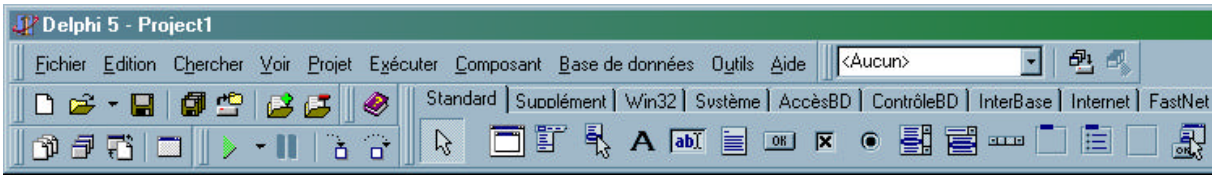
Delphi et KyliX : des descendants de Pascal

I. EDI : l'environnement de développement intégré

L'EDI est constitué de 4 fenêtres (seulement les 2 premières pour des applications consoles) .

A. La fenêtre principale



Cette partie est elle-même divisée en trois parties :

1. La barre de menus

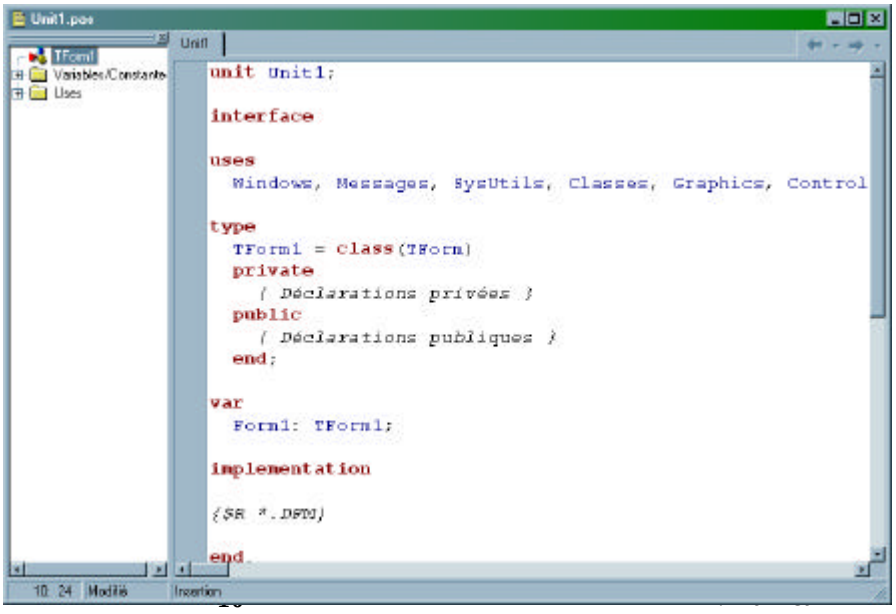
("File Edit Search..."),...
- **Les menus** : il n'y a pas grand chose à en dire. Je ne vais pas tous les passer en revue, les noms sont en général bien explicite et il y a l'aide si vraiment il y a peu de chose qui sert dans un premier temps. A part "New Application", "Run" (la flèche jaune en icône) et "Program Reset" pour arrêter l'exécution d'un programme qui ne marche pas, c'est à peu près tout...

2. La barre d'outils

Des boutons permettent d'accéder aux commandes les plus fréquemment utilisées.

3. La palette de composants

Les composants sont des bouts de programme qui ont déjà été fait pour vous. Exemple... Si vous voulez mettre un bouton dans une fenêtre, vous avez juste à prendre un bouton, et le poser sur votre fenêtre sans avoir besoin d'expliquer à l'ordinateur ce qu'est un bouton, comment on clique dessus, ce qu'il se passe graphiquement quand on clique dessus, bref, c'est déjà fait.
Les plus couramment utilisés sont ceux de la palette de composants "Standard" : dans l'ordre, on peut y voir un menu déroulant, une liste, un label (de l'écriture, quoi...), une ligne d'édition ("edit box"), une zone de texte ("menu"), un bouton, une case à cocher, etc... Tout les composants de base d'une fenêtre classique.
Il existe des tonnes de composants, et ils sont souvent compatibles entre Delphi et C++ Builder. Vous en trouverez beaucoup de gratuit sur Internet, et après, vous les ferez vous-même! Une adresse utile pour trouver de nouveaux composants : http://www.developpez.com/delphi/freewares.htm



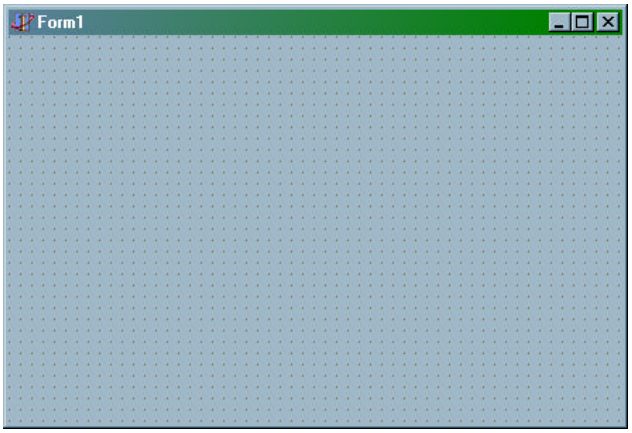
B. L'éditeur de code

Il écrit directement le minimum requis pour votre application et crée seul les fichiers associés. Par contre, le revers de la médaille, il n'est pas conseillé de modifier ce qu'il écrit. Si vous voulez supprimer un bouton par

exemple, ne l'effacez pas directement dans le code, mais effacez-le graphiquement : le compilateur s'occupera du reste. Nous verrons plus tard comment rajouter son programme...

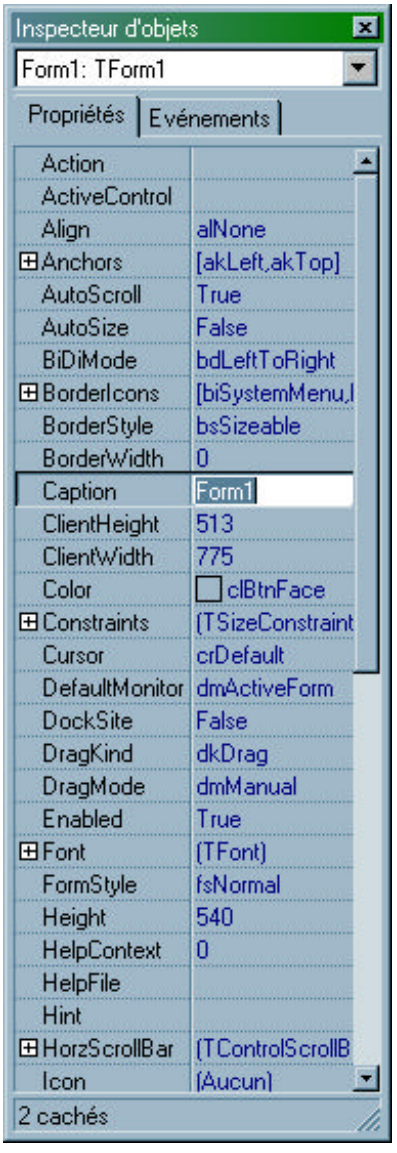
C. Le concepteur de fiches (ou de « forms »)

C'est la fenêtre que va voir l'utilisateur lorsqu'il lance le programme. Pour l'instant, elle est vide, mais c'est là qu'on peut rajouter les menus, les boutons et tout le reste... Vous pouvez, bien entendu, l'agrandir, la réduire, bref, faire tout ce que vous voulez sans taper une ligne de code...



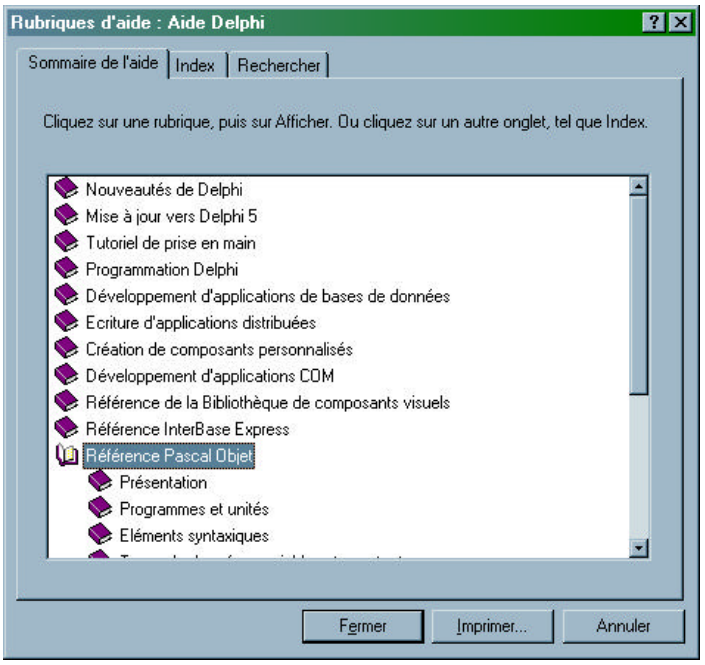
D. L'inspecteur d'objets

C'est dans cette partie qu'on donne **les caractéristiques des composants** que l'on place sur sa fenêtre ou les caractéristiques de la fenêtre générale.



Ici, on peut voir qu'il s'agit des caractéristiques générales de la fenêtre car on voit "**Form1: TForm1**". "Form1" signifie qu'on regarde les caractéristiques d'un composant de la fenêtre "Form1" (il peut en effet y avoir Form2, Form3, ou même canard ou poisson, puisque **vous donnez le nom que vous voulez** à chaque fenêtre...) et "TForm1" désigne la fenêtre en général.
Même s'il est possible de changer le nom des composants ou des fenêtres, je vous conseille de **laisser les noms par défaut** si vous n'avez pas l'intention de faire un programme énorme avec 15 fenêtres différentes.

Remarque : quand je parle de nom de la fenêtre, ne confondez pas avec ce qu'il y aura d'écrit comme titre de fenêtre. Je parle du nom de la fenêtre pour le programme ("name" dans les "properties") et non pas du titre qui s'affiche à l'écran ("caption" dans les "properties"). Ainsi, si vous placez un bouton sur votre fenêtre, vous changerez ce qu'il y a d'écrit **sur le bouton** en changeant le "Button1" de "caption" par "quitter" ou "enregistrer" ou ce que vous voulez, mais **le bouton s'appellera toujours "Button1" pour le programme**, seul l'affichage changera lors de l'exécution... Essayez de changer "caption" (la partie sélectionnée sur l'image) : le nom de la fenêtre change...



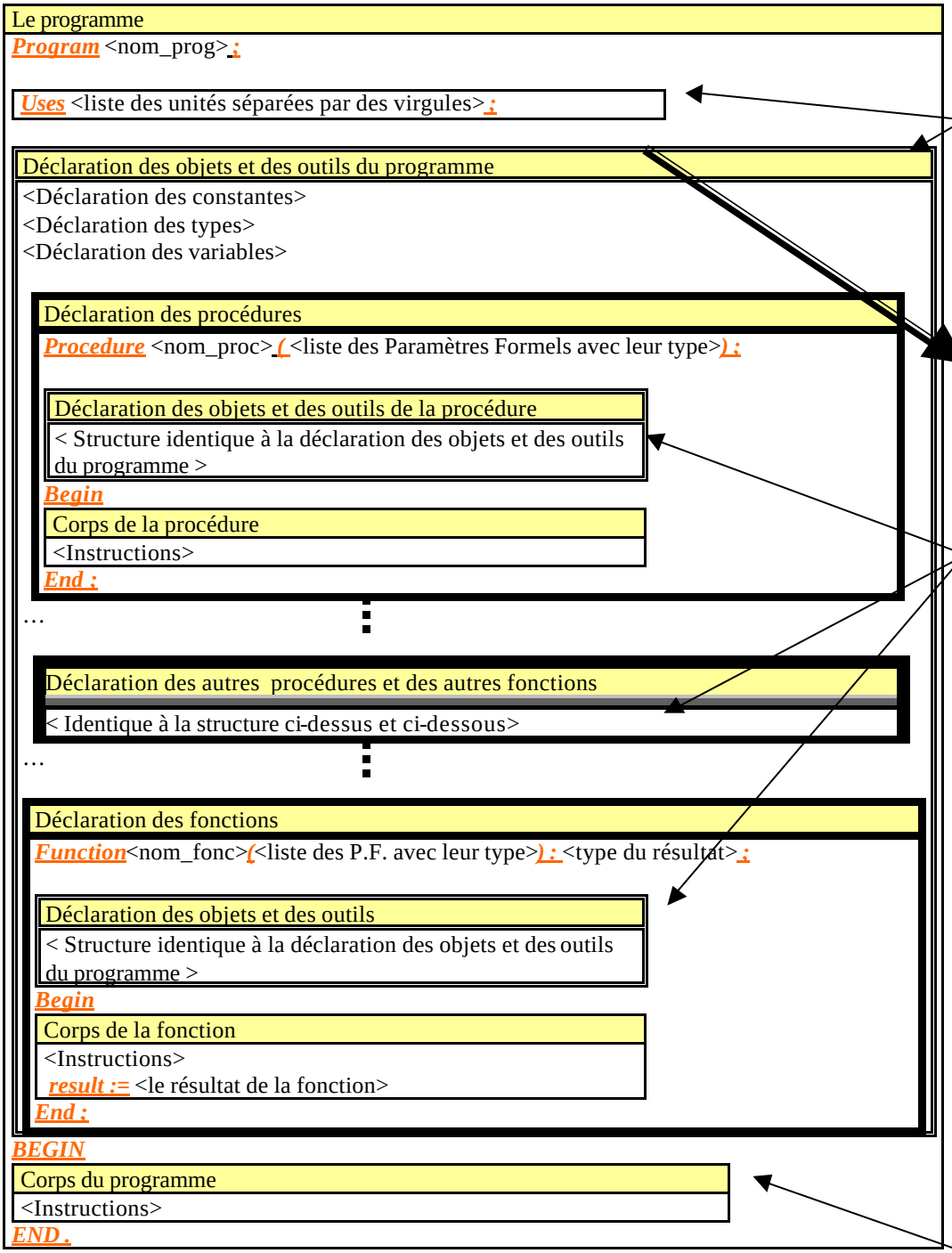
E. L'aide Delphi

Il ne faut pas hésiter à consulter l'aide Delphi pour obtenir tous les compléments nécessaires : en particulier les fonctions ne sont que décrites et on ne donne pas la liste des paramètres à employer, car ceci son décrits dans l'aide en utilisant la touche F1 ou l'index ou rechercher .

II. Présentation générale

A. Structure d'un programme Pascal, Delphi ou Kylix

Cette structure ressemble fort à celle des programmes des langages de haut niveau tels : ADA, C++,Visual basic, Java …



Le nom du programme doit être le même que le nom du fichier qui le contient.

Ces parties encadrées sont facultatives

Il n'est pas impératif de respecter l'ordre indiqué ci-contre

Remarquons que la structure des déclarations des outils et des objets (cadres doubles) est identique à la structure qui les contient : un peu à la manière des « poupées russes » qui s'ouvrent en 2 parties et qui contiennent des « poupées russes » qui s'ouvrent en 2 parties et qui contiennent …

A la différence des « poupées russes », cette structure de déclarations peut contenir récursivement non pas une mais plusieurs structures contiguës.

Une fonction renvoie un résultat par le biais de ***résult***

Cette partie aussi est facultative

En supprimant les parties facultatives, il reste le programme le plus simple que l'on puisse écrire.
Il s'agit d'une application console, la plus simple possible que vous pouvez compiler et exécuter depuis la ligne de commande

program Kedal; {$APPTYPE CONSOLE}	***déclare un programme appelé Kedal*** indique au compilateur que c'est une application console qui doit être exécutée depuis la ligne de commande déclaration des constantes, des types, des classes, des variables, des

	procédures et des fonctions (ici aucune)
begin	***début de l'exécution du programme***
	liste des instructions du programme (ici vide)
end.	***fin de l'exécution du programme***

Remarques :

- Ce qui est entre accolades { } est un commentaire (et donc ignoré du compilateur), mais
- S'il y a un $ collé contre l'accolade {$ } ceci devient une directive de compilation.
- Ne pas oublier le point-virgule (il sépare 2 instructions mais n'est pas une fin de ligne comme en ada et en C) à la fin de la première ligne et le point final.

Si Delphi est installé et si votre chemin d'accès contient le répertoire Delphi\Bin (où se trouvent les fichiers DCC32.EXE et DCC32.CFG), vous pouvez saisir ce programme dans un fichier appelé Kedal.PAS ou mieux Kedal.DPR à l'aide d'un éditeur texte (comme le bloc-notes de windows) puis le compiler en saisissant, dans une fenêtre dos, la ligne (il peut être nécessaire d'ajouter les chemins):
DCC32 Kedal
sur la ligne de commande qui produira l'exécutable résultant (kedal.EXE) dont la taille est d'environ 37 ko.
On peut l'exécuter en tapant sur la ligne de commande :
Kedal
Si l'on double-clique sur kedal.EXE dans la fenêtre de l'explorateur, une fenêtre console va s'ouvrir et se refermer immédiatement puisque ce programme ne fait rien (mais il le fait bien) dans la mesure où il ne contient aucune instruction. Ainsi, on peut « bloquer » l'exécution en ajoutant l'instruction ***readln*** qui attend que l'utilisateur ait tapé sur des touches du clavier (éventuellement aucune) puis sur la touche entrée. Il est d'ailleurs possible, avec ***readln(x),*** de mémoriser dans une variable ***x*** par exemple, ce que l'utilisateur a tapé. On aura déclaré ***x*** avant le begin. On préviendra l'utilisateur auparavant grâce à un message au moyen de ***writeln('Appuyez sur entrée')***. Tout ceci donne :

```
program Kedal;
{$APPTYPE CONSOLE}
begin
  writeln('Appuyez sur entrée');
  readln
end.
```

Mis à part sa simplicité, cet exemple diffère largement du type de programme que vous allez probablement écrire avec Delphi. Tout d'abord, c'est une application console. Delphi est généralement utilisé pour écrire des applications Windows ayant une interface graphique ; ainsi, dans une application Delphi vous ne devez normalement pas appeler Writeln. De plus, la totalité du programme exemple (à l'exception de Writeln) se trouve dans un seul fichier. Dans une application Delphi, l'en-tête du programme (la première ligne de cet exemple) se trouve dans un fichier projet séparé qui ne contient pas la logique réelle de l'application sinon quelques appels aux méthodes définies dans les fichiers unité.

B. Utiliser des unités

Il est possible, selon la nature des applications, d'utiliser ou non certaines bibliothèques qui contiennent fonctions, procédures et autres objets divers et variés. Le noyau se situe dans une unité nommée system qu'il est inutile de déclarer : on aurait très bien pu mettre :

```
uses system ;
```

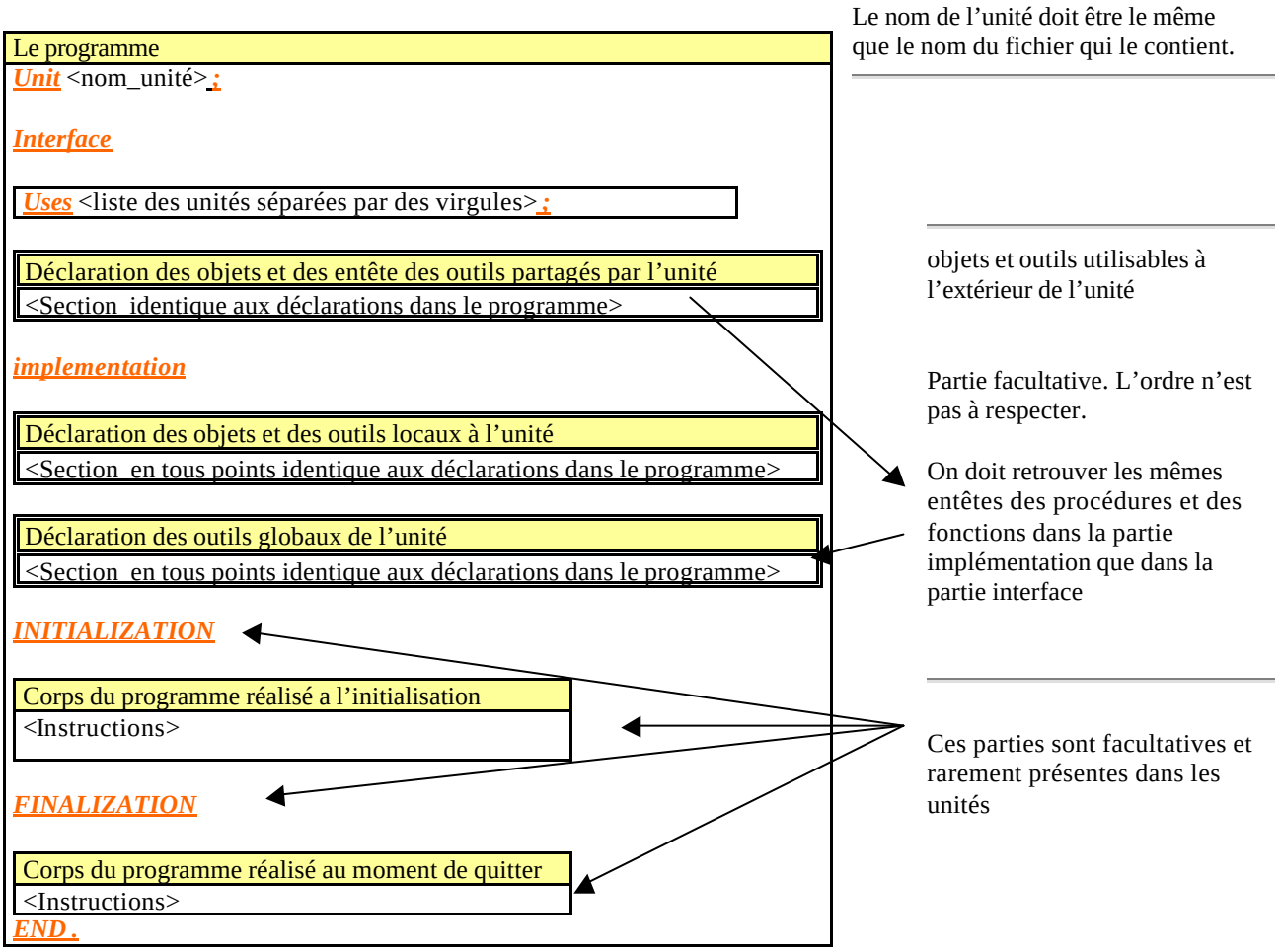
dans le programme précédent, ça n'aurait rien changé).

Writeln et readln font partie du noyau de delphi, alors que le programme :

```
program Kedal;
{$APPTYPE CONSOLE}
uses dialogs;
begin
  ShowMessage('Cliquez pour quitter');
end.
```

Le programme précédent va, en plus de l'application console, ouvrir une boîte avec un message et un bouton. Lorsque l'on clique sur le bouton, on ferme la boite à message ce qui a pour effet de terminer l'application puisque ShowMessage est la dernière instruction du programme.

C. Créer des unités



D. Les projets

L'exemple suivant propose un programme constitué de deux fichiers : un fichier projet et un fichier unité. Le fichier projet, que vous pouvez enregistrer sous le nom Kedal.DPR, a la forme suivante :

```
program Kedal;
{$APPTYPE CONSOLE}

uses Unitstop;

begin
  stop
end.
```

```
unit Unitstop;
interface

procedure stop;

implementation

procedure stop;
begin
  writeln('Appuyez sur entrée') ;
  readln
end;

end.
```

La première ligne déclare un programme appelé Kedal qui, encore une fois, est une application console. La clause uses UnitStop; spécifie au compilateur que Kedal inclut une unité appelée UnitStop. Enfin le programme appelle la procédure

Stop. Mais où se trouve la procédure Stop ? Elle est définie dans UnitStop. Le code source de UnitStop (colonne de droite) que vous devez enregistrer dans un fichier appelé UNITSTOP.PAS

UnitStop définit une procédure appelée Stop. En Pascal, les routines qui ne renvoient pas de valeur sont appelées des procédures. Les routines qui renvoient une valeur sont appelées des fonctions. Remarquez la double déclaration de Stop dans UnitStop. La première déclaration, après le mot réservé interface, rend Stop accessible aux modules (comme Kedal) qui utilisent UnitStop. Le seconde déclaration, placée après le mot réservé implémentation, définit réellement Stop.

Vous pouvez maintenant compiler Kedal depuis la ligne de commande en saisissant :
DCC32 KEDAL

Il n'est pas nécessaire d'inclure UnitStop comme argument de la ligne de commande. Quand le compilateur traite le fichier KEDAL.DPR, il recherche automatiquement les fichiers unité dont dépend le programme Kedal. L'exécutable résultant (KEDAL.EXE) fait la même chose que le premier exemple : Rien de plus que d'afficher un message et d'attendre que l'utilisateur appuie sur entrée.
Utiliser des unités

- évite au compilateur de recompiler une partie de code si le programmeur n'a pas modifié l'unité correspondante
- permet au programmeur de fournir un fichier .DCU (Unité Delphi Compilée) et un mode d'emploi rendant cette unité parfaitement utilisable tout en protégeant son code.

E. Fichiers d'un projet

En général, on regroupe tous ces fichiers (il peut y avoir plus d'une unité, et même d'autres fichiers) dans un répertoire le fichier dpr en faisant partie lui aussi. On y trouve notamment :

Extension du fichier	Description et Commentaires
DPR	(**D**elphi **PR**oject) Contient l'unité principale du projet
PAS	(**PAS**cal) Contient une unité écrite en Pascal. Peut avoir un .DFM correspondant
DFM	(**D**elphi **F**or**M** : fiche Delphi) 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) 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) 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 DSK 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, ...

III. Éléments de syntaxe

A. Commentaires

Les commentaires se placent entre accolades ou parenthèses et étoiles ou derrière 2 slaches:

{ceci est un commentaire}
*(*ceci est un autre commentaire*)*
// tout ce qui suit et jusqu'à la fin de ligne est aussi un commentaire

B. Directive de compilation

Comme les commentaires, elles se mettent entre *{ }*, mais la première accolade est suivie de $. Ce $ est lui-même immédiatement suivi de la directive. Ce ne sont pas des instructions du langage, elles servent à donner des instructions au compilateur, par exemple:

- Pour mettre au point un programme, on peut avoir besoin d'ajouter des instructions que l'on effacera dans la version définitive.

```
{$Define test}
...
{$IFDEF test}
... <Les instructions qui seront utilisées uniquement pendant la phase de mise au point>
{$ENDIF}
```

On compile le programme de cette manière pendant les tests, puis il suffit d'insérer un espace avant le $
{ $Define test}
pour transformer cette ligne en commentaire et de ce fait inhiber la définition test et par conséquent dé valider les instructions comprises entre le *{$IFDEF test}* et *{$ENDIF}*

- Pour écrire des programmes portables sous Windows et Linux :

```
uses
{$IFDEF WIN32}
  Windows, Messages, Graphics, Controls, Forms, Dialogs, Menus, StdCtrls, Buttons, ExtCtrls, MMSystem,
ComCtrls,
{$ENDIF}
{$IFDEF LINUX}
  QGraphics, QControls, QForms, QDialogs, QMenus, QStdCtrls, QButtons, QExtCtrls, QComCtrls,
{$ENDIF}
  SysUtils, Classes;
```

WIN32 ou LINUX sont automatiquement définis selon le Système d'Exploitation sous lequel **on** compile. Dans l'exemple précédent, les bibliothèques correspondant au bon S.E. seront utilisées. SysUtils et Classes étant communes aux 2 S.E.

- Pour forcer le compilateur à effectuer ou non certaines vérifications. Ce sont des directives bascules. Par exemple:
  {$B+}* Force l'évaluation complète des booléens alors que *{$B-} effectue l'évaluation partielle (Voir le **type** booléen)
  {$R+}* Vérifie le non-débordement des tableaux. Alors que *{$R-} permet de définir un tableau de 5 cellules et l'écriture d'une valeur dans la 6éme ne déclenchera pas d'erreur (Mais aura pour conséquence d'écraser peut-être d'autres variables en mémoire). Il est clair que la vérification du non-débordement une sécurité dans le déroulement du programme mais aussi génère des instructions machines supplémentaires et donc ralentissent le déroulement du programme. On peut cumules ces directives bascules et même les commenter:

```
{$B+,R-,S-}
{$R- Désactive la vérification des limites}
```

- **On** peut aussi utiliser ces directives dans la ligne de commande: (Ne pas oublier le /):

```
DCC32 MonProg  /$R-,I-,V-,U+
```

Pour les autres directives comme $I, $L etc., on se référera à l'aide de Delphi.
En utilisant L'EDI, Ces directives sont générées automatiquement et rassemblées dans un fichier DOF, DSK CFG

C. Identificateurs

Un identificateur commence obligatoirement par une lettre ou un trait-bas suivi de lettres, chiffres ou trait-bas à l'exclusion de tout autre caractère. Les caractères accentués sont aussi interdits. Comme en ADA et contrairement à C et Java, il n'y a pas de différence entre majuscule et minuscules.

D. Identificateurs qualifiés

Quand vous utilisez un identificateur qui a été déclaré à plusieurs endroits, il est parfois nécessaire de qualifier l'identificateur. On utilise un point entre les 2 identificateurs. La syntaxe d'un identificateur qualifié est :
identificateur1.identificateur2

où identificateur1 qualifie identificateur2. Si, par exemple, deux unités déclarent une variable appelée MaVar, vous pouvez désigner MaVar de Unit2 en écrivant :
Unit2.MaVar
Il est possible de chaîner les qualificateurs. Par exemple :
Form1.Button1.Click
Appelle la méthode Click de Button1 dans Form1.
Si vous ne qualifiez pas un identificateur, son interprétation est déterminée par les règles de portée décrites dans Blocs et portée.

E. Affectation

On utilise le symbole := pour donner une valeur à une variable. Par exemple:

```
a:=5;
b:='Bonjour';
```

Ou encore Pour permuter les valeurs des variables x et y :

```
t:=x; x:=y; y:=t;
```

F. Séparateur d'instruction.

Contrairement à d'autres langages (C, ADA, etc.) le Point-virgule n'est pas une fin d'instruction mais un séparateur. C'est à dire qu'il ne se place pas systématiquement à la fin d'une instruction, mais entre 2 instructions. En particulier il n'y a pas lieu de mettre un point-virgule avant un ***end*** même si le compilateur le tolère alors que le point-virgule devant un ***else*** provoque une erreur de compilation

G. Déclaration de Types, de constante et de variables

On utilise **Type Const** et **Var**. ainsi que les symboles **:** et **=** (pas le symbole **:=**)

```
Const
        lgmax=100;
        nl = #10{$IfNDef LINUX}#13{$end if} ;   //Définition des caractères de contrôles permettant un saut de ligne
Type
        Ttableau = array[1..lgmax] of integer;          // tableau de lgmax (=100) entiers
        Tchn32 = string[32];                            // chaine d'au plus 32 caractères
Var
        i : integer;
        x : real;
        c : string[12];
        c_32 : Tchn32;
        tb1 : array[1..5] of string[8];
        tb2 : Ttableau;
```

Remarquons le **=** pour les types et les constantes et le **:** pour les variables. Ainsi que les crochets pour les tableaux et les chaînes, mais on y reviendra plus tard.'

Il peut y avoir des déclarations de variables de types et de constantes en plusieurs endroits du programme Dans la mesure où l'utilisation ne peut se faire qu'après déclaration

IV. Types et structure de données simples

Les routines qui suivent sont données en vrac et correspondent à la version 5 de Delphi : Il convient de vérifier dans l'aide :

- si elles existent dans la version utilisée
- de quelle unité elles font partie (les principales font partie de l'unité system –unité par défaut- mais pas toutes !)
- quels sont les paramètres utilisés ainsi que leur type

A. Le type scalaire

1. Définition

Un type scalaire définit un ensemble ordonné de valeurs dont chaque valeur, sauf la première, a un prédécesseur unique et dont chaque valeur, sauf la dernière, a un successeur unique. Chaque valeur a un rang qui détermine l'ordre du type. Pour les types entiers, le rang d'une valeur est la valeur même ; pour tous les autres types scalaires à l'exception des intervalles, la première valeur a le rang 0, la valeur suivante a le rang 1, etc. Si une valeur a le rang n, son prédécesseur a le rang n – 1 et son successeur a le rang n + 1.

2. Routines ordinales

Dec, procédure	Décrémente une variable de 1 ou de N
Inc, procédure	Incrémente X de 1 ou de N
Odd, fonction	Renvoie True si argument est un nombre impair
Ord, fonction	Renvoie la valeur scalaire d'une expression de type scalaire
Pred, fonction	Renvoie le prédécesseur de l'argument
Succ, fonction	Renvoie le successeur de l'argument.
High, fonction	Renvoie la plus grande valeur du type
Low, fonction	Renvoie la plus petite valeur du type

3. Priorité des opérateurs

Opérateurs	Priorité
@, **not**	première (maximum)
*, /, **div**, **Mod**, **and**, **shl**, **shr**, **as**	seconde
+, –, **or**, **xor**	troisième
=, <>, <, >, <=, >=, **in**, **is**	quatrième (minimum)

Un opérateur de priorité plus élevée est évalué avant un opérateur de priorité plus basse, les opérateurs de même priorité étant évalués à partir de la gauche

4. Le type booléen

a) Définition

Type	Étendue	Format
Boolean	True et False	1 octet (8 bits)
ByteBool	True et False	1 octet
WordBool	True et False	2 octets
LongBool	True et False	4 octets

Quelques différences :

Boolean	ByteBool, WordBool, LongBool
False < True	False <> True
Ord(False) = 0	Ord(False) = 0
Ord(True) = 1	Ord(True) <> 0
Succ(False) = True	Succ(False) = True
Pred(True) = False	Pred(False) = True

b) Opérateurs

Opérateur	Opération	Types d'opérande	Types du résultat	Exemple
not	négation	booléen	Boolean	**not** (C **in** MySet)
and	conjonction	booléen	Boolean	Done **and** (Total > 0)
or	disjonction	booléen	Boolean	A **or** B
xor	disjonction exclusive	booléen	Boolean	A **xor** B

Table de vérité :

A	B	**not** A	A **or** B	A **and** B	A **xor** B
False	False	True	False	False	False

False	True	True	True	False	True
True	False	False	True	False	True
True	True	False	True	True	False

Retour sur la directive de compilation $B+ :

True or X donne **True** quelque soit la valeur de X, de même **False and** X donne **False** quelque soit la valeur de X
Dans une expression booléenne faisant intervenir un **or**, lorsque le premier opérande vaut True, il est inutile de calculer la valeur du second. Ceci pose un problème lorsque celui-ci est un prédicat (fonction à résultat booléen) et que cette fonction produit un effet de bord (Modification de l'environnement global) La seconde évaluation n'étant pas faite, l'effet de bord n'a pas lieu. Il en va de même lorsque le premier opérande d'un **and** vaut faux

c) Procédure et fonctions

Ce sont les routines ordinales valables pour tous les types scalaires

d) Opérateurs relationnels

Les opérateurs relationnels sont utilisés pour comparer deux opérandes. Les opérateurs =, <>, <= et >= s'appliquent également aux ensembles (voir Opérateurs d'ensembles) ; = et <> s'appliquent également aux pointeurs (voir Opérateurs de pointeurs).

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
=	égalité	simple, classe, référence de classe, interface, chaîne, chaîne compactée	Boolean	I = Max
<>	différence	simple, classe, référence de classe, interface, chaîne, chaîne compactée	Boolean	X <> Y
<	inférieur à	simple, chaîne, chaîne compactée, PChar	Boolean	X < Y
>	supérieur à	simple, chaîne, chaîne compactée, PChar	Boolean	Len > 0
<=	inférieur ou égal à	simple, chaîne, chaîne compactée, PChar	Boolean	Cnt <= I
>=	Supérieur ou égal à	simple, chaîne, chaîne compactée, PChar	Boolean	I >= 1
in	Membre de	opérande gauche: tout **type** ordinal T, opérande droit: ensemble de base compatible avec T.	Boolean	5 **in** [3..9]; 'K' **in**['a'..'z'];

Dans la plupart des cas simples, la comparaison est évidente. Par exemple I = J vaut True uniquement si I et J ont la même valeur, sinon I>J vaut True. Les règles suivantes s'appliquent aux opérateurs de comparaison :

Les opérandes doivent être de types compatibles, sauf pour un réel et un entier qui peuvent être comparés.

e) Exemples

Selon la règle des priorités des opérateurs :
A>B **or** C<=D provoquera une erreur car B **or** C sera évalué avant toute chose : Il faut donc systématiquement mettre de parenthèses pour modifier l'ordre d'évaluation:

```
(A>B) or (C<=D)
```

5. Le type entier

a) Définition

Type	Étendue	Format
Integer	–2147483648..2147483647	32 bits signé
Cardinal	0..4294967295	32 bits non signé
Shortint	–128..127	8 bits signé
Smallint	–32768..32767	16 bits signé
Longint	–2147483648..2147483647	32 bits signé
Int64	–2^63..2^63–1	64 bits signé
Byte	0..255	8 bits non signé
Word	0..65535	16 bits non signé
Longword	0..4294967295	32 bits non signé

Par défaut les entiers sont exprimés en décimal , mais il est possible de manipuler des nombres hexadécimaux :
$FF vaut 255

b) Opérateurs

En plus des opérateurs relationnels définis plus haut, les opérateurs arithmétiques suivants attendent des opérandes réels ou entiers : +, –, *, /, **div** et **Mod**.

Opérateur	Opération	Types d'opérande	Type du résultat	Exemple
+	addition	entier, réel	entier, réel	X + Y
–	soustraction	entier, réel	entier, réel	Result - 1
*	multiplication	entier, réel	entier, réel	P * InterestRate
/	division réelle	entier, réel	réel	X / 2
div	division entière	entier	entier	Total **div** UnitSize
Mod	reste	entier	entier	Y **Mod** 6
+ (unaire)	signe identité	entier, réel	entier, réel	+7
– (unaire)	signe négation	entier, réel	entier, réel	-X
not	négation bit à bit	entier	entier	
and	et bit à bit	entier	entier	
or	ou bit à bit	entier	entier	
xor	ou exclusif bit à bit	entier	entier	
shl	rotation des bits vers la gauche	entier	entier	
shr	rotation des bits vers la droite	entier	entier	

Les règles suivantes s'appliquent aux opérateurs arithmétiques.

La valeur de x/y est de type Extended (voir type réel), indépendamment du type de x et y. Pour les autres opérateurs, le résultat est de type Extended dès qu'au moins un des opérandes est de type réel ; sinon le résultat est de type Int64 quand au moins un des opérandes est de type Int64 ; sinon le résultat est de type Integer. Si le type d'un opérande est un sous-intervalle d'un type entier, il est traité comme étant de ce type entier.

La valeur de x **div** y est la valeur de x/y arrondi vers le bas à l'entier le plus proche.

L'opérateur **Mod** renvoie le reste obtenu par la division de ses opérandes. En d'autres termes : x **Mod** y = x – (x **div** y) * y.

Il y a une erreur d'exécution si y vaut zéro dans une expression de la forme x/y, x **div** y ou x **Mod** y.

A	B	**not** A	A **or** B	A **and** B	A **xor** B
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

shl produit un décalage des bits de l'entier vers la gauche et ajoute un 0 donc à multiplier par 10 en binaire (c'est à dire2 en décimal).

Shr produit un décalage dans l'autre sens c'est à dire diviser par 2.

n **shl** p (décalage de p bits a gauche avec introduction de p 0) donne $n*2^p$.

c) Procédure et fonctions

Fonction	
abs	Valeur absolue
Hi	poids fort
Lo	poids faible
odd	test de parité
Sqr	carré du nombre

On remarquera que la fonction puissance n'est pas définie en standard : il faut utiliser l'unité ***Math*** ou la fonction ***IntPower*** est définie

d) Exemple

compte tenu de la priorité des opérateurs:

5+ 3*2 donne 11
56 **or** 20 donne 60
56 **and** 20 donne 16
56 **xor** 20 donne 44
en binaire 56 s'écrit 00111000 et 20 s'écrit 00010100
56 **shl** 2 donne 224 décaler 00111000 de 2 rangs à gauche donne 11100000

6. Le type caractère

a) Définition

Les types de caractère fondamentaux sont AnsiChar et WideChar. Les valeurs AnsiChar sont des caractères sur un octet (8 bits) ordonnés selon le jeu de caractères ANSI étendu. Les valeurs WideChar sont des caractères sur un mot (16 bits) ordonnés selon le jeu de caractères Unicode. Les 256 premiers caractères Unicode correspondent aux caractères ANSI.
Le type de caractère générique Char est équivalent à AnsiChar. Comme l'implémentation de Char est susceptible de changer, il est judicieux d'utiliser la fonction standard SizeOf plutôt qu'une constante codée en dur dans les programmes qui doivent gérer des caractères de tailles différentes.

Une constante chaîne de longueur 1, comme 'A', peut désigner une valeur caractère. La fonction prédéfinie Chr renvoie la valeur caractère pour tout entier dans l'étendue de AnsiChar ou de WideChar ; ainsi, Chr(65) renvoie la lettre (code ASCII 65).
un entier préfixé par # donne le caractère dont le code ASCII est cet entier.

b) Opérateurs

Les opérateurs relationnels

c) Procédure et fonctions

Les routines ordinales et

Fonction	
chr	donne le caractère correspondant au code ASCII
upcase	convertit un caractère en majuscule si elle exis te et le conserve sinon. Attention aux caractères accentués

d) Exemples

Les valeurs de caractère, comme les entiers, bouclent quand elles sont décrémentées ou incrémentées au-delà du début ou de la fin de leur étendue (à moins que la vérification des limites ne soit activée). Ainsi, une fois le code suivant exécuté

```
var lettre : char;
...
begin
  lettre :=High(char); // le caractère dont le code ASCII est 255 . Le code du suivant est 0
..lettre:=High(lettre);  // est équivalent à la ligne précédente
  inc(lettre,66);  // caractère dont le code Ascii est 65 c'est à dire A
...
  lettre := #65;  // est équivalent à  lettre := chr(65)
pred('D') vaut C          'C'<>'c' vaut True      Chr(66) vaut B          ord('B') vaut 66
```

7. Le type énuméré

a) Définition

Un type énuméré définit une collection ordonnée de valeurs simplement en énumérant les identificateurs désignant ces valeurs. Les valeurs n'ont pas de signification propre et leur rang suit l'ordre d'énumération des identificateurs.
Pour déclarer un type énuméré, utilisez la syntaxe suivante :
type nomType = (val1, ..., valn)
où nomType et les val sont des identificateurs valides.

b) Exemples

```
…
Type
        Tcartes = (_7,_8,_9,_10,Valet,Dame,Roi,As); // Remarquons le trait_bas pour le respect des
règles sur les identifiants
var
        carte : Tcartes;
        i : byte;
BEGIN
  carte := _8;
  inc(carte,2);
  carte := pred(carte);
  i:= ord(carte);
  writeln(i,' ',_9<Dame,' ',ord(_7));  // affichera
2 TRUE 0
  readln;  // la tentative d'afficher carte
provoquerait une erreur
END.
```

8. Le type intervalle

a) Définition

Un type intervalle représente un sous-ensemble de valeurs d'un autre type (appelé le type de base). Toute construction de la forme Bas..Haut, où Bas et Haut sont des expressions constantes du même type scalaire, Bas étant inférieur à Haut, identifie un type intervalle qui inclut toutes les valeurs comprises entre Bas et Haut.

b) Exemples

```
Type
  Tchiffres =0..9;
  Tminuscules = 'a'..'z';
  Tcartes = (_7,_8,_9,_10,Valet,Dame,Roi,As);  // C'est un type énuméré pour définir le type suivant
  Tfigures = Valet..Roi;   //  qui lui est du type intervalle
```

B. Le type réel

1. Définition

Un type réel définit un ensemble de nombres pouvant être représentés par une notation à virgule flottante. Le tableau suivant donne l'étendue et le format de stockage des types réels fondamentaux.

Type	Étendue	Chiffres significatifs	Taille en octets
Real48	2.9 x 10^–39 .. 1.7 x 10^38	11–12	6
Single	1.5 x 10^–45 .. 3.4 x 10^38	7–8	4
Double	5.0 x 10^–324 .. 1.7 x 10^308	15–16	8
Real	5.0 x 10^–324 .. 1.7 x 10^308	15–16	8
Extended	3.6 x 10^–4951 .. 1.1 x 10^4932	19–20	10
Comp	–2^63+1 .. 2^63 –1	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8

Le type générique Real est équivalent, dans son implémentation actuelle, au type Double.
La notation classique avec le point décimal est employée. La notation scientifique (E ou e, puis un exposant) se lit "puissance 10" Currency est un type de données à virgule fixe qui limite les erreurs d'arrondi dans les calculs monétaires.

Il est possible de stocker un entier dans un réel mais pas l'inverse.

2. Opérateurs

Voir les opérateurs sur les entiers.

3. Routines arithmétiques

Vérifier dans l'aide s'il est nécessaire d'utiliser l'unité ***Math***

*Abs, fonction	Renvoie une valeur absolue	Log2, fonction	Calcule le logarithme en base 2
Ceil, fonction	Arrondit des variables vers l'infini positif	LogN, fonction	Calcule le logarithme en base N
Exp, fonction	Renvoie la valeur exponentielle de X	Max, fonction	Renvoie la plus élevée des valeurs parmi deux valeurs numériques
Floor, fonction	Arrondit les variables vers l'infini négatif	Min, fonction	Renvoie la plus petite de deux valeurs numériques
Frac, fonction	Renvoie la partie décimale d'un réel	Pi, fonction	Renvoie 3.14159265358979323 85
Frexp, procédure	Sépare la mantisse et l'exposant de X	Poly, fonction	Évalue une polynomiale uniforme

Int, fonction	Renvoie la partie entière d'un nombre réel		d'une variable à la valeur X
IntPower, fonction	Calcule la puissance entière d'une valeur de base	Power, fonction	Élève Base à n'importe quelle puissance
Ldexp, fonction	Calcule X * (2**P)	Round, fonction	Renvoie la valeur de X arrondi au plus proche entier
Ln, fonction	Renvoie le logarithme naturel d'une expression réelle	Sqr, fonction	Renvoie le carré d'un nombre
LnXP1, fonction	Renvoie le logarithme naturel de (X+1)	Sqrt, fonction	Renvoie la racine carrée de X
Log10, fonction	Calcule le logarithme en base 10	Trunc, fonction	Tronque un réel en entier.

4. Routines de nombres aléatoires

RandG, fonction	Génère des nombres aléatoires avec une distribution gaussienne
RandSeed, variable	RandSeed stocke la matrice du générateur de nombres aléatoires
Random, fonction	Génère des nombres aléatoires dans une étendue spécifiée
Randomize, procédure	Initialise le générateur interne de nombre aléatoire avec une valeur aléatoire.

5. Exemples

7E-2 signifie 7×10^{-2} et 12.25e+6 et 12.25e6 signifient 12.25×10^{6}.

C. Les types chaînes

1. Définition

Une chaîne représente une suite de caractères. Le **Pascal** Objet gère les types de chaîne prédéfinis suivants.

Type	Longueur maximum	Mémoire nécessaire	Utilisation
ShortString	255 caractères	de 2 à 256 octets	Compatibilité ascendante
AnsiString	~2^31 caractères	de 4 octets à 2Go	Caractère sur 8 bits (ANSI)
WideString	~2^30 caractères	de 4 octets à 2Go	Caractères Unicode;

Le mot réservé **string** fonctionne comme un identificateur de type générique. Dans l'état par défaut *{$H+}*, le compilateur interprète **string** (quand il apparaît sans être suivi d'un crochet ouvrant) comme désignant AnsiString. Utilisez la directive *{$H–}* pour que **string** soit interprété comme désignant ShortString.

2. Routines de gestion des chaînes

AdjustLineBreaks, fonction	Standardise les caractères de fin de ligne en paires CR/LF
AnsiCompareStr, fonction	Compare deux chaînes basées sur le pilote de langue de Windows (les différences de majuscules/minuscules sont détectées)
AnsiCompareText, fonction	Compare deux chaînes basées sur le pilote de langue de Windows (les différences de majuscules/minuscules ne sont pas détectées)
AnsiExtractQuotedStr, fonction	Convertit une chaîne guillemetée en une chaîne non guillemetée
AnsiLowerCase, fonction	Renvoie une chaîne, qui est une copie de la chaîne donnée convertie en minuscules
AnsiPos, fonction	Trouve la position d'une sous-chaîne dans une chaîne
AnsiQuotedStr, fonction	Renvoie la version guillemetée d'une chaîne
AnsiSameStr, fonction	Compare deux chaînes basées sur le pilote de langue de Windows (les différences de majuscules/minuscules sont détectées)
AnsiSameText, fonction	Compare deux chaînes basées sur le pilote de langue de Windows (les différences de majuscules/minuscules ne sont pas détectées)
AnsiUpperCase, fonction	Convertit une chaîne en majuscules
CompareStr, fonction	Compare des chaînes en tenant compte de la distinction minuscules/majuscules
CompareText, fonction	Compare des chaînes par valeur scalaire sans tenir compte de la distinction minuscules/majuscules
Concat, fonction	Concatène deux chaînes ou plus
Copy, fonction	Renvoie une sous-chaîne d'une chaîne ou un segment de tableau dynamique
Delete, procédure	Supprime une sous-chaîne d'une chaîne s
Insert, procédure	Insère une sous-chaîne dans une chaîne commençant au point spécifié
IsDelimiter, fonction	Indique si un caractère spécifié dans une chaîne correspond à un ensemble de délimiteurs
LastDelimiter, fonction	Renvoie l'index d'octet dans S du dernier caractère identique au caractère spécifié par la chaîne AnsiString Delimiters
Length, fonction	Renvoie le nombre de caractères dans une chaîne ou d'éléments dans un tableau

LowerCase, fonction	Convertit une chaîne ASCII en minuscules
NullStr, constante	Déclare un pointeur sur EmptyStr
Pos, fonction	Renvoie la valeur d'index du premier caractère dans une sous-chaîne spécifiée qui se trouve dans une chaîne
QuotedStr, fonction	Renvoie la version guillemetée d'une chaîne
SetLength, procédure	Définit la taille d'une variable chaîne ou tableau dynamique
SetString, procédure	Définit le contenu et la taille d'une chaîne
Str, procédure	Formate une chaîne et la renvoie dans une variable
StringOfChar, fonction	Renvoie une chaîne avec le nombre de caractères spécifié
StringReplace, fonction	Renvoie une chaîne AnsiString dans laquelle des occurrences d'une sous-chaîne sont remplacées par une autre sous-chaîne
Trim, fonction	Supprime les caractères de contrôle et les espaces se trouvant en début et en fin de chaîne
TrimLeft, fonction	Supprime les caractères de contrôle et les espaces se trouvant en début de chaîne
TrimRight, fonction	Supprime les caractères de contrôle et les espaces se trouvant en fin de chaîne
UpperCase, fonction	Revoie une copie d'une chaîne en majuscules
Val, procédure	Convertit une chaîne en sa représentation numérique
WrapText, fonction	Décompose une chaîne en plusieurs lignes quand sa longueur se rapproche d'une taille donnée.

3. Opérateurs de chaîne

Opérateur	Opération	Types d'opérande	**Type du résultat**	Exemple
+	concaténation	chaîne, chaîne compactée, caractère	**chaîne**	S + '. '

Les règles suivantes s'appliquent à la concaténation de chaîne :
Les opérateurs relationnels =, <>, <, >, <= et >= acceptent tous des opérandes chaîne (voir Opérateurs relationnels). L'opérateur + concatène deux chaînes.
Les opérandes pour l'opérateur + peuvent être des chaînes, des chaînes compactées (des tableaux compactés de type Char) ou des caractères. Cependant, si un opérande est de type WideChar, l'autre opérande doit être une chaîne longue.

Le résultat d'une opération + est compatible avec tout type de chaîne. Cependant, si les opérandes sont tous deux des chaînes courtes ou des caractères, et si leur longueur cumulée est supérieure à 255, le résultat est tronqué aux 255 premiers caractères.

4. Les chaînes courtes

a) Définition

Une variable de type chaîne ShortString est une séquence de caractères de longueur variable au cours de l'exécution et une taille prédéfinie entre 1et 255. On peut préciser la longueur maxi entre crochets.

b) Exemples

```
Type Tch32 = string[32]; // ou Tch32 = ShortString[32]
Var ch : Tch32 ; // Réserve 33 octets : ch[i] fait référence au ième caractère de la chaîne. ch[0] fait référence à sa longueur.
...
ch:=#98#111#110#106'o'#117#114;  // représente la chaîne 'bonjour '
ch := 'c''est une chaîne' // Lorsqu'une chaîne comporte une apostrophe, on doit la doubler.
```

5. Les chaînes longues et étendues

a) Définition

Les types AnsiString (caractères Ansi 8 bits) et WideString (caractères Unicodes 16 bits) ou **string** sans crochets sont des chaînes allouées dynamiquement. (Ce sont en réalité des pointeurs.) La gestion de la mémoire est entièrement automatique. A priori les différents types de chaînes ne semblent pas présenter de grosses différences : c'est illusoire, nous y reviendrons lors de l'étude du type fichier.

b) Exemples

```
Var Ch : string; // ou Ch : AnsiString;
...
```

```
Ch:=#98#111#110#106'o'#117#114;  // représente la chaîne 'bonjour '
Ch := 'c''est une chaîne' // Lorsqu'une chaîne comporte une apostrophe, on doit la doubler.
```

6. Les chaînes AZT

a) Définition

Pour des raisons de compatibilité avec d'autres langages (tel le C et C++), une chaîne à zéro terminal est un tableau de caractères d'indice de base zéro et terminé par NULL (#0). Comme le tableau n'a pas d'indicateur de longueur, le premier caractère NULL indique la fin de la chaîne.
Le type ***Pchar*** est en réalité un pointeur Pour plus de précision voir l'aide Delphi et le type pointeur ci-après.

b) Opérateurs

c) Procédure et fonctions

StrAlloc	Alloue sur le tas un tampon de caractères d'une taille donnée.
StrBufSize	Renvoie la taille du tampon de caractères alloué sur le tas en utilisant StrAlloc ou StrNew.
StrCat	Concatène deux chaînes.
StrComp	Compare deux chaînes.
StrCopy	Copie une chaîne.
StrDispose	Libère un tampon de caractères alloué en utilisant StrAlloc ou StrNew.
StrECopy	Copie une chaîne et renvoie un pointeur sur la fin de la chaîne.
StrEnd	Renvoie un pointeur sur la fin de la chaîne.
StrFmt	Formate une ou plusieurs valeurs dans une chaîne.
StrIComp	Compare deux chaînes sans tenir compte des différences majuscules/minuscules.
StrLCat	Concatène deux chaînes avec une longueur maximum donnée pour la chaîne résultante.
StrLComp	Compare deux chaînes sur une longueur maximum donnée.
StrLCopy	Copie une chaîne jusqu'à une longueur maximum donnée.
StrLen	Renvoie la longueur d'une chaîne.
StrLFmt	Formate une ou plusieurs valeurs dans une chaîne avec une longueur maximum donnée.
StrLIComp	Compare deux chaînes sur une longueur maximum donnée sans tenir compte des différences majuscules/minuscules.
StrLower	Convertit une chaîne en minuscules.
StrMove	Déplace un bloc de caractères d'une chaîne dans une autre.
StrNew	Alloue une chaîne sur le tas.
StrPCopy	Copie une chaîne Pascal dans une chaîne à zéro terminal.
StrPLCopy	Copie une chaîne Pascal dans une chaîne à zéro terminal avec une longueur maximum donnée.
StrPos	Renvoie un pointeur sur la première occurrence d'une sous-chaîne donnée dans une chaîne.
StrRScan	Renvoie un pointeur sur la dernière occurrence d'un caractère donné dans une chaîne.
StrScan	Renvoie un pointeur sur la première occurrence d'un caractère donné dans une chaîne.
StrUpper	Convertit une chaîne en majuscules.

d) Exemples

```
Var
Ch : Pchar;
...
Ch := 'bonjour'; // Là encore la gestion de mémoire est transparente pour le programmeur.
```

D. Chaînes de format

Les chaînes de format transmises aux routines de définition de format de chaînes contiennent deux types d'objets : les caractères simples et les spécificateurs de format. Les caractères simples sont copiés tels quels dans la chaîne résultante. Les spécificateurs de format récupèrent les arguments dans la liste des arguments en y appliquant un format.

Les spécificateurs de format ont la forme suivante :

"%" [index ":"] ["-"] [width] ["." prec] type

Un spécificateur de format commence par un caractère %. Ce qui suit % est, dans l'ordre :

Le spécificateur facultatif d'indice de l'argument, [index ":"]

L'indicateur facultatif d'alignement à gauche, ["-"]

Le spécificateur facultatif de taille, [width]

Le spécificateur facultatif de précision, ["." prec]

Le caractère de type de conversion, type

Le tableau suivant résume les valeurs possibles de type :

d Décimal. L'argument doit être une valeur entière. La valeur est convertie en chaîne de chiffres décimaux. Si la chaîne de format contient un spécificateur de précision, la chaîne résultante doit contenir au moins le nombre indiqué de chiffres ; si cela n'est pas le cas, des caractères zéro de remplissage sont rajoutés dans la partie gauche de la chaîne.

u Décimal sans signe. Comme 'd' mais sans signe en sortie.

e Scientifique. L'argument doit être une valeur à virgule flottante. La valeur est convertie en une chaîne de la forme "-d.ddd...E+ddd". La chaîne résultante débute par un signe moins si le nombre est négatif. Un chiffre précède toujours le séparateur décimal. Le nombre total de chiffres dans la chaîne résultante (y compris celui qui précède la virgule) est donné par le spécificateur de précision dans la chaîne de format. Si celui-ci est omis, une précision de 15 est prise en compte par défaut. Le caractère "E" dans la chaîne résultante est toujours suivi d'un signe plus ou moins, puis de trois chiffres au moins.

f Fixe. L'argument doit être une valeur à virgule flottante. La valeur est convertie en une chaîne de la forme "-ddd.ddd...". La chaîne résultante débute par un signe moins si le nombre est négatif.Le nombre de chiffres après la virgule est fourni par le spécificateur de précision de la chaîne de format ; 2 décimales sont prises en compte par défaut si le spécificateur de précision est omis.

g Général L'argument doit être une valeur à virgule flottante. La valeur est convertie en une chaîne la plus courte possible en utilisant le format fixe ou scientifique. Le nombre de chiffres significatifs dans la chaîne résultante est fourni par le spécificateur de précision dans la chaîne de format : une précision par défaut de 15 est prise en compte si le spécificateur de précision est omis. Les caractères zéro sont supprimés de la fin de la chaîne résultante et le séparateur décimal n'apparaît que s'il est nécessaire. La chaîne résultante utilise le format fixe si le nombre de chiffres à gauche de la virgule est inférieur ou égal à la précision indiquée et si la valeur est supérieure ou égale à 0,00001. Sinon, la chaîne résultante fait appel au format scientifique.

n Numérique L'argument doit être une valeur à virgule flottante. La valeur est convertie en une chaîne de la forme "-d,ddd,ddd.ddd...". Le format "n" correspond au format "f", sauf que la chaîne résultante contient le séparateur des milliers.

m Monétaire. L'argument doit être une valeur à virgule flottante. La valeur est convertie en une chaîne représentant un montant monétaire. La conversion est contrôlée par les variables globales CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator et CurrencyDecimals. Si la chaîne de format contient un spécificateur de précision, il remplace la valeur envoyée par la variable globale CurrencyDecimals.

p Pointeur. L'argument doit être une valeur de type pointeur. La valeur est convertie en une chaîne de 8 caractères qui représente des valeurs de pointeur en hexadécimal.

s Chaîne. L'argument doit être un caractère, une chaîne ou une valeur PChar. La chaîne ou le caractère est inséré à la place du spécificateur de format. Le spécificateur de précision, s'il est défini dans la chaîne de format, indique la taille maximale de la chaîne résultante. Si l'argument est une chaîne de taille supérieure, celle-ci est tronquée.

x Hexadécimal. L'argument doit être une valeur entière. La valeur est convertie en une chaîne de chiffres hexadécimaux. Si la chaîne de format contient un spécificateur de précision, ce dernier spécifie que la chaîne doit contenir au moins le nombre indiqué de chiffres ; si cela n'est pas le cas, des caractères zéro de remplissage sont rajoutés dans la partie gauche de la chaîne.

Les caractères de conversion peuvent être indiqués indifféremment en majuscules ou en minuscules : le résultat obtenu est le même.

Quel que soit le format flottant, les deux caractères utilisés comme séparateur décimal et séparateur des milliers sont respectivement définis par les variables globales DecimalSeparator et ThousandSeparator.

Les spécificateurs d'indice, de taille et de précision peuvent être directement spécifiés en utilisant des chaînes contenant des chiffres décimaux (par exemple "%10d") ou indirectement, en utilisant le caractère astérisque (par exemple "%*.*f").

Lorsque vous utilisez l'astérisque, l'argument suivant dans la liste (qui doit être obligatoirement une valeur entière) devient la valeur effectivement utilisée. Par exemple :

Format('%*.*f', [8, 2, 123.456])

équivaut à

Format('%8.2f', [123.456]).

Un spécificateur de taille définit la taille minimale du champ lors de la conversion. Si la chaîne résultante est de taille inférieure à la taille minimale définie, elle est comblée par des espaces afin d'accroître la taille du champ. Par défaut, le résultat est aligné à droite en faisant précéder la valeur d'espaces, mais si le spécificateur de format contient un indicateur d'alignement à gauche (un caractère "-" précédant le spécificateur de taille), le résultat est aligné à gauche par l'ajout d'espaces après la valeur.

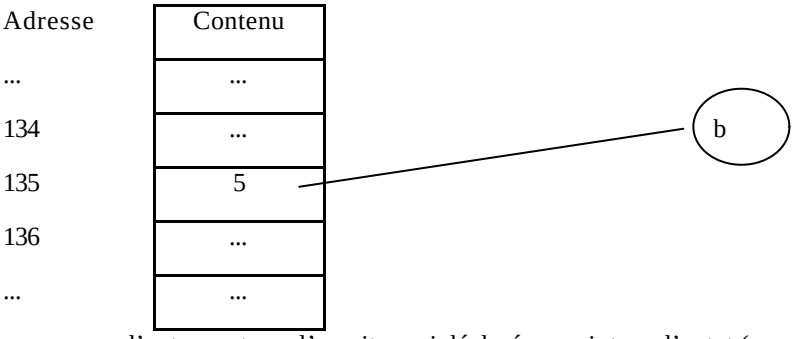
Un spécificateur d'indice définit la valeur courante de l'indice de la liste. L'indice du premier argument dans la liste est 0. A l'aide du spécificateur d'indice, vous pouvez formater un même argument plusieurs fois de suite. Par exemple "Format('%d %d %0:d %1:d', [10, 20])" produit la chaîne '10 20 10 20'.

Remarque : La définition du spécificateur d'indice affecte les formatages ultérieurs. Par exemple, Format('%d %d %d %0:d %d', [1, 2, 3, 4]) renvoie '1 2 3 1 2', et pas '1 2 3 1 4'. Pour obtenir ce dernier résultat, vous devez utiliser Format('%d %d %d %0:d %3:d', [1, 2, 3, 4]).

E. Le type pointeur

1. Définition

Un pointeur est une variable qui désigne une adresse mémoire. Quand un pointeur contient l'adresse d'une autre variable, on dit qu'il pointe sur l'emplacement en mémoire de cette variable ou sur les données qui y sont stockées. Dans le cas d'un tableau ou d'un type structuré, un pointeur contient l'adresse du premier élément de la structure.
Les pointeurs sont typés afin d'indiquer le type de données stockées à l'adresse qu'ils contiennent. Le type général Pointer peut représenter un pointeur sur tous les types de données alors que d'autres pointeurs, plus spécialisés, ne pointent que sur des types de données spécifiques. Les pointeurs occupent quatre octets en mémoire.
Le mot réservé *nil* est une constante spéciale qui peut être affectée à tout pointeur. Quand la valeur **nil** est affectée à un pointeur, le pointeur ne désigne plus rien.
Supposons que l'on ait déclaré une variable b de type byte (**var** b : byte;)et que 5 lui soit affecté (b:=5;) : Le système trouve une place en mémoire de 1 octet pour stocker cette valeur supposons que l'adresse trouvée soit 135. Voici une représentation partielle de la mémoire:



supposons d'autre part que l'on ait aussi déclaré un pointeur d'octet (**var** p : ^byte) l'affectation:

p := @b ; *// p vaut (pour notre exemple) 135 et p^ vaut 5 c'est à dire la valeur de b.*

2. Opérateurs

Opérateur	Opération	Types d'opérande	**Type** du résultat	Exemple
+	addition de pointeurs	pChar, entier	pChar	P + I
-	soustraction de pointeurs	pChar, entier	pChar, entier	P - Q
^	déréférencement de pointeur	pointeur	**type** de base du pointeur	P^
=	égalité	pointeur	Boolean	P = Q
<>	inégalité	pointeur	Boolean	P <> Q

L'opérateur ^ déréférence un pointeur. Son opérande peut être un pointeur de type quelconque sauf un pointeur générique (Pointer) qui doit être transtypé avant d'être déréférencé.
Le symbole ^ a deux fonctions, toutes deux illustrées dans l'exemple plus bas. Quand il apparaît avant un identificateur de type :
^nomType
il désigne un type qui représente des pointeurs sur des variables de type nomType. Quand il apparaît après une variable pointeur :
pointeur^
P = Q vaut True si P et Q pointent sur la même adresse ; sinon, P <> Q est True.

- L'opérateur **@** renvoie l'adresse d'une variable, d'une fonction, d'une procédure ou d'une méthode ; @ construit un pointeur sur son opérande. Pour davantage d'informations sur les pointeurs, voir Pointeurs et types de pointeurs.

3. routines d'adresses et de pointeurs

Addr, fonction.	Renvoie un pointeur sur un objet spécifique
Ptr, fonction	Convertit l'adresse spécifiée en pointeur

4. Routines d'allocation dynamique

Dispose, procédure	Libère la mémoire allouée à une variable dynamique
Finalize, procédure	Désinitialise une variable allouée dynamiquement
FreeMem, procédure	Libère une variable dynamique d'une taille donnée
GetMem, procédure	Crée une variable dynamique et un pointeur sur l'adresse du bloc
Initialize, procédure	Initialise une variable allouée dynamiquement
New, procédure	Crée une nouvelle variable dynamique et initialise P de telle façon qu'il pointe dessus.

5. Exemples

```
Var
  pb:^byte;  // un pointeur d'octet (8 bits)
  pw:^word; // un pointeur de mots(16 bits)
  p:pointer; // un pointeur non typé
  j:word;  // un mot (16 bits)
...
  j:=$10FF;  //  j reçoit la valeur 4351
  pw:=@j;  //  pw pointe sur l'adresse où est stocké j
{  pb:=pw;   //  provoque une erreur pour mélange de types}
  pb:=@j;  // alors que qu'ici : pas de problème
  p:=pw;  //  pas plus que là
  pb:=p;  //  ni là
  writeln(j,' = ',pw^);  //  pw est l'adresse de j, pw^ représente la valeur située à cette adresse soit j !
  writeln(pb^);  // Affiche 255 ($FF) les descendants du 8086 stockent les poids faibles avant les poids forts!
  pb:=pointer(longint(pb)+1);  // Il est nécessaire de transtyper, Delphi accepte pb := pb+1 uniquement pour les pChar (voir plus loin)
  writeln(pb^);  // affichage des poids faibles : 16 ($10)  essayer writeln((pb+1)^);
```

Autre exemple :

```
var
  p : ^byte ; // réserve 4 octets pour stocker la valeur d'une adresse; Mais PAS le contenu de ce que l'on veut stocker
...
new(p);  //  Réservation d'un emplacement mémoire capable de mémoriser un byte (soit 1 octet)
p^ := 17; // stockage de la valeur 17 à l'adresse trouvée ci-dessus.

Il aura donc fallu 5 octets pour mémoriser la valeur 17 : 4 pour l'adresse mémoire et 1 pour la valeur.
Alors que :
var
  b : byte;
```

```
...
  b:=17;
```

ne requière qu'un octet . Remarque @b (ou addr(b)) est codé sur 4 octets.

F. Autres pointeurs

Le type ***pointer*** est une adresse mémoire. ^***montype*** est un pointeur sur un type *montype*

Il existe aussi des pointeurs implicites tels que Pchar ou ansiString vus plus haut ou encore des tableaux dynamiques (que l'on verra plus bas) qu'il ne faut pas déréférencer (^).

Il existe aussi bien d'autres pointeurs . Pour plus d'informations voir l'aide Delphi.

G. Le type Variant

1. Définition

Il est parfois nécessaire de manipuler des données dont le type change ou est inconnu lors de la compilation. Dans ce cas, une des solutions consiste à utiliser des variables et des paramètres de type Variant qui représentent des valeurs dont le type peut changer à l'exécution. Les variants offrent une plus grande flexibilité que les variables standard mais consomment davantage de mémoire. De plus les opérations où les variants sont utilisés sont plus lentes que celles portant sur des types associés statiquement. Enfin, les opérations illégales portant sur des variants provoquent fréquemment des erreurs d'exécution, alors que les mêmes erreurs avec des variables normales sont détectées à la compilation.
À l'exception du type Int64, les variants peuvent tout contenir sauf les types structurés et les pointeurs.

Un variant occupe 16 octets de mémoire, il est constitué d'un code de type et d'une valeur ou d'un pointeur sur une valeur ayant le type spécifié par le code. Tous les variants sont initialisés à la création avec une valeur spéciale Unassigned. La valeur spéciale Null indique des données inconnues ou manquantes.
La fonction standard VarType renvoie le code de type d'un variant.

2. Opérateurs

ceux du type de base

3. Routines de gestion des variants

Null, variable	Null représente le variant null
Unassigned, constante	Utilisée pour indiquer qu'une variable Variant n'a pas encore été affectée d'une valeur
VarArrayCreate, fonction	Crée un tableau de variants
VarArrayDimCount, fonction	Renvoie le nombre de dimensions d'un tableau de variants
VarArrayHighBound, fonction	Renvoie la limite supérieure de la dimension donnée du tableau de variants
VarArrayLock, fonction	Ferme le tableau de variants et renvoie un pointeur sur les données
VarArrayLowBound, fonction	Renvoie la limite inférieure de la dimension donnée du tableau de variants
VarArrayOf, fonction	Crée et remplit un tableau de variants unidimensionnel
VarArrayRedim, procédure	Redimensionne un tableau de variants
VarArrayRef, fonction	Renvoie une référence au tableau de variants spécifié
VarArrayUnlock, procédure	Déverrouille un tableau de variants
VarAsType, fonction	Convertit un variant et le met dans le type spécifié
VarCast, procédure	Convertit un variant dans le type spécifié et stocke le résultat dans une variable
VarClear, procédure	Efface le variant spécifié afin qu'il ne soit pas affecté
VarCopy, procédure	Copie un Variant
VarFromDateTime, fonction	Renvoie un variant contenant la date-heure spécifiée
VarIsArray, fonction	Indique si le Variant spécifié est un tableau
VarIsEmpty, fonction	Indique si le Variant spécifié est Unassigned
VarIsNull, fonction	Indique si le Variant spécifié est Null
VarToDateTime, fonction	Convertit le variant spécifié en une valeur TDateTime
VarToStr, fonction	Convertit un variant en chaîne
VarType, fonction	Renvoie le code du type du variant spécifié
VarTypeToDataType, fonction	Renvoie la valeur du type de champ qui correspond le plus précisément à un type Variant.

H. Les types Ensemble

1. Définition

Un ensemble est une collection de valeurs ayant le même type scalaire. Les valeurs n'ont pas d'ordre intrinsèque, une même valeur ne peut donc pas apparaître deux fois dans un ensemble.
Les valeurs possibles du type ensemble sont tous les sous-ensembles du type de base, y compris l'ensemble vide. Le type de base ne peut avoir plus de 256 valeurs possibles et leur rang doit être compris entre 0 et 255. Toute construction de la forme :
set of <typeBase>
où <typeBase> est un type scalaire approprié, identifie un type ensemble.
En raison des limitations de taille des types de base, les types ensemble sont généralement définis avec des intervalles.

2. Opérateurs

Opérateur	Opération	Types d'opérande	**Type** du résultat	Exemple
+	union	ensemble	ensemble	ens1 + ens2
–	différence	ensemble	ensemble	S - T
*	intersection	ensemble	ensemble	S * T
<=	sous-ensemble	ensemble	Boolean	Q <= MonEns
>=	sur-ensemble	ensemble	Boolean	S1 >= S2
=	égalité	ensemble	Boolean	S2 = MonEns
<>	différence	ensemble	Boolean	MonEns <> S1
in	inclusion	scalaire, ensemble	Boolean	A **in** ens1

3. Exemples

```
Type
  tlettres='a'..'z';
  Elettres= set of Tlettres ;
var
  lettres,voyelles ,consonnes, vide,v2: Elettres;
begin
 voyelles :=  ['a','e','i','o','u','y'];
 v2:=['u','o','u','i','a','e']; // Le u est répété 2 fois  mais les doublons seront éliminés et il manque le y par rapport à voyelles
  lettres := ['a'..'z'];
  consonnes  := lettres - voyelles;
  vide := consonnes * voyelles;  // les 2 ensembles sont disjoints : leur intersection est vide
  writeln('f' in voyelles); // FALSE
{  writeln(voyelles); provoque une erreur }
 writeln(vide = [ ]);  // TRUE
{  writeln(v2 in voyelles);  // provoque une erreur }
  writeln(v2+['y']=voyelles); // TRUE
```

I. Les tableaux

1. Définition

Un tableau représente une collection indicée d'éléments de même type (appelé le type de base). Comme chaque élément a un indice unique, les tableaux (à la différence des ensembles) peuvent, sans ambiguïtés, contenir plusieurs fois la même valeur. Il est possible d'allouer des tableaux de manière statique ou dynamique.

2. Tableaux statiques

a) Définition

Les tableaux statiques sont désignés par des constructions de la forme :
array[typeIndex1, ..., typeIndexn] **of** typeBase
où chaque typeIndex est un type scalaire dont l'étendue de doit pas dépasser 2 Go. Comme les typeIndex indicent le tableau, le nombre d'éléments que le tableau peut contenir est limité par le produit de la taille des typeIndex. Pratiquement, les typeIndex sont en général des intervalles d'entiers.

Dans le cas le plus simple d'un tableau à une dimension, il n'y a qu'un seul typeIndex.

b) Exemples de tableaux à 1 dimension

```
Type
  Tlettres='a'..'z';
  Ttbl = array[boolean] of string[6];
  Tvoy=(a,e,o,i,u,y);  // n'a rien à voir avec
l'ensemble des voyelles ci-dessus
var
  t1:  array[byte] of Tlettres;
  t2 : array[tlettres] of byte;
  t3 : tTbl;
  t4 : array[Tvoy] of char;
...
  t1[19]:='e';
  t2['k']:= 11;
  t3[TRUE]:= 'salut';
  t4[o]:='R';
```

Exemple 2

```
Type
  Tindice = (x,y);  // Type énuméré pour éviter
les apostrophes  voir t2 ci-dessus
  Tpoint = array[x..y] of real;
Var
 P1: Tpoint;
...
P1[x] := 5.9;
P1[y] := 3;
```

c) Exemples de tableaux à plusieurs dimensions

```
Type
  Tvecteur = array [1..3] of real;
  Tmat1 = array[1..4] of Tvecteur;
  Tmat2 = array[1..4] of array[1..3] of real;
  Tmat3 = array[1..4,1..3] of real;
var
  m1: Tmat1;
  m2: Tmat2;
  m3: Tmat3;
  v : Tvecteur;
  m4 :array[1..4,1..3] of real;
...
  m1[2] := v;
{   m2[1]:= v;   m3[1]:= v;  // Provoquent une
erreur }
{  m2 := m1;   m3 := m1;   m4 := m1;   m4 := m3;
// Provoquent une erreur }
   m1[3][1] :=7.2
   m1[3,2] := 5.1;
   m2[3][1] :=7.2;
   m2[3,2] := 5.1;
   m3[3][1] :=7.2;
   m3[3,2] := 5.1;
```

3. Tableaux dynamiques

a) Définition

Les tableaux dynamiques n'ont pas de taille ou de longueur fixe. La mémoire d'un tableau dynamique est réallouée quand vous affectez une valeur au tableau ou quand vous le transmettez à la procédure SetLength. Les types de tableau dynamique sont désignés par des constructions de la forme :

array of <typeBase>

Si X et Y sont des variables du même type de tableau dynamique, X := Y fait pointer X sur le même tableau que Y. (Il n'est pas nécessaire d'allouer de la mémoire à X avant d'effectuer cette opération.) A la différence des chaînes ou des tableaux statiques, les tableaux dynamiques ne sont pas copiés quand ils vont être modifiés. Donc,

b) Fonctions et procédures

Routines	
copy (Fonctions)	Tronque une partie d'un tableau dynamique
high(Fonctions)	renvoie l'indice le plus élevé du tableau (c'est-à-dire Length – 1) . Dans le cas d'un tableau de longueur nulle, High renvoie –1 (avec cette anomalie que High < Low).
length (Fonctions)	renvoie le nombre d'éléments du tableau,
low(Fonctions)	renvoie 0
setLength (procédure)	Réserve de la mémoire

c) Exemples

- Exemple 1

```
t1 : array of byte;
```

```
t2 : array of array of real;
...
setLength(t1,3);
setLength(t2,4,3);
t1[2}:=17;
t2[3,1] := 3.6;
```

- Exemple2

```
var
  t : array of array of Byte;
...
 setLength(t,4);
 setlength(t[0],1);
 setlength(t[1],2);
 setlength(t[2],3);
 setlength(t[3],4);
 t[0,0] := 1;
 t[1,0] := 1;  t[1,1] := 1;
 t[2,0] := 1;  t[2,1] := 2;  t[2,2] := 1;
 t[3,0] := 1;  // Et ainsi de suite.  Ceci pourrait constituer le début du triangle de pascal.
```

- Exemple3

Après l'exécution du code suivant :

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

La valeur de A[0] est 2. Si A et B étaient des tableaux statiques, A[0] vaudrait toujours 1.
L'affectation d'un indice d'un tableau dynamique (par exemple, MonTableauFlexible[2] := 7) ne réalloue pas le tableau.
Les indices hors des bornes ne sont pas détectés à la compilation.

J. Le type record

Le mot enregistrement est souvent employé et peut prêter à confusion : Il ne s'agit pas d'enregistrer quoi que ce soit sur disque

1. Définition

Un **record** ou enregistrement (appelé aussi structure ou fiches dans certains langages) représente un ensemble de données hétérogènes. Chaque élément est appelé un champ ; la déclaration d'un type enregistrement spécifie le nom et le type de chaque champ. Une déclaration d'une variable de type enregistrement a la syntaxe suivante :

Var
nomVarEnregistrement = **record**
 Champ1: type1;
 ...
 Champn: typen;
end

où nomVarEnregistrement est un identificateur valide, où chaque type désigne un type, et chaque listeChamp est un identificateur valide ou une liste d'identificateurs délimitée par des virgules. Le point-virgule final est facultatif.

Pour accéder au champ *champi* de l'enregistrement , on utilise la notation des identificateurs qualifiés:(voir page 16)
nomVarEnregistrement.champi

2. Exemples

Exemple1:

```
type
  TDate = record
    Jour: 1..31;
    Mois: (Jan, Fev, Mar, Avr, Mai, Jun, Juil, Aou, Sep, Oct, Nov, Dec);
    Annee: Word;
  end;
Var
  D1,D2 : Tdate;
...
  D1.Jour := 10;
  D1.Mois := Mai;
  D1.annee := 2002;
  D2 := D1
```

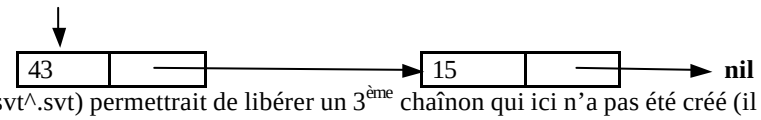
Exemple 2 : A comparer avec l'équivalent Tableau ci-dessus.
La règle veut que l'**on** utilise un tableau pour une collection d'éléments de même type et des records pour une collection d'éléments de types différents. Néanmoins :

```
Var
P2: Record
        x,y : Real;
    end;
...
P2.x := 5.9;
P2.y := 3;
```

Exemple 3 : listes chaînées:

```
Type
  TptListe = ^Tliste;
  Tliste =        record
                    valr : byte;
                    svt : TptListe
                  end;
var
  deb,p_cour : TptListe;
BEGIN
// initialisation
  new (p_cour);
  p_cour^.valr := 43;
  deb := p_cour;
  new (p_cour);
  deb^.svt := p_cour;
  p_cour^.valr := 15;
  p_cour^.svt := nil;  // inutile ici car un pointeur
non alloué est automatiquement à nil
//  Parcours
  p_cour := deb;
  writeln(p_cour^.valr);
  p_cour := p_cour^.svt;
  writeln(p_cour^.valr);
//  Libération de la mémoire
  dispose(deb^.svt);
  dispose(deb);
END.
```

Ce programme crée un chaînage de ce style :
deb



dispose(deb^.svt^.svt) permettrait de libérer un 3ème chaînon qui ici n'a pas été créé (il y a ***nil*** à sa place) . Dans cet exemple, cette instruction provoquerait une erreur d'exécution (et non de compilation car elle est correcte)

K. Enregistrements à partie variable

Voir l'aide de Delphi

L. Le type fichier

1. Définition

Un fichier est un ensemble ordonné d'éléments du même type. Les routines standard d'Entrées/Sorties utilisent les types prédéfinis TextFile et Text qui représentent un fichier contenant des caractères organisés en lignes. Pour davantage d'informations sur les entrées et sorties de fichier, voir Routines standard et d'E/S.
Pour déclarer un type fichier, utilisez la syntaxe :

type <nomTypeFichier> = file of <type>

où nom<TypeFichier> est un identificateur valide et type un type de taille fixe. Les types pointeur, implicites ou explicites ne sont pas permis. Un fichier ne peut donc pas contenir des tableaux dynamiques, des chaînes longues, des classes, des objets, des pointeurs, des variants, d'autres fichiers ou des types structurés en contenant.

2. routines d'entrées/sorties

Append, procédure	Prépare un fichier existant pour l'ajout de texte
BlockRead, procédure	Lit un ou plusieurs enregistrements d'un fichier ouvert et les place dans une variable
BlockWrite, procédure	Écrit un ou plusieurs enregistrements d'une variable mémoire dans un fichier ouvert
Eof, fonction	La fonction Eof détermine si la position en cours du pointeur se trouve en fin de fichier
FileMode, variable	Détermine le mode d'accès à utiliser lorsque des fichiers typés ou non typés sont ouverts avec la classe Reset
FilePos, fonction	Renvoie la position en cours dans un fichier
FileSize, fonction	Renvoie la taille d'un fichier (en octets) ou le nombre d'enregistrements dans le fichier
IOResult, fonction	Renvoie l'état de la dernière opération d'E/S
Input, variable	Spécifie un fichier en lecture seule associée à un périphérique d'entrée standard du système d'exploitation
MkDir, procédure	Crée un nouveau répertoire
Output, variable	Spécifie un fichier en écriture seulement associé à une sortie standard, généralement l'affichage
Rename, procédure	Renomme un fichier externe
Reset, procédure	Ouvre un fichier existant
Rewrite, procédure	Crée puis ouvre un nouveau fichier
RmDir, procédure	Supprime un sous-répertoire vide
Seek, procédure	Déplace la position en cours dans un fichier vers le composant spécifié
Truncate, procédure	Efface tous les enregistrements situés après la position en cours dans le fichier
Write, procédure (for typed files)	Écrit dans un fichier typé.

3. Routines de fichiers texte

AssignPrn, procédure	Affecte une variable fichier texte à l'imprimante
Eoln, fonction	Eoln détermine si la position en cours du pointeur se trouve en fin de ligne d'un fichier texte
Erase, procédure	Supprime un fichier externe
Flush, procédure	Efface le tampon associé à un fichier texte ouvert en écriture
Read, procédure	Read lit les données d'un fichier
Readln, procédure	Lit une ligne de texte dans un fichier
SeekEof, fonction	Renvoie l'état de fin d'un fichier
SeekEoln, fonction	Renvoie l'état de fin de ligne d'un fichier
SetTextBuf, procédure	Affecte un tampon d'E/S à un fichier texte
Write, procédure (for text files)	Écrit dans un fichier texte
Writeln, procédure	Place une marque de fin de ligne dans un fichier texte.

4. Routines de gestion de fichiers

AssignFile, procédure	Associe le nom d'un fichier externe à une variable fichier
ChDir, procédure	Change le répertoire en cours
CloseFile, procédure	Ferme l'association entre une variable fichier et un fichier disque externe (Delphi)

Constantes en mode ouverture de fichier	Les constantes de mode d'ouverture de fichier sont utilisées pour contrôler le mode d'accès a fichier ou au flux
Constantes mode de fichier	Les constantes mode de fichier sont utilisées pour ouvrir et fermer des fichiers disque
CreateDir, fonction	Crée un nouveau répertoire
DeleteFile, fonction	Supprime un fichier du disque
DiskFree, fonction	Renvoie le nombre d'octets disponibles sur le lecteur spécifié
DiskSize, fonction	Renvoie la taille en octets du disque spécifié
FileClose, procédure	Ferme le fichier spécifié
FileDateToDateTime, fonction	Convertit une valeur date ou heure DOS en valeur au format TDateTime
FileExists, fonction	Teste si le fichier spécifié existe.
FileGetAttr, fonction	Renvoie les attributs du fichier FileName
FileGetDate, fonction	Renvoie la date et l'heure DOS du fichier spécifié
FileOpen, fonction	Ouvre un fichier en utilisant le mode d'accès spécifié
FileRead, fonction	Lit le nombre d'octets spécifié dans un fichier
FileSearch, fonction	Recherche un fichier dans le chemin DOS spécifié
FileSeek, fonction	Positionne le pointeur d'un fichier préalablement ouvert
FileSetAttr, fonction	Définit les attributs du fichier spécifié
FileSetDate, fonction	Définit la marque horaire du fichier DOS spécifié
FileWrite, fonction	Ecrit le contenu du tampon à l'emplacement en cours dans un fichier
FindClose, procédure	Libère la mémoire allouée par FindFirst
FindFirst, fonction	Cherche la première occurrence d'un fichier avec un ensemble d'attributs précis dans un répertoire spécifié.
FindNext, fonction	Renvoie l'entrée suivante correspondant au nom et aux attributs spécifiés dans un précédent appel à FindFirst
GetCurrentDir, fonction	Renvoie le nom du répertoire en cours
GetDir, procédure	Renvoie le répertoire en cours sur le lecteur spécifié
RemoveDir, fonction	Efface un répertoire vide existant
RenameFile, fonction	Renomme un fichier
SetCurrentDir, fonction	Définit le répertoire en cours.

5. Exemples

```
Const Max=100;

type TDonnees=record
        Nom, Prenom : string[50]; // ne pas
utiliser des string, ansiString ou Pchar !!
        adresse:string[150];
        cdPostl:integer;
    end;
    TBase=record
      item:Array[1..Max] of TDonnees;
       nbitem :integer;
     end;

var
  fic : file of tBase;
  nomtb : tbase
...
// Pour enregistrer la base de données sur
disque
 assignfile(fic,'toto.tab');
 rewrite(fic);
 write(fic,montb);
 closefile(fic);
...
// Pour récupérer la base de données du
disque
 assignfile(fic,'toto.tab');
 reset(fic);
 read(fic,montb);
 closefile(fic);
```

ATTENTION:
Il serait très tentant d'utiliser:

```
type TDonnees=record
        Nom,Prenom, adresse:string;
        cdPostl:integer;
    end;
```

à la place de la déclaration ci-dessus. Cela éviterait de tronquer les chaînes ou de les surdimensionné (pour éviter un encombrement inutile de la mémoire). Il n'y aurait aucune différence de comportement (apparent) pour tout ce qui est stockage des données en mémoire.
Le problème apparaîtrait lors de l'instruction:

```
write(fic,montb);
```

en effet, on n'enregistrerait pas les valeurs des ***Nom,Prenom, adresse*** mais des pointeurs référençant les adresses de ces valeurs. Ce qui n'aurait aucun intérêt, puisque la création est dynamique et qu'une autre exécution sur le même ordinateur ne donnerait pas les mêmes adresses pour ces valeurs. Ne parlons pas d'une exécution sur une autre machine!
Et le compilateur s'en rend compte (et c'est heureux) et il le signale par une erreur à la compilation.

M. Les types procédure et fonction

1. Définition

Quand une variable procédurale se trouve dans la partie gauche d'une instruction d'affectation, le compilateur attend également une valeur procédurale à droite. L'affectation fait de la variable placée à gauche un pointeur sur la fonction ou la procédure indiquée à droite de l'affectation. Néanmoins, dans d'autres contextes, l'utilisation d'une variable procédurale produit un appel de la procédure ou de la fonction référencée. Vous pouvez même utiliser une variable procédurale pour transmettre des paramètres. Nous aurons l'occasion d'y revenir après avoir défini les procédures et fonctions

2. Exemple

```
var
  F: function(X: Integer): Integer;
```

N. Autres routines

1. routines de gestionnaire de mémoire

AllocMem, fonction	Alloue un bloc mémoire et initialise chaque octet à zéro
AllocMemCount, variable	Représente la taille totale des blocs de mémoire alloués dans une application
AllocMemSize, variable	Représente la taille totale des blocs de mémoire alloués
GetHeapStatus, fonction	Renvoie l'état actuel du gestionnaire de mémoire
GetMemoryManager, procédure	Renvoie les points d'entrée du gestionnaire de mémoire installé
HeapAllocFlags, variable	Indicateurs spécifiant comment le gestionnaire de mémoire obtient la mémoire depuis le système d'exploitation
IsMemoryManagerSet, fonction	Indique si le gestionnaire de mémoire a été surchargé en utilisant la procédure SetMemoryManager
ReallocMem, procédure	Réalloue une variable dynamique
SetMemoryManager, procédure	Définit les points d'entrée du gestionnaire de mémoire
SysFreeMem, fonction	Libère la mémoire sur laquelle pointe le pointeur spécifié
SysGetMem, fonction	Alloue un nombre spécifié d'octets et leur renvoie un pointeur
SysReallocMem, fonction	Renvoie un pointeur sur le nombre d'octets spécifié, préservant les valeurs pointées par le paramètre Pointer.

2. routines diverses

Assert, procédure	Teste la validité d'une expression booléenne
Assigned, fonction	Teste un pointeur **nil** (non affecté) ou une variable procédurale
Beep, procédure	Génère un bip standard sur le haut-parleur
Chr, fonction	Renvoie le caractère correspondant à une valeur ASCII
CollectionsEqual, fonction	Compare le contenu de deux collections
CompareMem, fonction	Effectue une comparaison binaire de deux images mémoire
DLLProc, variable	Pointe sur une procédure déclenchée par un point d'entrée d'une DLL
Default8087CW, variable	Default 8087 est le mot de contrôle par défaut
FillChar, procédure	Remplit une succession d'octets avec la valeur spécifiée
FormatMaskText, fonction	Renvoie une chaîne formatée à l'aide d'un masque d'édition
FreeAndNil, procédure	Libère une référence d'objet et la remplace par **nil**
Hi, fonction	Renvoie l'octet de poids fort de X comme valeur non signée
High, fonction	Renvoie la plus grande valeur dans l'étendue d'un argument
HtmlTable, fonction	Génère l'image HTML d'un ensemble de données à l'aide des propriétés et des événements d'un objet générateur de tableau
IsAccel, fonction	Indique si un caractère particulier est un caractère accélérateur (ou touche de raccourci) à l'intérieur d'un menu donné ou d'une autre chaîne de texte
IsValidIdent, fonction	Teste un identificateur Pascal
Lo, fonction	Renvoie l'octet de poids faible de l'argument X
Low, fonction	Renvoie la valeur la moins élevée d'une étendue d'arguments
MaxInt, constante	Valeur maximale du type de données Integer

MaxLongint, constante	Valeur maximale du type de données Longint
Move, procédure	Copie des octets de la source vers la destination
Printer, fonction	Renvoie une instance globale d'un TPrinter pour gérer l'interaction avec l'imprimante
Set8087CW, procédure	Définit à la fois le mot de contrôle dans l'unité virgule flottante et la variable Default8087CW déclarée dans l'unité System
SizeOf, fonction	renvoie le nombre d'octets occupés par une variable ou un type
Slice, fonction	Renvoie une sous-section d'un tableau
UniqueString, procédure	Vérifie qu'une chaîne donnée a un compteur de référence à 1
UpCase, fonction	Convertit un caractère en majuscules
ValidParentForm, fonction	Renvoie la fiche ou la page de propriétés qui contient le contrôle spécifié.

3. Informations au niveau de l'application.

Application, variable (pour les applications standard)	Représente les informations au niveau de l'application
CmdShow, variable	CmdShow est transmise à la routine ShowWindow de l'API Windows
HInstance, variable	Indique le handle fourni par Windows pour une application ou bibliothèque
HintWindowClass, variable	Indique la classe de la fenêtre utilisée pour l'affichage des conseils d'aide
IsConsole, variable	Indique si le module a été compilé en tant qu'application console
IsLibrary, variable	Indique si le module est une DLL
JITEnable, variable	Contrôle lorsque le débogueur juste à temps est appelé
Languages, fonction	Énumère les localisations pour lesquelles le support est disponible
MainInstance, variable	Le handle Instance pour l'exécutable principal
MainThreadID, variable	Le handle Instance pour le thread d'exécution principal des modules en cours
NoErrMsg, variable	Contrôle si l'application affiche un message d'erreur lorsqu'une erreur d'exécution se produit
PopupList, variable	Fournit une gestion centralisée des messages Windows adressés à des menus déroulants
Screen, variable	Représente un périphérique écran
Win32Platform, variable	Spécifie l'identificateur de la plate-forme Win32.

4. Routines de conversion de type

BCDToCurr, fonction	Convertit une valeur décimale codée binaire (BCD) en la valeur monétaire correspondante
Bounds, fonction	Renvoie le TRect d'un rectangle de dimensions données
CompToCurrency, fonction	Convertit une valeur Comp en une valeur Currency
CompToDouble, fonction	Convertit une valeur en une valeur double
CurrToBCD, fonction	Convertit une valeur monétaire en la valeur décimale codée binaire (BCD) correspondante
CurrencyToComp, procédure	CurrencyToComp convertit une valeur Currency en Comp
Point, fonction	Crée une structure point Windows avec un couple de coordonnées
Rect, fonction	Crée une structure TRect à partir de coordonnées fournies
StrToInt, fonction	Convertit en nombre une chaîne AnsiString qui représente un entier (décimal ou hexadécimal)
StrToInt64, fonction	Convertit en nombre une chaîne qui représente un entier (décimal ou hexadécimal)
StrToInt64Def, fonction	Convertit en nombre une chaîne qui représente un entier (décimal ou hexadécimal)
StrToIntDef, fonction	Convertit en nombre une chaîne qui représente un entier (décimal ou hexadécimal).

5. Routines de contrôle de flux

Abort, procédure	Permet de sortir d'un chemin d'exécution sans signaler d'erreur
Break, procédure	La procédure Break provoque l'interruption d'une boucle **for**, **while** ou **repeat**
Continue, procédure	Continue provoque le passage du contrôle de l'exécution à l'itération suivante dans une instruction **for**, **while** ou **repeat**
Exit, procédure	Quitte la procédure en cours
Halt, procédure	Exécute une fin anormale d'un programme
RunError, procédure	Interrompt l'exécution et génère une erreur d'exécution.

6. Utilitaires de ligne de commande

CmdLine, variable	CmdLine est un pointeur sur les arguments de la ligne de commande spécifié

	quand une application est appelée
FindCmdLineSwitch, fonction	Détermine si une chaîne de caractères a été transmise à l'application en tant qu'argument de la ligne de commande
ParamCount, fonction	Renvoie le nombre de paramètres passés dans la ligne de commande
ParamStr, fonction	Renvoie le paramètre spécifié depuis la ligne de commande.

7. Utilitaires com

ClassIDToProgID, fonction	Renvoie le PROGID d'un ID de classe (CLSID) spécifié
CoInitFlags, variable	Indique le niveau de gestion de threads requis pour un serveur .EXE COM
ComClassManager, fonction	Renvoie un objet TComClassManager
ComServer, variable	Fournit des informations sur la classe et le registre pour des objets serveur
CreateClassID, fonction	CreateClassID génère un nouveau GUID et le renvoie sous forme de chaîne de caractères
CreateComObject, fonction	Instancie une instance unique d'un objet COM
CreateOleObject, fonction	Instancie une instance unique d'un objet Automation
CreateRegKey, procédure	Crée ou ouvre une clé de base de registres qui est la clé secondaire de HKEY_CLASSES_ROOT
CreateRemoteComObject, fonction	Crée un objet COM sur une autre machine et renvoie une interface IUnknown pour cet objet
DeleteRegKey, procédure	Supprime une clé secondaire de HKEY_CLASSES_ROOT de la base de registres
DllGetClassObject, fonction	Utilisée pour obtenir un fabricant de classe pour un objet ActiveX lorsque l'objet ActiveX réside dans un serveur ActiveX en processus (DLL)
DllRegisterServer, fonction	Recense un serveur ActiveX en processus du module en cours dans la base des registres
EmptyParam, variable	Indique qu'un paramètre facultatif sur une interface double n'est pas utilisé
EnumDispatchProperties, procédure	Remplit un TstringList avec les noms de toutes les propriétés et les DispID d'une interface IDispatch spécifiée
FontToOleFont, fonction	Renvoie un Variant contenant une interface IFontDispatch représentant un objet TFont
GUIDToString, fonction	Convertit un GUID identificateur de classes en chaîne
GetActiveOleObject, fonction	Transmet une référence à une interface IDispatch à un objet COM actif et recensé
GetDispatchPropValue, fonction	Renvoie la valeur d'une propriété sur une interface IDispatch
GetOleFont, procédure	Crée un objet police OLE directement mappé à un TFont natif
GetOlePicture, procédure	Crée un objet image OLE directement mappé à un TPicture natif
GetOleStrings, procédure	Implémente un objet TStrings en tant qu'interface IStrings utilisable par des objets OLE
GetRegStringValue, fonction	Supprime une valeur stockée sous une clé secondaire de HKEY_CLASSES_ROOT de la base de registres
InterfaceConnect, procédure	Connecte une interface IConnectionPoint
InterfaceDisconnect, procédure	Déconnecte une interface IConnectionPoint précédemment connectée par la procédure InterfaceConnect
OleCheck, procédure	Déclenche une exception EOleSysError si le code de résultat indique une erreur
OleError, procédure	Déclenche une exception EOleSysError
OleFontToFont, procédure	Remplit une structure TFont pour représenter un IFontDispatch
OleStrToStrVar, procédure	Copie une chaîne de sa représentation COM dans une chaîne Pascal existante
OleStrToString, fonction	Copie les données reçues d'une interface COM dans une chaîne
ParkingWindow, fonction	Propose une fenêtre parent temporaire pour les contrôles activeX lorsque le conteneur n'est pas prêt à agir comme un parent
ProgIDToClassID, fonction	Renvoie l'ID de classe (le CLSID) correspondant à la chaîne spécifiée dans le paramètre ProgID
RegisterAsService, procédure	Recense un objet COM comme un service NT
RegisterComServer, procédure	RegisterComServer recense un serveur COM en processus avec le système d'exploitation
SetDispatchPropValue, procédure	Définit la valeur d'une propriété sur une interface IDispatch
SetOleFont, procédure	Connecte un objet police OLE à un objet TFont et copie ses propriétés vers TFont
SetOlePicture, procédure	Connecte un objet image OLE à un objet TPicture et copie ses propriétés vers TPicture
SetOleStrings, procédure	Utilise une interface IStrings pour attribuer le contenu d'un objet TStrings
StringToGUID, fonction	Convertit une chaîne en GUID

StringToOleStr, fonction	Alloue de la mémoire et copie une chaîne vers le format OLE
Supports, fonction	Indique si un objet donné ou l'interface Iunknown supporte une interface spécifiée.

8. Routines de compatibilité descendante

AddExitProc, procédure	N'existe que pour des raisons de compatibilité descendante
AppendStr, procédure	Ajoute une chaîne allouée dynamiquement à une chaîne existante
AssignStr, procédure	Affecte une nouvelle chaîne allouée dynamiquement au pointeur spécifié
Close, procédure	Ferme l'association entre une variable fichier et un fichier externe (Pascal)
DisposeStr, procédure	Libère un pointeur chaîne ayant été alloué avec NewStr
ExitCode, variable	Contient le code de sortie de l'application (fourni pour assurer une compatibilité descendante)
LoadStr, fonction	Charge une chaîne depuis le fichier exécutable de l'application
NewStr, fonction	Alloue une chaîne sur le tas
StrAlloc, fonction	Alloue un tampon pour une chaîne à zéro terminal et renvoie un pointeur sur son premier caractère
StrBufSize, fonction	Renvoie le nombre de caractères maximum pouvant être placé dans un tampon alloué par StrAlloc
StrDispose, procédure	Libère une chaîne
StrNew, fonction	Alloue de l'espace sur et copie une chaîne dans le tas, renvoyant un pointeur sur la chaîne
StrPas, fonction	Convertit une chaîne terminée par le caractère Null en une chaîne Pascal
Swap, fonction	Inverse les octets de poids fort avec les octets de poids faible d'un entier ou d'un mot.

9. Informations au niveau de l'application.

Application, variable (pour les applications standard)	Représente les informations au niveau de l'application
CmdShow, variable	CmdShow est transmise à la routine ShowWindow de l'API Windows
HInstance, variable	Indique le handle fourni par Windows pour une application ou bibliothèque
HintWindowClass, variable	Indique la classe de la fenêtre utilisée pour l'affichage des conseils d'aide
IsConsole, variable	Indique si le module a été compilé en tant qu'application console
IsLibrary, variable	Indique si le module est une DLL
JITEnable, variable	Contrôle lorsque le débogueur juste à temps est appelé
Languages, fonction	Énumère les localisations pour lesquelles le support est disponible
MainInstance, variable	Le handle Instance pour l'exécutable principal
MainThreadID, variable	Le handle Instance pour le thread d'exécution principal des modules en cours
NoErrMsg, variable	Contrôle si l'application affiche un message d'erreur lorsqu'une erreur d'exécution se produit
PopupList, variable	Fournit une gestion centralisée des messages Windows adressés à des menus déroulants
Screen, variable	Représente un périphérique écran
Win32Platform, variable	Spécifie l'identificateur de la plate-forme Win32.

10. routines de gestion des exceptions

DatabaseError, procédure	Crée et déclenche une exception EDatabaseError
DatabaseErrorFmt, procédure	Crée et déclenche une exception EDatabaseError avec un message d'erreur formaté
ErrorAddr, variable	Contient l'adresse d'une instruction qui a provoqué une erreur d'exécution
ErrorProc, variable	Pointe sur le gestionnaire d'erreur d'exécution RTL
ExceptAddr, fonction	Renvoie l'adresse à laquelle l'exception en cours a été déclenchée
ExceptObject, fonction	Renvoie une référence à l'objet associé à l'exception en cours
ExceptProc, variable	Pointe sur le gestionnaire d'exception RTL de niveau le plus bas
ExceptionErrorMessage, fonction	Formate un message d'erreur standard
OutOfMemoryError, procédure	Déclenche une exception EOutOfMemory
RaiseLastWin32Error, procédure	Déclenche une exception pour la dernière erreur Win32
SetErrorProc, fonction	Remplace le gestionnaire d'exceptions concernant les messages d'erreur obtenus par une connexion de socket Windows
ShowException, procédure	Affiche un message d'exception et son adresse physique
SysErrorMessage, fonction	Convertit des codes d'erreur d'API Win32 en chaînes

Win32Check, fonction	Vérifie la valeur renvoyée par un appel d'API Windows et déclenche éventuellement une exception.

11. Utilitaires de flux

FindClass, fonction	Trouve et renvoie une classe dérivée de TPersistent
FindClassHInstance, fonction	Renvoie le handle d'instance du module dans lequel un type de classe est défini
FindGlobalComponent, variable	Renvoie un composant conteneur du niveau le plus élevé
FindHInstance, fonction	Renvoie le handle d'instance du module contenant l'adresse spécifiée
FindResourceHInstance, fonction	Renvoie le handle d'instance du module ressource associé à un HINSTANCE spécifié
GetClass, fonction	Renvoie une classe persistante recensée à partir de son nom de classe
ObjectBinaryToText, procédure	Convertit la représentation binaire d'un objet en un texte à lecture plus facile
ObjectResourceToText, procédure	Convertit la représentation binaire d'une ressource en un texte à lecture plus facile
ObjectTextToBinary, procédure	Convertit une représentation littérale symbolique d'un objet en une version binaire utilisable pour enregistrer l'objet dans un flux fichier ou mémoire
ObjectTextToResource, procédure	Convertit une représentation texte symbolique d'un objet en sa représentation binaire interne
ReadComponentRes, fonction	Lit des composants et leurs propriétés dans la ressource Windows spécifiée
ReadComponentResEx, fonction	Lit un composant dans une ressource
ReadComponentResFile, fonction	Lit des composants et leurs propriétés dans le fichier de ressources Windows spécifié
RegisterClass, procédure	Recense une classe d'objet persistant pour que le type de classe puisse être retrouvé
RegisterClassAlias, procédure	Recense une classe qui est identique à une autre classe, à l'exception du nom
RegisterClasses, procédure	Recense un ensemble de classes
RegisterIntegerConsts, procédure	Recense les fonctions de conversion pour les identificateurs de chaînes qui représentent des valeurs de types
TypeInfo, fonction	Renvoie un pointeur sur les informations de type pour un identificateur de type
UnregisterClass, procédure	Dérecense une classe objet
UnregisterClasses, procédure	Dérecense un ensemble de classes
UnregisterModuleClasses, procédure	Dérecense toutes les classes définies dans le module spécifié
WriteComponentResFile, procédure	Place des composants et leur propriétés dans un fichier, dans un format de ressource Windows.

est évalué avant un opérateur de priorité plus basse, les opérateurs de même priorité étant évalués à partir de la gauche.

O. Constantes typées / Variables initialisées

1. définition

Les constantes typées, à la différences des vraies constantes, peuvent contenir des valeurs de type tableau, enregistrement, procédure ou pointeur. Les constantes typées ne peuvent intervenir dans des expressions constantes.
Dans l'état par défaut du compilateur *{$J+}*, il est même possible d'affecter de nouvelles valeurs à des constantes typées : elles se comportent alors essentiellement comme des variables initialisées. Mais si la directive de compilation *{$J–}* est active, il n'est pas possible à l'exécution de modifier la valeur des constantes typées ; en effet ce sont alors des variables en lecture seule.

Déclarez une constante typée de la manière suivante :

const identificateur: **type** = valeur

où identificateur est un identificateur valide, **type** est un type quelconque (sauf un type fichier ou variant) et valeur est une expression de type **type**.

2. exemples

```
const Max: Integer = 100;
```

```
const Chiffres: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

```
type TCube = array[0..1, 0..1, 0..1] of Integer;

const Labyrinthe : TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

crée un tableau appelé Labyrinthe ou :

```
Labyrinthe[0,0,0] = 0
Labyrinthe[0,0,1] = 1
Labyrinthe[0,1,0] = 2
Labyrinthe[0,1,1] = 3
```

```
Labyrinthe[1,0,0] = 4
Labyrinthe[1,0,1] = 5
Labyrinthe[1,1,0] = 6
Labyrinthe[1,1,1] = 7
```

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVecteur = array[0..1] of TPoint;
  TMois = (Jan, Fev, Mar, Avr, Mai, Jun, Jul, Aou,
Sep, Oct, Nov, Dec);
  TDate = record
    J: 1..31;
    M: TMois;
    A: 1900..1999;
  end;
const
  Origine: TPoint = (X: 0.0; Y: 0.0);
  Ligne: TVecteur = ((X: -3.1; Y: 1.5), (X: 5.8; Y:
3.0));
  UnJour: TDate = (J: 2; M: Dec; A: 1960);
```

P. Transtypage

1. définition

Il est parfois utile de traiter une expression ayant un type donné comme si elle était d'un type différent. Le transtypage permet de le faire en modifiant, temporairement, le type d'une expression. Par exemple, Integer('A') transtype le caractère A en un entier.
La syntaxe du transtypage est :
IdentificateurType(expression)
Le transtypage de variable peut apparaître des deux côtés d'une affectation.

2. exemple

```
Boolean(0)  // vaut FALSE
Color(2)  // la 3éme couleur si color est un type énuméré
I := Integer('A');  // affecte 65 à la variable I
Char(I)
Boolean(Compteur)
TUnTypeDefini(MaVariable)

var MonCar: char;
...
Shortint(MonCar) := 122;  // affecte le caractère z (ASCII 122) à MonCar.
```

V. Les routines

A. Procédure

1. définition

Déclaration des procédures
Procedure <nom_proc>***(*** <liste des Paramètres Formels avec leur **type**>***) ;***
<Déclaration des objets et des outils de la procédure
Structure identique à la déclaration des objets et des outils du programme >
Begin
<Instructions du corps de la procédure >
End ;

2. exemple

```
procedure stop;
begin
  writeln('Appuyez sur entrée') ;
  readln
end;
...
  stop;   // Appel de la procédure stop définie dans la zone des déclarations.
```

B. Fonctions

1. définition

Déclaration des fonctions
Function<nom_fonc>***(***<liste des P.F. avec leur **type**>***) :*** <**type** du résultat>***;***
<Déclaration des objets et des outils
Structure identique à la déclaration des objets et des outils du programme >
Begin
<Instructions Corps de la fonction>
result := <le résultat de la fonction>
End ;

2. exemple

```
function f(x : real) : real;
Begin
  result := x*x+2;   //   on aurait pu  écrire f := x*x +2 Mais on préférera utiliser la variable locale implicite result (impossible en turbo pascal)
End;

Var
 y : real;
...
y := f(3);
```

C. Paramètre donnée variable

1. définition

Il s'agit d'un passage de paramètres par valeur. **On** ne précède le nom du paramètre d'aucun mot (ni **var**, ni **const** ni **out**).

Le paramètre peut avoir une valeur avant l'appel, sa valeur peut être modifiée dans la procédure, la modification n'est pas transmise au retour.

2. exemple

```
procedure essai (x,y : byte);
```

```
begin
  y := x  // y prend la valeur de x
end;
Var
  a,b : byte:
...
a:= 3; b:= 2;
essai (a,b);  //  a et b conservent leurs valeurs
```

D. Paramètre Résultat

1. définition

On précède le nom du paramètre du mot **out**.

Le paramètre ne doit pas avoir une valeur avant l'appel, sa valeur doit être donnée dans la procédure, la modification est transmise au retour.

2. exemple

```
procedure essai (x:byte; out y : byte);
begin
  y := x  // y prend la valeur de x
end;
Var
  a,b : byte:
...
a:= 3; b:= 2;
essai (a,b);  //  a et b valent 3 mais la valeur initiale de b ( 2 ) n'a pas été transmise à y!
```

E. Paramètre Donnée/résultat

1. définition

Il s'agit d'un passage de paramètres par adresse. **On** précède le nom du paramètre du mot **var**.

Le paramètre peut avoir une valeur avant l'appel, sa valeur peut être modifiée dans la procédure, la modification est transmise au retour.

2. exemple

```
procedure essai (x:byte; var y : byte);
begin
  y := x  // y prend la valeur de x
end;
Var
  a,b : byte:
...
a:= 3; b:= 2;
essai (a,b);  //  a et b valent 3 et  la valeur initiale de b ( 2 ) a été transmise à y!
```

F. Paramètre Donnée Constante

1. définition

On précède le nom du paramètre du mot **Const**.

Le paramètre doit avoir une valeur avant l'appel, sa valeur ne peut pas être modifiée dans la procédure, et par conséquent la modification n'est pas transmise au retour.

2. exemple

```
procedure essai (x:byte; Const y : byte);
begin
  y := x  // provoquera une erreur : il est impossible de modifier la valeur d'un paramètre constant
end;
```

G. Paramètres facultatifs ou initialisés

1. définition

Il est possible de définir des routines (procédures et fonctions) avec n paramètres formels et de les utiliser avec p(<n) paramètres effectifs. Les n-p sont utilisés avec une valeur par défaut!; C'est par exemple le cas de la fonction prédéfinies inc qui s'utilise avec 1 ou 2 paramètres :

inc(i) équivaut à i:=i+1 alors que inc(i,4) équivaut à i:=i+4 . Il faut remarquer que l'utilisation de inc évite le double calcul d'adresse de i et le code généré est donc plus efficace.

Pour définir la valeur par défaut d'un paramètre, il suffit de faire suivre le **type** du paramètre par = suivi de sa valeur par défaut

2. exemple

Si la procédure inc n'était pas prédéfinie en delphi, **on** pourrait l'écrire comme ceci :

```
procedure incr (var n : byte;increment : byte = 1);
Begin
  n:= n+ incrément
End;
```

Remarque : bien entendu, la routine prédéfinie est plus performante et a surtout l'avantage d'admettre les types scalaires.

H. Paramètres sans type

1. définition

Dans le corps d'une procédure ou d'une fonction, les paramètres sans **type** sont incompatibles avec tous les types. Pour agir sur un paramètre sans **type**, vous devez le transtyper. En général, le compilateur ne peut vérifier si les opérations effectuées sur les paramètres sans **type** sont légales.

2. exemples

```
function Egal(var Source, Dest; Taille: Integer): Boolean;
type
  TOctets = array[0..MaxInt - 1] of Byte; // définition d'un type permettant le transtypage
var
  N: Integer;
begin
  N := 0;
  while (N < Taille) and (TOctets(Dest)[N] = TOctets(Source)[N]) do
    Inc(N);
  Egal := N = Taille;
end;
type
  TVecteur = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
```

```
  Vec1, Vec2: TVecteur;
  N: Integer;
  P: TPoint;
...
  Egal(Vec1, Vec2, SizeOf(TVecteur))       // compare Vec1 et Vec2
  Egal(Vec1, Vec2, SizeOf(Integer) * N)    // compare les N premiers éléments de Vec1 et Vec2
  Egal(Vec1[1], Vec1[6], SizeOf(Integer) * 5) // compare les 5 premiers éléments aux 5 derniers
éléments de Vec1
  Egal(Vec1[1], P, 4)                      // compare Vec1[1] à P.X et Vec1[2] à P.Y
```

Exemple 2

```
...
procedure permute(var a,b;taille:word);
var c : pointer;
begin
  getmem(c,taille);
  move(a,c^,taille);
  move(b,a,taille);    // a:=b déclanche une erreur!
  move(c^,b,taille);
  freemem(c,taille)
end;
var ch1,ch2 : string[20];
x,y:real;
BEGIN
  ch1:='bonjour';
  ch2:='salut';
  x:=3;y:=5.9;
  writeln(ch1,ch2,x,y);
  permute(x,y,sizeof(real));
  permute(ch1,ch2,21);
  writeln(ch1,ch2,x,y);
END.
```

I. Paramètres tableau ouvert

1. définition

Les paramètres tableau ouvert permettent de transmettre des tableaux de tailles différentes à la même routine. Pour définir une routine ayant un paramètre tableau ouvert, utilisez la syntaxe **array of type** (au lieu de **array**[X..Y] **of type**) dans la déclaration du paramètre

2. Exemple

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

J. Paramètres tableau ouvert variant

1. définition

Les paramètres tableau ouvert variant permettent de transmettre un tableau d'expressions de types différents, à une seule routine. Pour définir une routine utilisant un paramètre tableau ouvert variant, spécifiez **array of const** comme **type** du paramètre. Ainsi :

procedure FaireQuelquechose(A: **array of const**);

déclare une procédure appelée FaireQuelquechose qui peut agir sur des tableaux de données hétérogènes.
La construction **array of const** est équivalente à **array of** TVarRec. TVarRec, déclaré dans l'unité System, représente un enregistrement avec une partie variable qui peut contenir des valeurs de **type** entier, booléen, caractère, réel, chaîne, pointeur, classe, référence de classe, **interface** et variant. Le champ VType de TVarRec indique le **type** de chaque élément du tableau. Certains types sont transmis comme pointeur et non comme valeur ; en particulier les chaînes longues sont transmises comme Pointer et doivent être transtypées en **string**.

2. Exemple

L'exemple suivant utilise un paramètre tableau ouvert variant dans une fonction qui crée une représentation sous forme de chaîne de chaque élément transmis et concatène le résultat dans une seule chaîne. Les routines de manipulation de chaînes utilisées dans cette **function** sont définies dans l'unité SysUtils.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
...
MakeStr(['test', 100, ' ', True, 3.14159, Tform]) ; //  renvoie la chaîne "test100 T3.14159TForm".
```

K. Appel de procédure et de fonctions

1. définition

L'appel (utilisation) d'une procédure s'effectue comme pour toute instruction en indiquant le nom de la procédure

Par défaut, Sauf directive de compilation, il est possible d'ignorer le résultat d'une fonction. C'est le cas de fausses procédures :les fonctions à effet de bord et qui donne un compte rendu booléen (ou autre)

2. Exemple

```
function exemple : boolean;
begin
 // le corps de la procédure
  résult := true; // la procédure s'est bien déroulée
end;
var tst : boolean;
```

```
...
  tst := exemple ; // utilisation normale d'un prédicat
..
..exemple ;// il est possible d'utiliser la fonction comme une procédure si l'on ne souhaite pas tenir compte du « compte-rendu »
```

Exemples :

```
procedure P1 (ch : string[20]);  // erreur de syntaxe
procedure P2 (ch : Array[1..20] of  byte);  // erreur de syntaxe
procedure P3 (ch : string);  // pas d'erreur de syntaxe
procedure P4 (ch : Array of  byte);  // pas d'erreur de syntaxe
Type
  Tch20 = string[20;
  Ttab = Array[1..20] of byte;

procedure P1 (ch : Tch20);  // pas d'erreur de syntaxe
procedure P2 (ch : Ttab);  // pas d'erreur de syntaxe
```

Exemple pratique :

```
Var t : array[1..2] of byte;
som : byte;
function Somme(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Somme := S;
end;
...
t[1] := 5;t[2] := 11;
  som := somme(t);
...
  som := somme([3,8,9,6]);
```

L. Retour sur le type procédure ou fonction

1. définition

Comme en LISP ou en Scheme, les routines sont des types comme les autres et peuvent donc être affectés ou passés en paramètres.

2. Exemple

```
{$F+}
function plus( a : integer; b : integer):integer;
Begin
  result := a+b
End;

function fois( m : integer; n : integer):integer;
Begin
  result:= m*n
End;
type Toper = function (x : integer; y :
integer):integer;
    Ttab = array[1..5] of byte;
```

```
function somprod(tb : Ttab; oper :
Toper):integer;
var i:integer;
Begin
  result:=oper(1,1);
  for i:=low(tb) to high(tb) do
    result:=oper(result,tb[i]);
  somprod:= result;
End;

function som(tb: Ttab):integer;
Begin
  result:=somprod(tb,plus)-1
End;
```

```
function prod(tb: Ttab):integer;
Begin
  result:=somprod(tb,fois)
End;

Const
  t:Ttab=(3,6,7,2,3);

BEGIN
  writeln(som(t));
  writeln(prod(t));
  readln
END.
```

3. Exercice

Exercice sur les paramètres de **type** fonction qui peut être traité même sans comprendre le fonctionnement de l'algorithme qui lui pourra être étudié après le paragraphe : « Structure de contrôle»

Soit le programme:

```
program QSort;
{$apptype console}
const
  Max = 1000;

type
  List = array[1..Max] of Integer;

var
  Data: List;
  I: Integer;

procedure QuickSort(var A: List; Bas, Haut:
Integer);

  procedure Sort(l, r: Integer);
  var
    i, j, x, y: integer;

  Begin
    i := l; j := r; x := a[(l+r) DIV 2];
    repeat
      while a[i] < x do i := i + 1;
      while x < a[j] do j := j - 1;
      if i <= j then begin
        y := a[i]; a[i] := a[j]; a[j] := y;
        i := i + 1; j := j - 1;
      end;
    until i > j;
    if l < j then Sort(l, j);
    if i < r then Sort(i, r);
  End{Sort};

Begin {QuickSort};
  Sort(Bas,Haut);
End{QuickSort};

BEGIN {QSort}
  Write('Generation de 1000 nombres aléatoires...');
  Randomize;
  for i := 1 to Max do Data[i] := Random(30000);
  Writeln;
  Write('Tri de ces nombres...');
  QuickSort(Data, 1, Max);
  Writeln;
  for i := 1 to 1000 do Write(Data[i]:8);
  readln;
END.
```

{ Ce programme génère une liste de 1000 nombres aléatoires entre 0 et 29999, puis les trie en utilisant l'algorithme du TRI-RAPIDE. Finalement affiche le résultat du tri à l'écran. }

{ QUICKSORT trie les éléments d'un tableau A d'indices entre Bas et Haut (bornes inclues) et ce de façon récursive. L'algorithme consiste à choisir un élément (appelé pivot) de la liste et de placer avant lui, tous ceux qui sont plus petits et après tous ceux qui sont plus grands. Cet élément se retrouvera donc à sa place. On recommence avec le sous-tableau devant cet élément et le sous tableau après. On peut choisir au hasard le pivot (premier, dernier, aléatoirement...) mais une optimisation consiste à utiliser la méthode ci-dessous}

Il est simple de transformer la procédure QUICKSORT en lui ajoutant un paramètre fonction qui est en fait une relation d'ordre que l'**on** utilisera à la place de <.

VI. Structure de contrôle

A. La séquence d'instructions et l'instruction composée

Les instructions sont exécutées les unes à la suite des autres en séquence. Elles sont séparées par des points-virgules.

Pour obtenir une instruction composée de plusieurs instructions, il faut les encadrer d'un ***begin*** et d'un ***end*** (à la manière d'un parenthèsage en mathématiques) ce qui est utile pour les structures de test ou de boucles qui ne réalisent qu'une instruction simple ou composée.

B. L'instruction Si

1. définition

L'instruction si a deux formes : **if**...**then** et **if**...**then**...**else**. La syntaxe de l'instruction **if**...**then** est :
if expression **then** instruction
où expression renvoie une valeur booléenne. Si expression vaut True, alors instruction est exécutée ; sinon elle ne l'est pas.
Par exemple :

```
if J <> 0 then Resultat := I/J;
```

La syntaxe de l'instruction **if**...**then**...**else** est :
if expression **then** instruction1 **else** instruction2
où expression renvoie une valeur booléenne. Si expression vaut True, alors instruction1 est exécutée ; sinon instruction2 est exécutée.
Par exemple :

```
if J = 0 then
  Exit
else
  Resultat := I/J;
```

Les clauses **then** et **else** contiennent une seule instruction chacune, mais ce peut être une instruction structurée.Par exemple :

```
if J <> 0 then begin
  Resultat := I/J;
  Compteur := Compteur + 1;
end else if Compteur = Fin then
  Arret := True
else
  Exit;
```

Remarquez qu'il n'y a jamais de point-virgule entre la clause **then** et le mot **else**. Vous pouvez placer un point-virgule après une instruction **if** pour la séparer de l'instruction suivante du bloc mais les clauses **then** et **else** ne nécessitent rien d'autre qu'un espace ou un passage à la ligne entre elles. Le fait de placer un point-virgule immédiatement avant le **else** (dans une instruction **if**) est une erreur de programmation courante.
Un problème particulier se présente quand des instructions **if** sont imbriquées. Le problème se pose car certaines instructions **if** ont une clause **else** alors que d'autres ne l'ont pas, mais la syntaxe des deux variétés de l'instruction est pour le reste la même. Dans une série de conditions imbriquées où il y a moins de clauses **else** que d'instructions **if**, il n'est pas toujours évident de savoir à quel **if** une clause **else** est rattachée. Soit une instruction de la forme

if expression1 **then if** expression2 **then** instruction1 **else** instruction2;

Il y a deux manières d'analyser cette instruction :

if expression1 **then** [**if** expression2 **then** instruction1 **else** instruction2];
if expression1 **then** [**if** expression2 **then** instruction1] **else** instruction2;

Le compilateur analyse toujours de la première manière. C'est-à-dire que dans du véritable code, l'instruction :

est équivalent à :

if ... *{ expression1 }* **then**
  if ... *{ expression2 }* **then**
  ... *{ instruction1 }*
  else
  ... *{ instruction2 }* ;
	if ... *{ expression1 }* **then begin**
	  if ... *{ expression2 }* **then**
	  ... *{ instruction1 }*
	  else
	  ... *{ instruction2 }*
	end;

La règle veut que les conditions imbriquées sont analysées en partant de la condition la plus interne, chaque **else** étant lié au plus proche **if** disponible à sa gauche. Pour forcer le compilateur à lire notre exemple de la deuxième manière, vous devez l'écrire explicitement de la manière suivante :

if ... *{ expression1 }* **then begin**
 if ... *{ expression2 }* **then**
  ... *{ instruction1 }*
end else
 ... *{ instruction2 }* ;

C. Instructions Case

1. définition

L'instruction **case** propose une alternative plus lisible à l'utilisation de conditions **if** imbriquées complexes. Une instruction **case** a la forme

case expressionSelection **of**
 listeCas1: instruction1;
 ...
 listeCasn: instructionn;
end

où expressionSelection est une expression de **type** scalaire (les types chaîne sont interdits) et chaque listeCas est l'un des éléments suivants :
Un nombre, une constante déclarée ou une expression que le compilateur peut évaluer sans exécuter le programme. Ce doit être une valeur de **type** scalaire compatible avec expressionSelection. Ainsi 7, True, 4 + 5 * 3, 'A', et Integer('A') peuvent être utilisés comme listeCas, mais les variables et la plupart des appels de fonctions ne peuvent être utilisés.
Un intervalle de la forme Premier..Dernier, où Premier et Dernier respectent tous les deux les critères précédents et où Premier est inférieur ou égal à Dernier.
Une liste de la forme élément1, ..., élémentn, où chaque élément respecte l'un des critères précédents.
Chaque valeur représentée par une listeCas doit être unique dans l'instruction **case** ; les intervalles et les listes ne peuvent se chevaucher. Une instruction **case** peut avoir une clause **else** finale :
case expressionSelection **of**
 listeCas1: instruction1;
 ...
 listeCasn: instructionn;
else
 instruction;
end

Le **premier** listeCas dont la valeur est égale à celle de expressionSelection détermine l'instruction à utiliser. Si aucun des listeCas n'a la même valeur que expressionSelection, alors c'est l'instruction de la clause **else** (si elle existe) qui est exécutée.

2. exemples

L'instruction **case** :

```
case I of
  1..5: Caption := 'Bas';
  6..9: Caption := 'Baut';
  0, 10..99: Caption := 'Hors de l''intervalle';
else
  Caption := '';
end;
```

est équivalent à la condition imbriquée suivante :

```
if I in [1..5] then
  Caption := 'Bas'
  else if I in [6..10] then
    Caption := 'Haut'
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Hors de l''intervalle'
      else
        Caption := '';
```

Attention :

```
var i,x :byte;
..
```

```
i:=3;
case i of
```

```
  2..5 : x:=2;
  1..4 : x:=1;
 end;
 // ici x vaut 2 bien que 3 appartienne aussi à l'intervalle 1..4, alors que

i:=3;
```

```
 case i of
  1..4 : x:=1;
  2..5 : x:=2;
end;
 // ici x vaut 1 bien que 3 appartienne aussi à l'intervalle 2..5, seule le premier test valide compte !
```

Voici d'autres exemples d'instructions **case** :

```
case MaCouleur of
 Rouge: X := 1;
 Vert: X := 2;
 Bleu: X := 3;
 Jaune, Orange, Noir: X := 0;
end;
```

```
case Selection of
 Fin: Form1.Close;
 Calcul: CalculTotal(CourUnit, Quant);
else
 Beep;
end;
```

D. La boucle Répéter

1. définition

L'instruction **repeat** a la syntaxe suivante :
repeat instruction1; ...; instructionn; **until** expression
où expression renvoie une valeur booléenne. Le dernier point-virgule avant **until** est facultatif. L'instruction **repeat** exécute répétitivement la séquence d'instructions qu'elle contient en testant expression à chaque itération. Quand expression renvoie True, l'instruction **repeat** s'arrête. La séquence est toujours exécutée au moins une fois car expression n'est évaluée qu'après la première itération.

2. exemples

```
repeat
 K := I Mod J;
 I := J;
 J := K;
until J = 0;
```

```
repeat
 Write('Entrez une valeur (0..9): ');
 Readln(I);
until (I >= 0) and (I <= 9);
```

E. La boucle tant-que

1. définition

Une instruction **while** est similaire à l'instruction **repeat** à cette différence près que la condition de contrôle est évaluée avant la première itération de la séquence d'instructions. Donc si la condition est fausse, la séquence d'instructions n'est jamais exécutée.
L'instruction **while** a la syntaxe suivante :
while expression **do** instruction
où expression renvoie une valeur booléenne et instruction peut être une instruction composée. L'instruction **while** exécute répétitivement son instruction, en testant expression avant chaque itération. Tant que expression renvoie True, l'exécution se poursuit.

2. exemples

```
while I > 0 do begin
 if Odd(I) then Z := Z * X;
 I := I div 2;
 X := Sqr(X);
end;
```

```
while not Eof(FicSource) do begin
 Readln(FicSource, Ligne);
 Process(Ligne);
end;
while Data[I] <> X do I := I + 1;
```

F. La boucle Pour

1. définition

Une instruction **for**, à la différence des instructions **repeat** et **while**, nécessite la spécification explicite du nombre d'itérations que la boucle doit effectuer. L'instruction **for** a la syntaxe suivante :

for compteur := valeurInitiale **to** valeurFinale **do** instruction

ou

for compteur := valeurInitiale **downto** valeurFinale **do** instruction

où compteur est une variable locale (déclarée dans le bloc contenant l'instruction **for**) de **type** scalaire sans aucun qualificateur.

- valeurInitiale et valeurFinale sont des expressions compatibles pour l'affectation avec compteur.
- instruction est une instruction simple ou structurée qui ne modifie pas la valeur de compteur.

L'instruction **for** affecte la valeur valeurInitiale à compteur, puis exécute répétitivement instruction, en incrémentant ou en décrémentant compteur après chaque itération. La syntaxe **for**...**to** incrémente compteur alors que la syntaxe **for**...**downto** le décrémente. Quand compteur renvoie la même valeur que valeurFinale, l'instruction est exécutée une dernière fois puis l'instruction **for** s'arrête. En d'autres termes, instruction est exécutée une fois pour chaque valeur de l'intervalle allant de valeurInitiale à valeurFinale. Si valeurInitiale est égale à valeurFinale, instruction est exécutée une seule fois. Si valeurInitiale est supérieure à valeurFinale dans une instruction **for**...**to** ou inférieure ou égale à valeurFinale dans une instruction **for**...**downto**, alors l'instruction n'est jamais exécutée. Après l'arrêt de l'instruction **for**, la valeur de compteur est non définie.

Afin de contrôler l'exécution de la boucle, la valeur des expressions valeurInitiale et valeurFinale n'est évaluée qu'une seule fois, avant le commencement de la boucle. Donc, une instruction **for**...**to** est presque identique à la construction **while** suivante :

```
begin
  compteur := valeurInitiale;
  while compteur <= valeurFinale do  begin
    instruction;
    compteur := Succ(compteur);
  end;
end
```

La différence entre cette construction et l'instruction **for**...**to** est que la boucle **while** réévalue valeurFinale avant chaque itération. Cela peut réduire la vitesse d'exécution de manière sensible si valeurFinale est une expression complexe. De plus, cela signifie qu'une modification de la valeur valeurFinale dans instruction peut affecter l'exécution de la boucle.

2. exemples

```
for I := 2 to 63 do
  if Donnees[I] > Max then
    Max := Donnees[I];
```

```
for I := ListBox1.Items.Count - 1 downto 0 do
    ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
```

```
for I := 1 to 10 do
  for J := 1 to 10 do   begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;
```

```
for C := Red to Blue do Verif(C);
```

G. Break, Exit et Halt

- Halt : Interrompt l'exécution du programme et rend le contrôle au système d'exploitation.
- Exit : : Lorsque Exit est appelée dans un sous-programme (procédure ou fonction), elle induit un retour immédiat à l'appelant. Quand elle est appelée en tant qu'instruction du corps de programme principal, elle provoque la fin de l'exécution du programme.

- Break : Provoque la fin immédiate d'une boucle **FOR**, **WHILE** ou **REPEAT**
- Continue : permet de reprendre un boucle interrompue.

H. Boucles infinies

Il va de soi que si les instructions du corps de boucles ne modifie pas le test de boucle, on obtiendra une boucle infinie

Pour réaliser volontairement une boucle Infinie, on peut utiliser l'une des structures:

```
Repeat
  // instructions
  if <test> then break
  // instructions
until FALSE
```

ou encore:

```
While TRUE do begin
  // instructions
  if <test> then break
  // instructions
end {while}
```

I. Les Exceptions

Une exception est déclenchée quand une erreur ou un autre événement interrompt le déroulement normal d'un programme. L'exception transfère le contrôle à un gestionnaire d'exceptions, ce qui vous permet de séparer la logique normale d'exécution du programme de la gestion des erreurs. Comme les exceptions sont des objets, elles peuvent être regroupées en hiérarchies en utilisant l'héritage et de nouvelles exceptions peuvent être ajoutées sans affecter le code existant. Une exception peut véhiculer des informations, par exemple un message d'erreur, depuis le point où elle est déclenchée jusqu'au point où elle est gérée.
Quand une application utilise l'unité SysUtils, toutes les erreurs d'exécution sont automatiquement converties en exceptions. Les erreurs qui autrement provoqueraient l'arrêt d'une application (mémoire insuffisante, division par zéro, erreurs de protection générales) peuvent ainsi être interceptées et gérées.

1. Instructions Try...except

Les exceptions sont gérées dans des instructions **try**...**except**. Par exemple :

```
try
  X := Y/Z;
except
  on EZeroDivide do GereDivisionParZero;
end;
```

Cette instruction tente de diviser Y par Z mais appelle la routine appelée GereDivisionParZero si une exception EZeroDivide est déclenchée.
L'instruction **try**...**except** a la syntaxe suivante :
try instructions **except** blocException **end**
où instructions est une suite d'instructions, délimitée par des points-virgule et blocException est :
une autre suite d'instruction ou
une suite de gestionnaires d'exceptions, éventuellement suivie par :
else instructions
Un gestionnaire d'exception a la forme :
on identificateur: **type do** instruction
où identificateur: est facultatif (si identificateur est précisé, ce doit être un identificateur valide), **type** est le **type** utilisé pour représenter les exceptions et instruction est une instruction quelconque.
Une instruction **try**...**except** exécute les instructions dans la liste initiale instructions. Si aucune exception n'est déclenchée, le bloc exception (blocException) n'est pas pris en compte et le contrôle passe à l'instruction suivante du programme.
Si une exception est déclenchée lors de l'exécution de la liste instructions initiale, que ce soit par une instruction **raise** dans la liste instructions ou par une procédure ou une fonction appelée dans la liste instructions, il va y avoir une tentative de "gestion" de l'exception :
Si un des gestionnaires du bloc exception ne correspond à l'exception, le contrôle passe au premier d'entre eux. Un gestionnaire d'exceptions "correspond" à une exception si le **type** du gestionnaire est la classe de l'exception ou un ancêtre de cette classe.
Si aucun gestionnaire correspondant n'est trouvé, le contrôle passe à l'instruction de la clause **else** si elle est définie.

Si le bloc d'exception est simplement une suite d'instructions sans gestionnaire d'exception, le contrôle passe à la première instruction de la liste.
Si aucune de ces conditions n'est respectée, la recherche continue dans le bloc exception de l'avant-dernière instruction **try**...**except** dans laquelle le flux du programme est entré et n'est pas encore sorti. Si, là encore, il n'y ni gestionnaire approprié, ni clause **else**, ni liste d'instructions, la recherche se propage à l'instruction en cours **try**...**except** précédente, etc. Si l'instruction **try**...**except** la plus éloignée est atteinte sans que l'exception soit gérée, le programme s'interrompt.
Quand l'exception est gérée, le pointeur de la pile est ramené en arrière jusqu'à la procédure ou la fonction contenant l'instruction **try**...**except** où la gestion a lieu et le contrôle d'exécution passe au gestionnaire d'exception exécuté, à la clause **else** ou à la liste d'instructions. Ce processus efface tous les appels de procédure ou de fonction effectués à partir de l'entrée dans l'instruction **try**...**except** où l'exception est gérée. L'objet exception est alors automatiquement détruit par un appel de son destructeur Destroy et le contrôle revient à l'instruction suivant l'instruction **try**...**except**. Si un appel des procédures standard Exit, Break ou Continue force la sortie du gestionnaire d'exception, l'objet exception est quand même détruit automatiquement.
Dans l'exemple suivant, le premier gestionnaire d'exceptions gère les exceptions division-par-zéro, le second gère les exceptions de débordement et le dernier gère toutes les autres exceptions mathématiques. EMathError apparaît en dernier dans le bloc exception car c'est l'ancêtre des deux autres classes d'exception : s'il apparaît en premier, les deux autres gestionnaires ne sont jamais utilisés.

```
try
...
except
  on EZeroDivide do GereDivisionParZero;
  on EOverflow do GereDebordement;
  on EMathError do GereErreurMath;
end;
```

Un gestionnaire d'exceptions peut spécifier un identificateur avant le nom de la classe exception. Cela déclare l'identificateur représentant l'objet exception pendant l'exécution de l'instruction suivant **on**...**do**. La portée de l'identificateur est limitée à celle de l'instruction. Par exemple :

```
try
 ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

Si le bloc exception spécifie une clause **else**, la clause **else** gère toutes les exceptions qui ne sont pas gérées par les gestionnaires du bloc. Par exemple :

```
try
...
except
  on EZeroDivide do GereDivisionParZero;
  on EOverflow do GereDebordement;
  on EMathError do GereErreurMath;
else
  GereLesAutres;
end;
```

Ici la clause **else** gère toutes les exceptions qui ne sont pas des erreurs mathématiques (EMathError).
Si le bloc exception ne contient pas de gestionnaires d'exceptions mais une liste d'instructions, cette liste gère toutes les exceptions. Par exemple :

```
try
...
except
  GereException;
end;
```

Ici la routine GereException gère toutes les exceptions se produis ant lors de l'exécution des instructions comprises entre **try** et **except**.
Redéclenchement d'exceptions
Quand le mot réservé **raise** apparaît dans un bloc exception sans être suivi d'une référence d'objet, il déclenche l'exception qui était gérée par le bloc. Cela permet à un gestionnaire d'exception de répondre à une erreur d'une manière partielle, puis de redéclencher l'exception. Cela est pratique quand une procédure ou une fonction doit "faire le ménage" après le déclenchement d'une exception sans pouvoir gérer complètement l'exception.

Par exemple, la fonction GetFileList alloue un objet TStringList et le remplit avec les noms de fichiers correspondant au chemin de recherche spécifié :

```
function GetFileList(const Path: string):
TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

GetFileList crée un objet TStringList puis utilise les fonctions FindFirst et FindNext (définies dans SysUtils) pour l'initialiser. Si l'initialisation échoue (car le chemin d'initialisation est incorrect ou parce qu'il n'y a pas assez de mémoire pour remplir la liste de chaînes), c'est GetFileList qui doit libérer la nouvelle liste de chaînes car l'appelant ne connaît même pas son existence. C'est pour cela que l'initialisation de la liste de chaînes se fait dans une instruction **try**...**except**. Si une exception a lieu, le bloc exception de l'instruction libère la liste de chaînes puis redéclenche l'exception.

Exceptions imbriquées

Le code exécuté dans un gestionnaire d'exceptions peut lui aussi déclencher et gérer des exceptions. Tant que ces exceptions sont également gérées dans le gestionnaire d'exceptions, elles n'affectent pas l'exception initiale. Par contre, si une exception déclenchée dans un gestionnaire d'exceptions commence à se propager au-delà du gestionnaire, l'exception d'origine est perdue. Ce phénomène est illustré par la fonction Tan suivante.

```
type
  ETrigError = class(EMathError);
...
function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Argument incorrect pour Tan');
  end;
end;
```

Si une exception EMathError se produit lors de l'exécution de Tan, le gestionnaire d'exceptions déclenche une exception ETrigError. Comme Tan ne dispose pas de gestionnaire pour ETrigError, l'exception se propage au-delà du gestionnaire d'exceptions initial, ce qui provoque la destruction de l'objet exception EMathError. Ainsi, pour l'appelant, tout se passe comme si la fonction Tan avait déclenché une exception ETrigError.

2. Instructions try...finally

Dans certains cas, il est indispensable que certaines parties d'une opération s'effectuent, que l'opération soit ou non interrompue par une exception. Si, par exemple, une routine prend le contrôle d'une ressource, il est souvent important que cette ressource soit libérée quelle que soit la manière dont la routine s'achève. Vous pouvez, dans ce genre de situations, utiliser une instruction **try**...**finally**.

L'exemple suivant illustre comment du code qui ouvre et traite un fichier peut garantir que le fichier est fermé, même s'il y a une erreur à l'exécution.

```
Reset(F);
try
  ... // traiter le fichier F
finally
  CloseFile(F);
end;
```

Une instruction **try**...**finally** a la syntaxe suivante :

try listeInstruction1 **finally** listeInstruction2 **end**

où chaque listeInstruction est une suite d'instructions délimitées par des points-virgule. L'instruction **try**...**finally** exécute les instructions de listeInstruction1 (la clause **try**). Si listeInstruction1 se termine sans déclencher d'exception, listeInstruction2 (la clause **finally**) est exécutée. Si une exception est déclenchée lors de l'exécution de listeInstruction1, le

contrôle est transféré à listeInstruction2 ; quand listeInstruction2 a fini de s'exécuter, l'exception est redéclenchée. Si un appel des procédures Exit, Break ou Continue force la sortie de listeInstruction1, listeInstruction2 est exécutée automatiquement. Ainsi, la clause **finally** est toujours exécutée quelle que soit la manière dont se termine l'exécution de la clause **try**.
Si une exception est déclenchée sans être gérée par la clause **finally**, cette exception se propage hors de l'instruction **try**...**finally** et toute exception déjà déclenchée dans la clause **try** est perdue. La clause **finally** doit donc gérer toutes les exceptions déclenchées localement afin de ne pas perturber la propagation des autres exceptions.
Classes et routines standard des exceptions
L'unité SysUtils déclare plusieurs routines standard de gestion d'exceptions, dont ExceptObject, ExceptAddr et ShowException. SysUtils et d'autres unités de la VCL contiennent également de nombreuses classes d'exceptions qui dérivent toutes (sauf OutlineError) de Exception.
La classe Exception contient les propriétés **Message** et HelpContext qui peuvent être utilisées pour transmettre une description de l'erreur et un identificateur de contexte pour une aide contextuelle. Elle définit également divers constructeurs qui permettent de spécifier la description et l'identificateur de contexte de différentes manières. Pour davantage d'informations, voir l'aide en ligne.

J. L'instruction with

Une instruction **with** est un raccourci permettant de référencer les champs d'un enregistrement ou les propriétés et méthodes d'un objet. L'instruction **with** a la syntaxe suivante
with obj **do** instruction
ou
with obj1, ..., objn **do** instruction
où obj est une référence de variable désignant un objet ou un enregistrement et instruction est une instruction simple ou structurée. A l'intérieur de instruction, vous pouvez faire référence aux champs, propriétés et méthodes de obj en utilisant seulement leur identificateur, sans utiliser de qualificateur.
Par exemple, étant donné les déclarations suivantes :

```
type TDate = record
  Jour: Integer;
  Mois: Integer;
  Annee: Integer;
end;
var DateCommande: TDate;
```

Vous pouvez écrire l'instruction **with** suivante :

```
with DateCommande do
  if Mois = 12 then   begin
    Mois := 1;
    Annee := Annee + 1;
  End   else
    Mois := Mois + 1;
```

Qui est équivalente à

```
if DateCommande.Mois = 12 then begin
  DateCommande.Mois := 1;
  DateCommande.Annee := DateCommande.Annee + 1;
end
else
  DateCommande.Mois := DateCommande.Mois + 1;
```

Si l'interprétation de obj suppose des indices de tableau ou le déréférencement de pointeurs, ces actions ne sont effectuées qu'une seule fois, avant l'exécution de l'instruction. Cela rend les instructions **with** aussi efficaces que concises. Mais cela signifie également que les affectations d'une variable à l'intérieur de l'instruction ne peuvent changer l'interprétation de obj pendant l'exécution en cours de l'instruction **with**.
Chaque référence de variable ou nom de méthode d'une instruction **with** est interprété, si c'est possible, comme un membre de l'objet ou de l'enregistrement spécifié. Pour désigner une autre variable ou méthode portant le même nom que celui auquel vous accédez avec l'instruction **with**, vous devez le préfixer avec un qualificateur comme dans l'exemple suivant :

```
with DateCommande do  begin
    Annee := Unit1.Annee
    ...
  end;
```

Quand plusieurs objets ou enregistrements apparaissent après le mot réservé **with**, l'instruction est traité comme une série d'instructions **with** imbriquée. Ainsi

```
with obj1, obj2, ..., objn do instruction
```

est équivalent à

```
with obj1 do
  with obj2 do
    ...
    with objn do
      instruction
```

Dans ce cas, chaque référence de variable ou nom de méthode de instruction est interprété, si c'est possible, comme un membre de objn ; sinon, il est interprété, si c'est possible, comme un membre de objn−1; et ainsi de suite. La même règle s'applique pour l'interprétation même des objs : si objn est un membre de obj1 et de obj2, il est interprété comme obj2.objn.

K. Amélioration de la lisibilité

Afin de les rendre plus lisible, on s'inspirera des programmes ADA en terminant les structures par un commentaire de fin de structure :

```
Function toto ...
end {toto} ;
```

```
if ... begin
...
end{if};
```

```
case ...
end{case};
```

```
With ...
end{with};
```

```
while ...
end{while};
```

```
for k...
end{for k};
```

```
record toto ...
end{record}; // ou
end{toto};
```

L. Blocs et portée

Les déclarations et les instructions sont organisées en blocs qui définissent des noms de domaine locaux (ou portées) pour les labels et les identificateurs. Les blocs permettent à un même identificateur, par exemple un nom de variable, d'avoir des significations différentes dans différentes parties d'un programme. Chaque bloc fait partie de la déclaration d'un programme, d'une fonction ou d'une procédure ; la déclaration de chaque programme, fonction ou procédure est composée d'un seul bloc.

1. Blocs

Un bloc est composé d'une série de déclarations suivies d'une instruction composée. Toutes les déclarations doivent se trouver rassemblées au début du bloc. Un bloc a donc la forme suivante :

déclarations
begin
  instructions
end

La section déclarations peut contenir, dans un ordre quelconque, des déclarations de variables, de constantes (y compris des chaînes de ressource), de types, de procédures, de fonctions et de labels. Dans un bloc de programme, la section déclarations peut également contenir une ou plusieurs clauses exports (voir Bibliothèques de liaison dynamique et paquets).
Par exemple, dans la déclaration de fonction suivante :

```
function Majuscule (const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ...
end;
```

La première ligne de la déclaration est l'en-tête de fonction et toutes les lignes suivantes constituent le bloc de la fonction. Ch, L, Source et Dest sont des variables locales ; leur déclaration n'est valable que dans le bloc de la fonction Majuscule et redéfinit (uniquement dans ce bloc) toute déclaration des mêmes identificateurs faite dans le bloc du programme ou dans les sections **interface** ou implémentation d'une unité.

2. Portée

Un identificateur, par exemple une variable ou un nom de fonction, ne peut être utilisé qu'à l'intérieur de la portée de sa déclaration. L'emplacement d'une déclaration détermine sa portée. La portée d'un identificateur déclaré dans la déclaration d'un programme, d'une fonction ou d'une procédure est limitée au bloc dans lequel il a été déclaré. Un identificateur déclaré dans la section **interface** d'une unité a une portée qui inclut toutes les autres unités et programmes utilisant l'unité où cette déclaration est faite. Les identificateurs ayant une portée plus restreinte (en particulier les identificateurs déclarés dans les fonctions et procédures) sont parfois dits locaux alors que les identificateurs ayant une portée plus étendue sont appelée globaux.

Les règles déterminant la portée d'un identificateur sont résumées ci-dessous :

Si l'identificateur est déclaré dans ...	Sa portée s'étend ...
La déclaration d'un programme, d'une fonction ou d'une procédure.	Depuis le point où il a été déclaré jusqu'à la fin du bloc en cours, y compris tous les blocs inclus dans cette portée.
La section **interface** d'une unité.	Depuis le point où il a été déclaré jusqu'à la fin de l'unité et dans toutes les unités ou programmes utilisant cette unité. Voir Programmes et unités.)
La section implémentation d'une unité mais hors du bloc d'une fonction ou d'une procédure.	Depuis le point où il a été déclaré jusqu'à la fin de la section implémentation. L'identificateur est disponible dans toutes les fonctions et procédures de la section implémentation.
La définition d'un type enregistrement (c'est-à-dire que l'identificateur est le nom d'un champ de l'enregistrement).	Depuis le point où il a été déclaré jusqu'à la fin de la définition du type de champ. Voir Enregistrements.)
La définition d'une classe (c'est-à-dire que l'identificateur est le nom d'une propriété ou d'une méthode de la classe).	Depuis le point où il a été déclaré jusqu'à la fin de la définition du type classe et également dans les définitions des descendants de la classe et les blocs de toutes les méthodes de la classe et de ses descendants. Voir Classes et objets.

3. Conflits de nom

Quand un bloc en comprend un autre, le premier est appelé bloc extérieur et l'autre est appelé bloc intérieur. Si un identificateur déclaré dans le bloc extérieur est redéclaré dans le bloc intérieur, la déclaration intérieure redéfinit l'extérieure et détermine la signification de l'identificateur pour la durée du bloc intérieur. Si, par exemple, vous avez déclaré une variable appelée ValeurMax dans la section **interface** d'une unité, puis si vous déclarez une autre variable de même nom dans une déclaration de fonction de cette unité, toute occurrence non qualifiée de ValeurMax dans le bloc de la fonction est régit par la deuxième définition, celle qui est locale. De même, une fonction déclarée à l'intérieur d'une autre fonction crée une nouvelle portée interne dans laquelle les identificateurs utilisés par la fonction externe peuvent être localement redéfinis.

L'utilisation de plusieurs unités complique davantage la définition de portée. Chaque unité énumérée dans une clause **uses** impose une nouvelle portée qui inclut les unités restantes utilisées et le programme ou l'unité contenant la clause **uses**. La première unité d'une clause **uses** représente la portée la plus externe, et chaque unité successive représente une nouvelle portée interne à la précédente. Si plusieurs unités déclarent le même identificateur dans leur section **interface**, une référence sans qualificateur à l'identificateur sélectionne la déclaration effectuée dans la portée la plus externe, c'est-à-dire dans l'unité où la référence est faite, ou, si cette unité ne déclare pas l'identificateur dans la dernière unité de la clause **uses** qui déclare cet identificateur.

L'unité System est utilisée automatiquement par chaque programme et unité. Les déclarations de System ainsi que les types prédéfinis, les routines et les constantes reconnues automatiquement par le compilateur ont toujours la portée la plus extérieure.
Vous pouvez redéfinir ces règles de portée et court-circuiter une déclaration intérieure en utilisant un identificateur qualifié (voir Identificateurs qualifiés) ou une instruction **with** (voir Instructions **with**).

4. Identificateurs qualifiés

Quand vous utilisez un identificateur qui a été déclaré à plusieurs endroits, il est parfois nécessaire de qualifier l'identificateur. La syntaxe d'un identificateur qualifié est :
identificateur1.identificateur2
où identificateur1 qualifie identificateur2. Si, par exemple, deux unités déclarent une variable appelée ValeurEnCours, vous pouvez désigner ValeurEnCours de Unit2 en écrivant :
Unit2.ValeurEncours
Il est possible de chaîner les qualificateurs. Par exemple :
Form1.Button1.Click
Appelle la méthode Click de Button1 dans Form1.
Si vous ne qualifiez pas un identificateur, son interprétation est déterminée par les règles de portée décrites dans Blocs et portée.

M. Exercice

Reprendre l'étude de l'algorithme du tri rapide page 48

VII. Surcharge des routines

1. définition

Il est possible de redéclarer plusieurs fois une routine dans la même portée sous le même nom. C'est ce que l'on appelle la redéfinition (ou surcharge). Les routines redéfinies doivent être redéclarées avec la directive **overload** et doivent utiliser une liste de paramètres différente. Soit, par exemple, les déclarations :

```
function Diviser(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;

function Diviser(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

Ces déclarations créent deux fonctions appelées toutes les deux Diviser qui attendent des paramètres de types différents. Quand vous appelez Diviser, le compilateur détermine la fonction à utiliser en examinant les paramètres effectivement transmis dans l'appel. Ainsi, Diviser(6.0, 3.0) appelle la première fonction Diviser car ses arguments sont des valeurs réelles.
Lorsqu'une routine est redéfinie, vous pouvez transmettre des paramètres qui ne sont pas de mêmes types que ceux des déclarations de la routine, mais qui sont compatibles pour l'affectation avec les paramètres de plus d'une déclaration. Par exemple, cela se produit très fréquement lorsqu'une routine est surchargée avec des types d'entiers différents ou des types de réels différents.

```
procedure Store(X: Longint); overload;

procedure Store(X: Shortint); overload;
```

Dans ces cas, lorsque cela est possible sans ambiguïté, le compilateur invoque la routine dont le type des paramètres a l'étendue la plus courte supportant les paramètres réels passés dans l'appel. (Souvenez-vous que les expressions constantes de valeur réelle sont toujours de type Extended.)
Les routines redéfinies doivent pouvoir se distinguer par le nombre ou le type de leurs paramètres. Ainsi, la paire de déclarations suivante déclenche une erreur de compilation :

```
function Maj(S: string): string; overload;
 ...
procedure Maj(var Str: string); overload;
 ...
```

Alors que les déclarations :

```
function Fonc(X: Real; Y: Integer): Real; overload;
 ...
function Fonc(X: Integer; Y: Real): Real; overload;
```

...
sont légales.
Quand une routine redéfinie est déclarée dans une déclaration **forward** ou d'**interface**, la déclaration de définition doit obligatoirement répéter la liste des paramètres de la routine.
Si vous utilisez des paramètres par défaut dans des routines redéfinies, méfiez-vous des signatures de paramètres ambiguës. Pour davantage d'informations, voir Paramètres par défaut et routines redéfinies.
Vous pouvez limiter les effets potentiels de la redéfinition en qualifiant le nom d'une routine lors de son appel. Par exemple, Unit1.MaProcedure(X, Y) n'appelle que les routines déclarées dans Unit1 ; si aucune routine de Unit1 ne correspond au nom et à la liste des paramètres, il y a une erreur de compilation.
Pour des informations sur la distribution de méthodes redéfinies dans une hiérarchie de classes, voir Redéfinition de méthodes. Pour plus d'informations sur l'exportation depuis une DLL de routines redéfinies, voir La clause exports de l'aide Delphi.

2. exemple

```
//Surcharge de la procédure val de l'unité system
function val(chn:string):real ; overload;
var err:integer ;
 begin
   system.val(chn,result,err);  // récupération  de la
         // procédure d'origine. Val seulement ferait
         //référence (récursive) à la fonction
 end;
```

VIII. Exemples récapitulatifs

Ces 2 exemples traitent une liste de chaînes de caractères

A. exemple 1

Ce premier exemple utilise une variable globale (la liste)

```
program strlst1;
{$APPTYPE CONSOLE}

uses  dialogs;

Const
 nl=#10#13;
 max=100;
var
         nb:word;
         item:array[1..max] of string;

procedure clear;
Begin
  nb := 0
End;

procedure add (chn:string);
Begin
inc(nb);
   item[nb]:=chn;
End;

procedure delete (num:word);
var i :word;
Begin
for i:= num to nb do
     item[i]:=item[i+1];
   dec(nb);
End;

procedure insert (num:word;chn:string);
var i :word;
Begin
  inc(nb);
  for i:= nb downto num do
    item[i]:=item[i-1];
  item[num]:= chn;
End;

procedure exchange (pos1,pos2:word);
var tmp:string;
Begin
   tmp :=  item[pos1];
   item[pos1]:= item[pos2];
   item[pos2]:= tmp;
End;
```

```
function getItem(Const Index: Integer): string;
Begin
   result := item[Index];
End ;

function toutTexte :  string;
var indx : word;
Begin
  result:='';
  for indx:=1 to nb do
    result := result+nl+item[indx];
End ;

function Find(const S: string; var Index: Integer):
                                          Boolean;

{ Fonction qui recherche une chaîne S dans le
tableau T la fonction renvoie  -  Vrai si S est
dans T et la paramètre résultat Index contient
l'adresse  -  Faux sinon et index contient
n'importe quoi}
var  i : word ;
Begin
// A écrire ...
End ;

procedure sort() ;  // ou procedure sort;
{Trie les chaînes du tableau en ordre croissant}
Begin
// A écrire ... revoir le Qsort page 48
End;

BEGIN
   clear();  // ou plus simplement clear ;
   Add('première ligne');
   Add('deuxième ligne');
   Add('toisième ligne');
   ShowMessage('utilisation de la fonction :'+nl+nl+toutTexte);
   writeln( 'utilisation du champ : chaîne d''indice 1:'+nl+nl+item[1]);
   writeln;
   writeln('remarquer les lettres accentuées en mode console et fenêtré');
   ShowMessage('utilisation de la fonction :chaîne d''indice 1'+nl+nl+GetItem(1));
   insert(3,'ligne 2 bis',maListe);
   ShowMessage('utilisation de la fonction :'+nl+nl+toutTexte());
END.
```

B. exemple 2

Ce second exemple est une modification du précédent utilisant un **record** et un passage de paramètres pae adresse :

```
program strlst2;
{$APPTYPE CONSOLE}

uses  dialogs;

Const
 nl=#10#13;
 max=100;
type TtabStr =record
          nb:word;
          item:array[1..max] of string;
           end{record};

procedure clear (var t: TtabStr);
Begin
  with t do nb:=0  // ou plus simplement  t.nb :=
0
End;

procedure add (chn:string;var t: TtabStr);
Begin
  with t do begin  // On pourrait ne pas utiliser
with et
    inc(nb);        // qualifier chaque champ :
inc(t.nb);
    item[nb]:=chn;//  t.item[t.nb] := chn
  end;{with}
End{add};

procedure delete (num:word;var t: TtabStr);
var i :word;
Begin
  with t do begin
    for i:= num to nb do
      item[i]:=item[i+1];
    dec(nb);
  end{with};
End;{delete}

procedure insert (num:word;chn:string;var t:
                                        TtabStr);
var i :word;
```

```
Begin
  with t do begin
    inc(nb);
    for i:= nb downto num do
      item[i]:=item[i-1];
    item[num]:= chn;
  end{with};
End;{insert}

procedure exchange (pos1,pos2:word;var :
                                    TtabStr);
var tmp:string;
Begin
  with t do begin
    tmp :=  item[pos1];
    item[pos1]:= item[pos2];
    item[pos2]:= tmp;
  end{with};
End;{exchange}

function getItem(Const Index: Integer; Const t:
TtabStr): string;
Begin
  with t do result := item[Index]
// ou plus simplement : result := t.item[Index];
End {getItem};

 var maListe:  TtabStr;

BEGIN { strlst2}
   clear(maListe);
   Add('première ligne',maListe);
   Add('deuxième ligne',maListe);
   Add('toisième ligne',maListe);
   ShowMessage('utilisation de la fonction :'+nl+nl+toutTexte(maListe));
   writeln( 'utilisation du champ : chaîne d''indice 1:'+nl+nl+maListe.item[1]);
   writeln;
   writeln('remarquer les lettres accentuées en mode console et fenêtré');
   ShowMessage('utilisation de la fonction :chaîne d''indice 1'+nl+nl+GetItem(1,maListe));
   insert(3,'ligne 2 bis',maListe);
   ShowMessage('utilisation de la fonction :'+nl+nl+toutTexte(maListe));
END.
```

```
function toutTexte( Const t: TtabStr): string;
var indx : word;
Begin
  result:='';
  with t do
  for indx:=1 to nb do
    result := result+nl+item[indx];
End{toutTexte} ;

function Find(const S: string; Const T: TtabStr;
var Index: Integer): Boolean;
{ Fonction qui recherche une chaîne S dans le
tableau T la fonction renvoie Vrai si S est dans T
et la paramètre résultat Index contient
l'adresse Faux sinon et index contient n'importe
quoi}
var  i : word ;
Begin
// A écrire ...
End ;

procedure sort (var t: TtabStr);
{Trie les chaines du tableau en ordre croissant}
Begin
// A écrire ... revoir le Qsort page 48
End;
```

IX. Structure de données orientée objet

A. Terminologie

Une classe (un type classe) définit une structure composée de champs, de méthodes et de propriétés.

Les instances d'un type classe sont appelées des objets.

Les champs, méthodes et propriétés d'une classe sont appelés ses composants ou ses membres.

Un champ est essentiellement une variable faisant partie d'un objet. Comme les champs d'un enregistrement, un champ de classe représente des éléments de données qui existent dans chaque instance de la classe.

Une méthode est une procédure ou une fonction associée à la classe. La plupart des méthodes portent sur des objets, c'est-à-dire sur des instances d'une classe. Certaines méthodes, appelées méthodes de classe, portent sur les types classe même.

Une propriété est une interface avec les données associées à un objet (souvent stockées dans un champ). Les propriétés ont des spécificateurs d'accès qui déterminent comment leurs données sont lues et modifiées. Pour le reste d'un programme (hors de l'objet même), une propriété apparaît à bien des points de vue comme un champ.

Les objets sont des blocs de mémoire alloués dynamiquement dont la structure est déterminée par leur type de classe. Chaque objet détient une copie unique de chaque champ défini dans la classe. Par contre, toutes les instances d'une classe partagent les mêmes méthodes. Les objets sont créés et détruits par des méthodes spéciales appelées constructeurs et destructeurs.

Une variable de type classe est en fait un pointeur qui référence un objet. Plusieurs variables peuvent donc désigner le même objet. Comme les autres pointeurs, les variables de type classe peuvent contenir la valeur nil. Cependant, il n'est pas nécessaire de déréférencer explicitement une variable de type classe pour accéder à l'objet qu'elle désigne. Par exemple, UnObjet.Taille := 100 affecte la valeur 100 à la propriété Taille de l'objet référencé, UnObjet ; vous ne devez pas l'écrire sous la forme UnObjet^.Taille := 100.

Un type classe doit être déclaré et nommé avant de pouvoir être instancié. Il n'est donc pas possible de définir un type classe dans une déclaration de variable.

Déclarez les classes uniquement dans la portée la plus large d'un programme ou d'une unité, mais pas dans une déclaration de procédure ou de fonction.

La déclaration d'un type classe a la forme suivante :

type nomClasse = **class** (classeAncêtre)

listeMembre

end;

où nomClasse est un identificateur valide, (classeAncêtre) est facultatif et listeMembre déclare les membres (les champs, méthodes et propriétés) de la classe. Si vous omettez (classeAncêtre), la nouvelle classe hérite directement de la classe prédéfinie TObject. Si vous précisez (classeAncêtre) en laissant vide listeMembre, vous pouvez omettre le **end** final. Une déclaration de type classe peut également contenir une liste des interfaces implémentées par la classe ; voir Implémentation des interfaces.

Les méthodes apparaissent dans une déclaration de classe sous la forme d'en-tête de fonction ou de procédure sans le corps. La déclaration de définition de chaque méthode est faite ailleurs dans le programme.

Quand vous déclarez une classe, vous pouvez spécifier son ancêtre immédiat. Par exemple :

type TUnControle = **class**(TWinControl);

déclare une classe appelée TUnControle qui descend (dérive) de TWinControl. Un type classe hérite automatiquement de tous les membres de son ancêtre immédiat. Chaque classe peut déclarer de nouveaux membres et redéfinir les membres hérités. Par contre, une classe ne peut supprimer des membres définis dans son ancêtre. Ainsi TUnControle contient tous les membres définis dans TWinControl et dans chacun des ancêtres de TWinControl.

La portée de l'identificateur d'un membre commence à l'endroit où le membre est déclaré et se poursuit jusqu'à la fin de la déclaration de la classe et s'étend à tous les descendants de la classe et les blocs de toutes les méthodes définies dans la classe et ses descendants.

B. TObject et TClass

La classe TObject, déclarée dans l'unité System, est l'ancêtre ultime de toutes les autres classes. TObject définit seulement quelques méthodes, dont un constructeur et un destructeur de base. Outre la classe TObject, l'unité System déclare le type référence de classe TClass :

TClass = **class of** TObject;

Quand la déclaration d'un type classe ne spécifie pas d'ancêtre, la classe hérite directement de TObject. Donc :

```
type TMaClasse = class
 ...
end;
```

est équivalent à :

```
type TMaClasse = class(TObject)
...
end;
```

C. Exemple

Reprenons l'exemple page 61 des listes de chaînes utilisant un **record** et changeons ce mot en **class** (les **seuls** changements apparaissent en gras):

```
program strlst3;
{$APPTYPE CONSOLE}

uses  dialogs;

Const
 nl=#10#13;
 max=100;
type TtabStr =class
          nb:word;
          item:array[1..max] of string;
          end{class};
// Idem à strlst2

BEGIN
   maListe:= TtabStr.create; // instancions l'objet
   clear(maListe);
// Idem à strlst2
   maListe.destroy  //  libérons la mémoire
END.
```

Remarques

Dans l'exemple précédent on utilise le constructeur create des Tobject

create est toujours qualifié par une classe et non par l'objet alors que destroy est qualifié par l'objet

D. Compatibilité des types classe

Un type classe est compatible pour l'affectation avec ses ancêtres. Une variable d'un type classe peut donc référencer une instance de tout type descendant. Par exemple, étant donné la déclaration :

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TCarre = class(TRectangle);
var
  Fig: TFigure;
```

il est possible d'affecter à la variable Fig des valeurs de type TFigure, TRectangle TCarre.

E. Visibilité des membres de classes

Chaque membre d'une classe a un attribut appelé visibilité, indiqué par l'un des mots réservés suivants : **private**, **protected**, **public**, **published** ou **automated**. Par exemple :

published property Couleur: TColor **read** LitCouleur **write** EcritCouleur;

déclare une propriété publiée appelée Couleur. La visibilité détermine où et comment il est possible d'accéder à un membre :

private (privée) représente l'accès minimum, ==> Ne sont accessibles que par les instances de la classe elle-même.

protected (protégée) représente un niveau intermédiaire d'accès, ==> Sont accessibles par les instances de la classe elle-même et par les instances des classes dérivées de la classe en question.

public (publique), **published** (publiée) ==> Sont accessibles par toutes les instances de les toutes classes

automated (automatisée) représentant l'accès le plus large. ==> Sont accessibles par toutes les instances de les toutes classes et par l'inspecteur d'objet.

Si la déclaration d'un membre ne spécifie pas sa visibilité, le membre a la même visibilité que celui qui le précède. Les membres au début de la déclaration d'une classe dont la visibilité n'est pas spécifiée sont par défaut publiés si la classe a été compilée dans l'état *{$M+}* ou si elle dérive d'une classe compilée à l'état *{$M+}* ; sinon ces membres sont publics.

Dans un souci de lisibilité, il est préférable d'organiser la déclaration d'une classe en fonction de la visibilité, en plaçant tous les membres privés ensemble, suivis de tous les membres protégés, etc. De cette manière, chaque mot réservé spécifiant la visibilité apparaît au maximum une fois et marque le début d'une nouvelle "section" de la déclaration. Une déclaration de classe standard doit donc avoir la forme :

```
type
  TMaClasse = class(TControl)
  private
   ... { déclarations privées}
  protected
   ... { déclarations protégées }
  public
   ... { déclarations publiques }
  published
   ... { déclarations publiées }
  end;
```

Vous pouvez augmenter la visibilité d'un membre dans une classe dérivée en le redéclarant, mais il n'est pas possible de réduire sa visibilité. Par exemple, une propriété protégée peut être rendue publique dans un descendant mais pas privée. De plus, les membres publiés ne peuvent devenir publics dans une classe dérivée. Pour davantage d'informations, voir Surcharge et redéclaration de propriétés.

F. Constructeurs et destructeurs

Comme pour les objets, la mémoire est allouée dynamiquement. Pour les pointeurs, on utilise l'instruction new ou getmem lorsque l'on veut allouer de la mémoire et dispose ou freemem pour la libérer. Pour les objets, on utilise des méthodes spéciales que l'on appelle constructeur (que l'on nomme généralement create) et destructeur (que l'on nomme généralement destroy). Et au lieu de ***procedure*** et ***function*** comme pour les autres méthodes, on utilise ***constructor*** et ***destructor***.

G. Exemple

Revenons sur l'exemple de la liste de chaînes (page 61) avec une version objet

```
program strlst5;   {$APPTYPE CONSOLE}

uses    dialogs;

Const  nl=#10#13;
 max=100;
type TtabStr =class
            nb:word;
            item:array[1..max] of string;
            constructor Create ;
            procedure clear ;
            procedure add (chn:string);
            procedure delete (num:word);
            procedure insert
(num:word;chn:string);
            procedure exchange
(pos1,pos2:word);
            function getItem(Const Index:
Integer): string;
            function toutTexte : string;
      function Find(const S: string;  var Index:
Integer): Boolean;
            procedure sort (); // les () sont
facultatives comme pour
          end;             // create et clear

constructor TtabStr.Create () ;
Begin
  Inherited Create;
  clear;
End;

procedure TtabStr.clear;
Begin
  nb:=0
End;

procedure TtabStr.add (chn:string);
Begin
  inc(nb);
  item[nb]:=chn;
```

```
End;

procedure TtabStr.delete (num:word);
var i :word;
Begin
  for i:= num to nb do
    item[i]:=item[i+1];
  dec(nb);
End;

procedure TtabStr.insert (num:word;chn:string);
var i :word;
Begin
  inc(nb);
  for i:= nb downto num do
    item[i]:=item[i-1];
  item[num]:= chn;
End;

procedure TtabStr.exchange (pos1,pos2:word);
var tmp:string;
Begin
  tmp :=  item[pos1];
  item[pos1]:= item[pos2];
  item[pos2]:= tmp;
End;

function TtabStr.getItem(Const Index: Integer):
string;
Begin
  result := item[Index];
End ;

function TtabStr.toutTexte: string;
var indx : word;
Begin
  result:='';
  for indx:=1 to nb do
    result := result+nl+item[indx];
End ;

function TtabStr.Find(const S: string;  var Index:
Integer): Boolean;
{ Fonction qui recherche une chaîne S dans le
tableau T la fonction renvoie
  - Vrai si S est dans T et le paramètre résultat
Index contient l'adresse
  - Faux sinon et index contient n'importe quoi}
var  i : word ;
Begin
// A écrire ...
End ;

procedure TtabStr.sort ();
{Trie les chaînes du tableau en ordre croissant}
Begin
// A écrire ... revoir le Qsort page 48
End;
          var maListe:  TtabStr;
BEGIN
concat;
   maListe:= TtabStr.create; //Maintenant le
constructeur initialise aussi nb. La 2ème ligne
clear est donc inutile.
   maListe.Add('première ligne');
   maListe.Add('deuxième ligne');
   maListe.Add('toisième ligne');
   ShowMessage('utilisation de la fonction
:'+nl+nl+maListe.toutTexte);
   writeln( 'utilisation du champ : chaîne
d''indice 1:'+nl+nl+maListe.item[1]);
   writeln;
   writeln('remarquer les lettres accentuées en
mode console et fenêtré');
   ShowMessage('utilisation de la fonction
:chaîne d''indice 1'+nl+nl+maListe.GetItem(1));
   maListe.insert(3,'ligne 2 bis');
   ShowMessage('utilisation de la fonction
:'+nl+nl+maListe.toutTexte);
   maListe.exchange(1,3);
   ShowMessage('aprés utilisation de echange
:'+nl+nl+maListe.toutTexte);
   maListe.delete(2);
   ShowMessage('après utilisation de delete
:'+nl+nl+maListe.toutTexte);
   maListe.destroy
END.
```

H. Exemple d'utilisation des objets prédéfinis de delphi

1. Exemple 1

TStringList est une implémentation (plus efficaces) des listes de chaînes de caractères en delphi .

(voir en particulier TStringList.CustomSort et find dans l'aide delphi)

```
program str_list;
{$APPTYPE CONSOLE}

uses  classes,dialogs;

Const nl=#10#13;
 var maListe:  TStringList;

begin
```

```
  maListe:=TStringList.create;
  maListe.Add('première ligne');
  maListe.Add('deuxième ligne');
  maListe.Add('toisième ligne');
  writeln( 'utilisation de la
propriété :'+nl+nl+maListe.Text);
  ShowMessage('utilisation de la
propriété :'+nl+nl+maListe.Text);
  ShowMessage('utilisation de la
fonction :'+nl+nl+maListe.GetText);
  ShowMessage('chaîne d''indice 0 :
'+nl+nl+maListe.Strings[0]);
  maListe.Text := 'tout le texte est changé';
  ShowMessage('utilisation de la propriété
:'+nl+nl+maListe.Text);
  ShowMessage('chaîne d''indice 0 :
'+nl+nl+maListe.Strings[0]);
end.
```

2. Exemple 2

```
program tst_form;

uses
 Forms,stdctrls;

var
 maForm:tform;
 MonBouton : TButton;


begin
 Application.Initialize;
 Application.CreateForm(TForm, maForm);
 maForm.visible := true;
 maForm.top:=30;
 maForm.left:=100;
 maForm.height:=300;
 maForm.width:=400;
 maForm.caption := 'nouvelle fiche';
 MonBouton := TButton.Create(maForm);
 with MonBouton do begin
  Parent := maForm;
  top := 20;
  left := 20;
  height := 20;
  width := 200;
  caption := 'nouveau bouton';
 end;
 Application.Run;
end.
```

Remarques et commentaires :

Il ne s'agit plus ici d'une application console, mais un simple éditeur et la compilation avec DCC32 suffisent.

Cet exemple est donné dans un but pédagogique et ne correspond pas à la façon dont on développe des applications avec Delphi

Application est une variable de **type *Tapplication*** définie automatiquement par Delphi. Les 3 lignes :

Application.Initialize; Application.CreateForm(TForm, maForm); et Application.Run; seront automatiquement générée par l'EDI et feront partie du programme principal (fichier .DPR) menu :Projet/Voir le source

Tapplication et Tform sont définis dans l'unité Forms et Tbutton dans l'unité stdctrls

Nous aurions pu mettre ***maForm*** « en facteur » avec **with** comme c'est fait pour ***monBouton***

Les propriétés top, left, height, width, caption sont en général définies dans l'inspecteur d'objet (voir page 72) et non dans le code du programme : on peut leur donner une valeur numérique au clavier ou déplacer visuellement les objets

Les coordonnées top et left du coin supérieur gauche sont données à partir du coin supérieur gauche de l'écran, celles du bouton à partir du coin supérieur gauche de la fiche (le propriéraire ***TButton.Create(maForm)*** et parent ***Parent := maForm***)

3. Exemple 3

Reprenons l'exemple 2 sans déclarer de variable *MonBouton* :

```
program tst_form;

uses
 Forms,stdctrls;

var
 maForm:tform;


begin
 Application.Initialize;
 Application.CreateForm(TForm, maForm);
 maForm.visible := true;
 maForm.top:=30;
 maForm.left:=100;
 maForm.height:=300;
```

```
  maForm.width:=400;
  maForm.caption := 'nouvelle fiche';
  with TButton.Create(maForm) do begin
    Parent := maForm;
    top := 20;
    left := 20;
    height := 20;
    width := 200;
    caption := 'nouveau bouton';
  end;
  Application.Run;
end.
```

4. Exemple 4

Cela se complique, si l'on veut faire de même avec la form : en effet, l'objet application gère automatiquement la création et la destruction de la forme ainsi que de tous les objets créés par l'application

```
program tst_form;
{$APPTYPE CONSOLE}

uses
  Forms,stdctrls;

begin
  with Tform.create(nil) do begin
    visible := true;
    top:=30;
    left:=100;
    height:=300;
    width:=400;
    caption := 'nouvelle fiche';
  end;
  readln;
end.
```

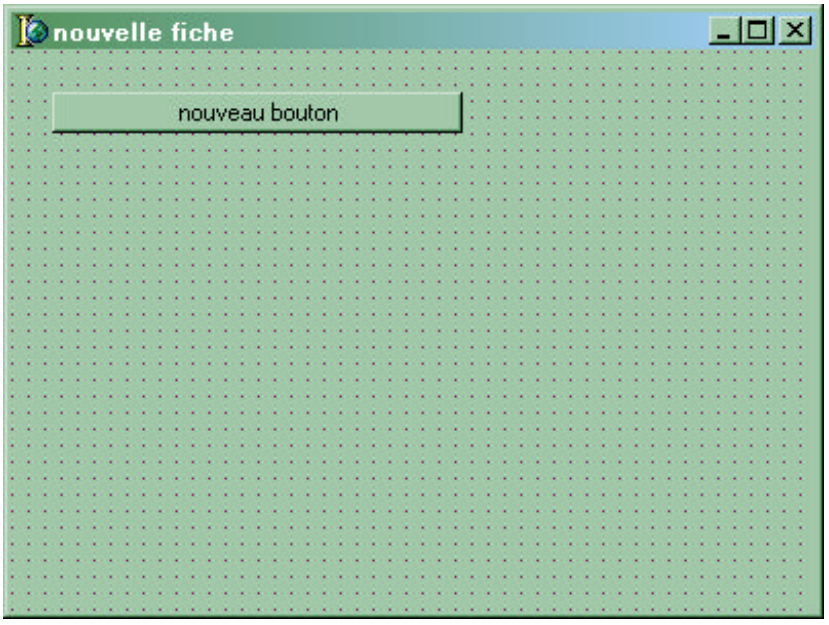
Remarquons ici qu'il s'agit d'une application console et la présence du *readln*.

Sans ce ***Readln***, l'absence de ***Application.Initialize; Application.CreateForm(TForm, maForm); Application.Run***; terminerait le programme dès la fin de la création de la fiche et donc on ne verrait rien.

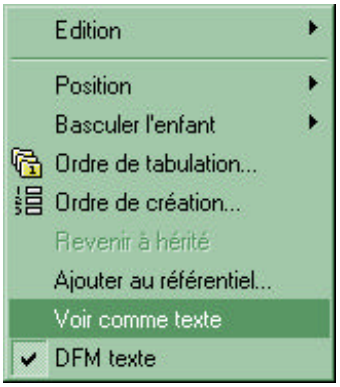
L'absence de nom à la fiche (pas de déclaration de variable) empêche d'y placer un élément visuel par exemple un bouton comme précédemment, faute de pouvoir nommer propriétaire er parent.

I. Constitution des fichiers DPR, DMF, PAS

Créons un nouveau projet. Nommons la fiche « nouvelle fiche » plaçons y un bouton que l'on nommera « nouveau bouton », nous obtenons quelquechose comme ceci :



Un click droit sur cette fiche fait apparaître :



Choisissons ***voir comme du texte,*** nos observons le texte suivant :

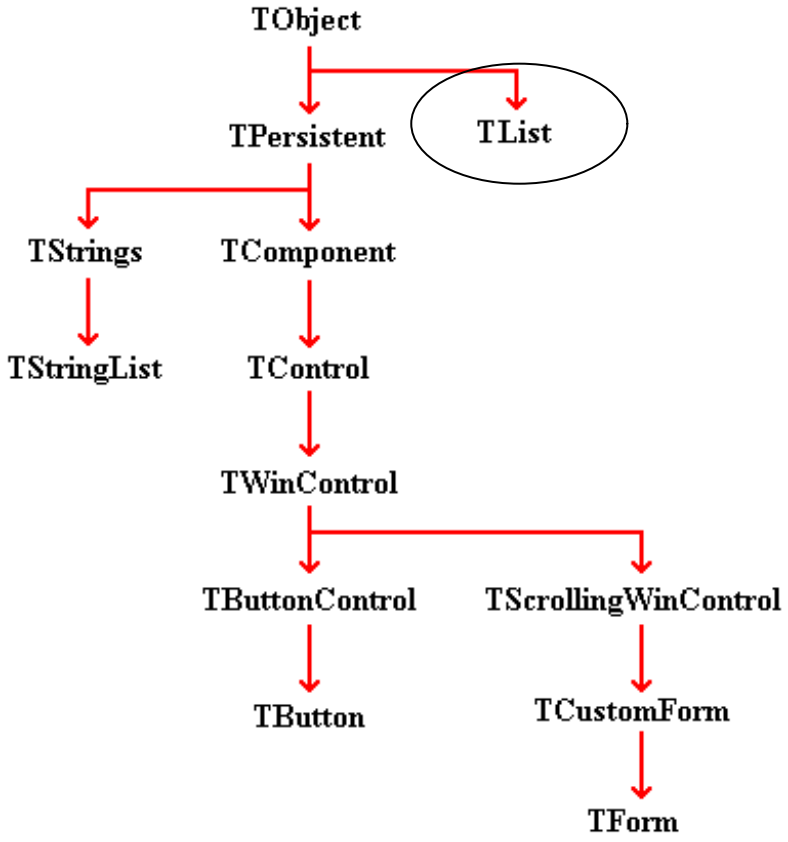
```
object Form1: TForm1
  Left = 100
  Top = 30
  Width = 400
  Height = 300
  Caption = 'nouvelle fiche'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 20
    Top = 20
    Width = 200
    Height = 20
    Caption = 'nouveau bouton'
    TabOrder = 0
  end
end
```

Si nous comparons à l'exemple précédent, nous remarquons une hiérarchie comparable à l'exemple 2 ci-dessus.
Bien entendu, l'absence de ***points-virgules*** et de **:=** montre qu'il ne s'agit pas de code delphi, mais de description des objets visuels : cest ce que contient le fichier ***.DFM*** (Delphi ForM)

X. Hiérarchie des classes, héritage et surcharge

A. Définition

Nous avons déjà parlé des classes en expliquant qu'elles 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.
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 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-descendant. Chaque classe possède 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. **On** dit dans ce cas que l'**on dérive** la classe parente.

Une telle hiérarchie impose l'existence d'un unique ancêtre 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.
Voici (une partie de) l'arbre représentant la hiérarchie des classes Delphi:

Dans l'exemple des TtabStr, nous avons ajouté un descendant à Tobject (et donc un « frère » à Tpersistent). TtabStr a donc immédiatement hérité des membres de la classe Tobject : nous avons d'ailleurs dans un premier temps utilisé le constructeur create des Tobject avant de le **surcharger** pour obtenir un constructeur plus adapté.

Nous remarquerons que TStringsList est un descendant de Tstrings et non de TList

B. Exemple

Dans cet exemple, nous allons dériver un Tlist pour réaliser une liste de réels:

```
unit U_ListReal;
(* ----------------------------------------------------
   Implémentation (partielle) d'une liste de
réels
   Sur le modèle des TstringList
  ----------------------------------------------------*)
interface
uses classes;

type
 tmaliste =class(Tlist)
    constructor Create ;
    destructor Destroy;
    function Add(Item: real): Integer;
    function getitem(Index:Integer):real ;        //
devrait être dans la zone private
    procedure setitem(Index:Integer;Item: real);
// devrait être dans la zone private
  private
      { Déclarations privées }
  protected
   { Déclarations protégées }
   public
   { Déclarations publiques }
  property Items[Index: Integer]:real read
getitem write setitem;

  published
  { Déclarations publiées }
  end;

implementation

constructor tmaliste.create;
begin
  Inherited Create;
end;

destructor tmaliste.Destroy;
begin
  Inherited Destroy;
end;

function tmaliste.Add(Item: real): Integer;
var p :^real;
begin
   new(p);
   p^:=Item ;
   Result:= inherited add(p);
end;

function tmaliste.getitem(index:Integer):real ;
begin
  result := real(inherited Items[index]^) ;
end;

procedure tmaliste.setitem(Index:Integer;Item:
real);
begin
  real(inherited Items[index]^) := item
end;

end.
```

C. Suite de l'exemple

Nous allons améliorer l'exemple en fournissant une procédure de tri de la liste des réels ; Tout est prévu dans Delphi:
Il suffit de définir une relation d'ordre par le biais d'une fonction : l'aide de Delphi indique pour ***tList.sort*** :

Trie la liste, en employant l'algorithme Tri Rapide (QuickSort) et en utilisant Compare comme fonction de comparaison.
type TListSortCompare = **function** (Item1, Item2: Pointer): Integer;
procedure Sort(Compare: TListSortCompare);
Description

La méthode Sort permet de trier les éléments du tableau Items. Compare est une fonction de comparaison indiquant comment les éléments sont ordonnés. Compare renvoie < 0 si Item1 est inférieur à Item2, 0 s'ils sont égaux et > 0 si Item1 est supérieur à Item2.

```
function cmpUp(Item1, Item2: Pointer): Integer;
begin
  if(real(Item1^))>(real(Item2^)) then
    result := 1
  else if (real(Item1^))=(real(Item2^)) then
    result := 0
  else
    result := -1
end;

function cmpDn(Item1, Item2: Pointer): Integer;
begin
  result := -cmpUp(Item1, Item2);
end;
procedure tmaliste.triUp;
begin
 sort(cmpUp)
end;

procedure tmaliste.triDn;
begin
 sort(cmpDn)
end;
```

D. Méthodes statiques, virtuelles et dynamiques ou abstraites

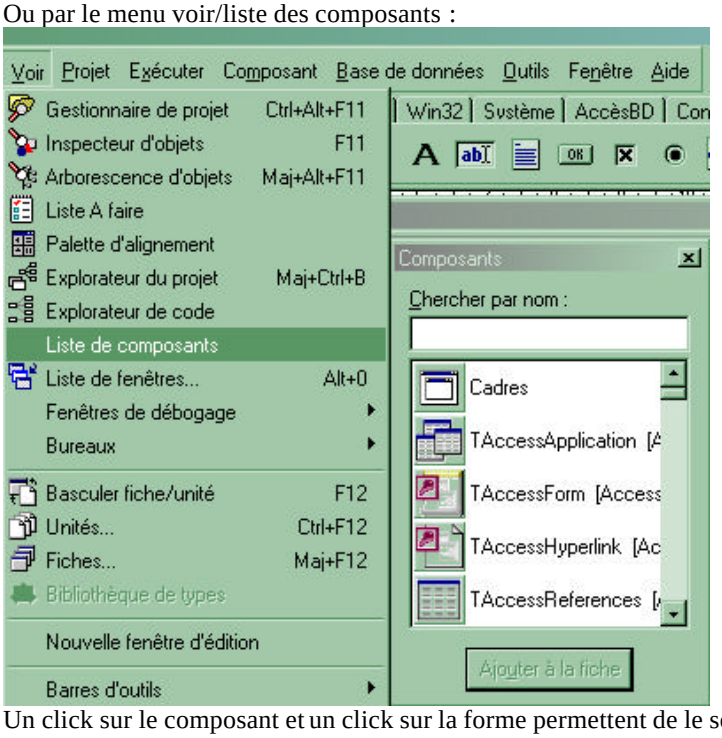
Ceci sera vu plus loin dans ce cours (Notions avancées sur les objets), mais pour plus d'information consulter l'aide Delphi :
Sommaire de l'aide/ Référence **Pascal** Objet/ Classes et objets/ Méthodes/ Liaisons de méthodes

XI. La programmation visuelle : l'EDI et la VCL

A. Utilisation

Environnement de développement intégré et la librairie de composants visuels.
On les obtient directement par les onglets :

Ou par le menu voir/liste des composants :



Un click sur le composant et un click sur la forme permettent de le sélectionner puis ensuite de le positionner.

B. Programme le plus simple avec l'EDI

Il n'y a aucun code à écrire : le simple fait d'ouvrir une nouvelle application, de la compiler et de l'exécuter, suffit à créer un exécutable ouvrant une fiche qui réagit aux événements d'agrandissement, dimensionnement, fermeture etc.…

L'exécutable produit à quand même une taille de 353 ko

C. API Windows

Le programme suivant fait exactement la même chose, mais n'utilise que l'API Windows :

```
program Window1;
{ Application Standard API Windows  écrite en
Pascal Objet.Aucun code VCL inclu. Tout est
fait au niveau de l'API Windows  .Il faut quand
même inclure Windows et Messages!}
uses  Windows,  Messages;

const AppName = 'Window1';

function WindowProc(Window: HWnd;
AMessage, WParam,
      LParam: Longint): Longint; stdcall; export;
begin
 WindowProc := 0;
 case AMessage of
  wm_Destroy: begin
     PostQuitMessage(0);
     Exit;
  end;
 end;
 WindowProc := DefWindowProc(Window,
AMessage, WParam, LParam);
end;

{ enregistrer la Classe Window  }
function WinRegister: Boolean;
var WindowClass: TWndClass;
begin
 WindowClass.Style := cs_hRedraw or
cs_vRedraw;
 WindowClass.lpfnWndProc := @WindowProc;
 WindowClass.cbClsExtra := 0;
 WindowClass.cbWndExtra := 0;
 WindowClass.hInstance := HInstance;
 WindowClass.hIcon := LoadIcon(0,
idi_Application);
 WindowClass.hCursor := LoadCursor(0,
idc_Arrow);
 WindowClass.hbrBackground :=
HBrush(Color_Window);
 WindowClass.lpszMenuName := nil;
 WindowClass.lpszClassName := AppName;
 Result := RegisterClass(WindowClass) <> 0;
end;

{ Créer la Class Window  }
function WinCreate: HWnd;
var hWindow: HWnd;
begin
 hWindow := CreateWindow(AppName,
'Fenêtre en Pascal Objet',
 ws_OverlappedWindow, cw_UseDefault,
cw_UseDefault,
 cw_UseDefault, cw_UseDefault, 0, 0,
HInstance, nil);
 if hWindow <> 0 then begin
    ShowWindow(hWindow, CmdShow);
    UpdateWindow(hWindow);
 end;
 Result := hWindow;
end;

var AMessage: TMsg;
   hWindow: HWnd;
begin
 if not WinRegister then begin
    MessageBox(0, 'l''enregistrement àéchoué',
nil, mb_Ok);
    Exit;
 end;
 hWindow := WinCreate;
 if hWindow = 0 then begin
    MessageBox(0, 'WinCreate a échoué', nil,
mb_Ok);
    Exit;
 end;
 while GetMessage(AMessage, 0, 0, 0) do
begin
   TranslateMessage(AMessage);
   DispatchMessage(AMessage);
 end;
 Halt(AMessage.wParam);
end.
```

Il est beaucoup plus long, mais il peut s'écrire avec un simple éditeur et se compiler avec DCC32 : il ne prendra que 9,5 ko !!!soit 37 fois moins de place !!

XII. Composants usuels

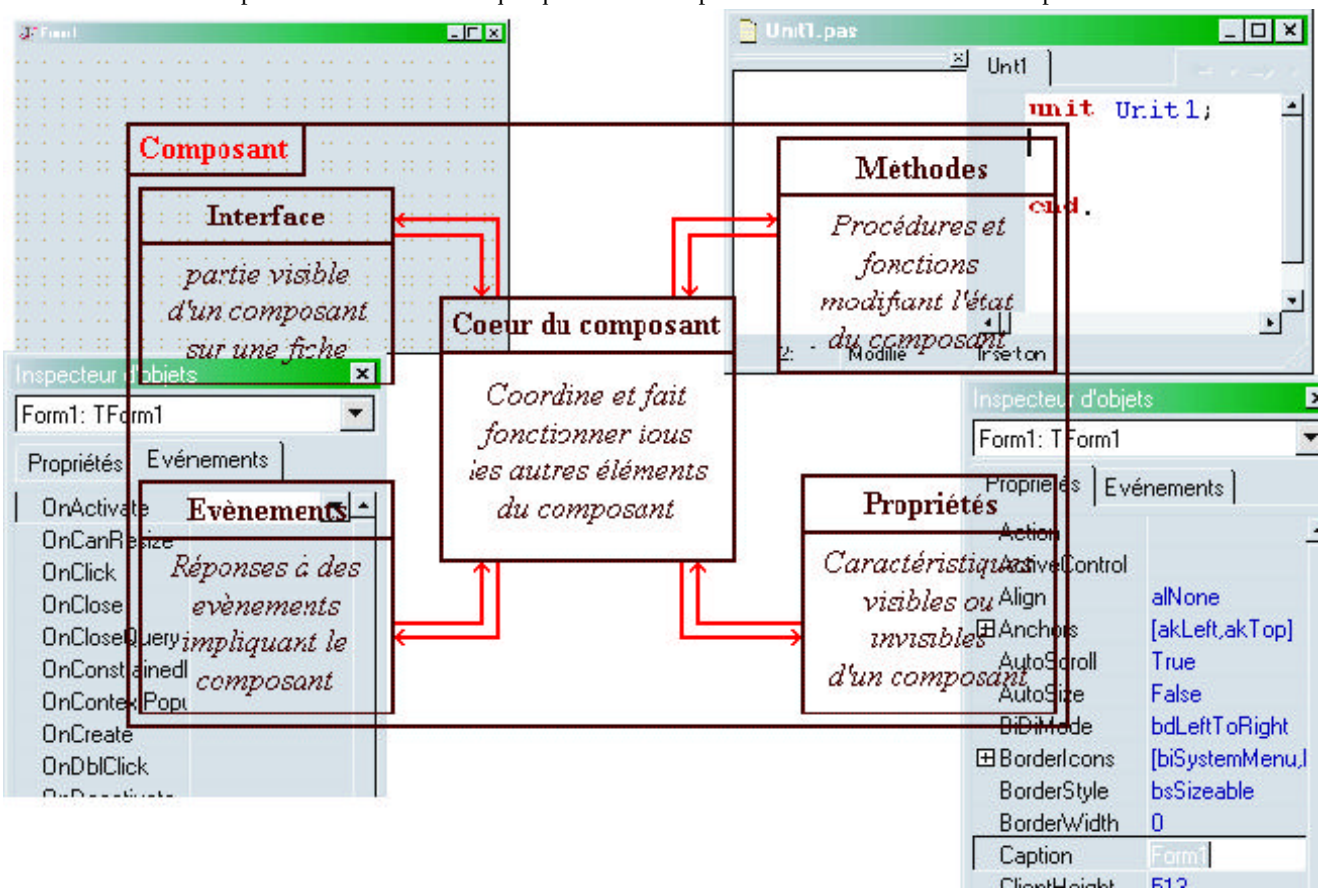
Tous les composants ont une propriété name : celle-ci ne sera donc pas répétée à chaque fois.

Lors du placement d'un composant sur la fiche, l'EDI lui affecte automatiquement un nom (du style button1). Il est impératif dans un projet, de lui attribuer un nom significatif (du style monBouton).

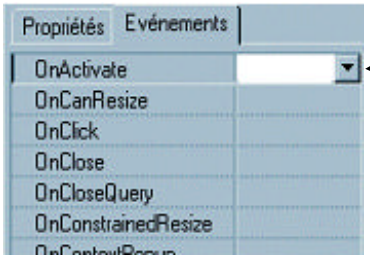
A. La fiche : Composant "Form"

1. Caractèristiques

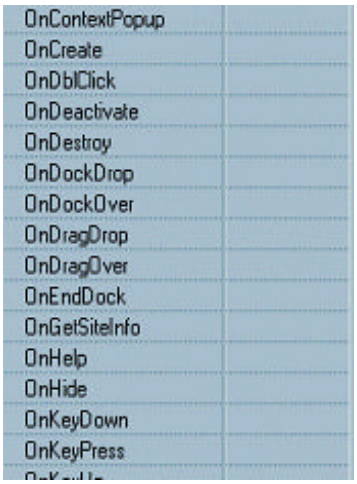
C'est LE composant conteneur classique qu'on utilise le plus souvent sans se rendre compte de cette fonctionalité pourtant indispensable.



2. Quelques événements pour «form»



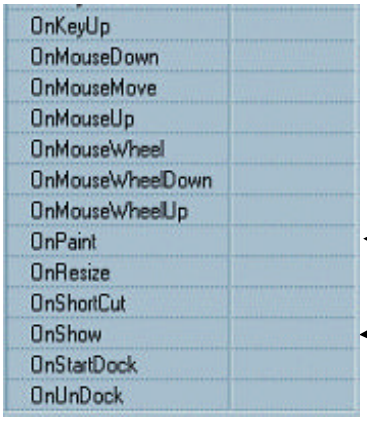
Une fiche devient active quand elle obtient la focalisation, par exemple quand l'utilisateur clique dans la fiche



Se produit à la création de la fiche.

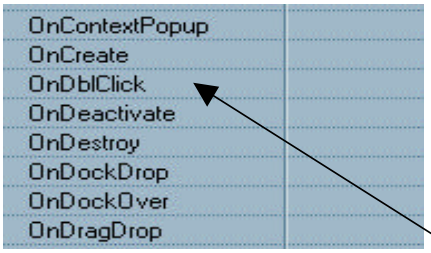
```
procedure TForm1.test(Sender: TObject);
    begin
      showmessage('hello')
    end;
```

La méthode test appelée par onActivate produira un affichage au dessus de la form
La même méthode appelée par onCreate produira un affichage avant que la form ne soit affichée.

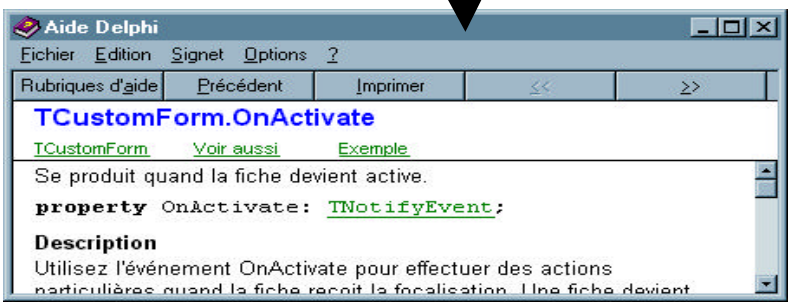


Se produit quand la fiche est redessinée.(très souvent!)

Se produit quand la fiche est affichée (c'est-à-dire quand la propriété Visible de la fiche prend la valeur True).



Autres événements :
Ils sont très nombreux et dépendent énormément du composant choisi!
Il faut consulter l'aide:
→ On clique sur l'événement dans l'inspecteur d'objet et on appuie sur F1



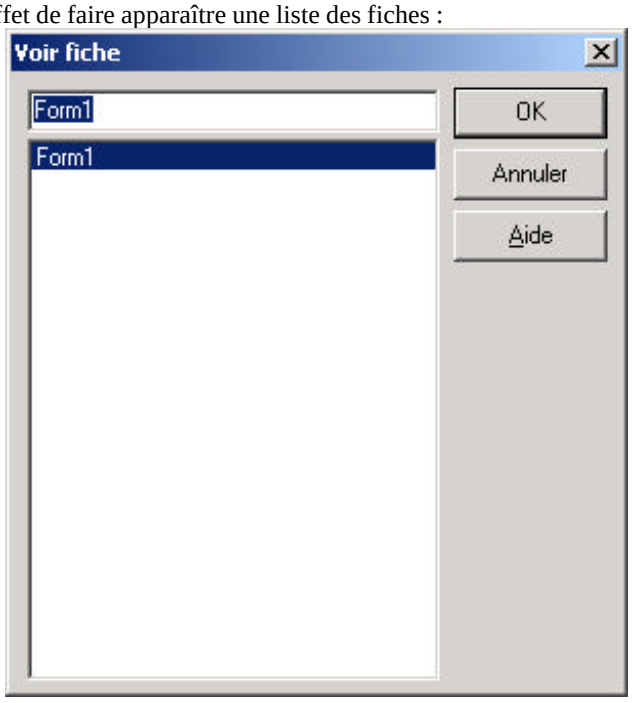
Voici un petit tableau qui donne les caractéristiques importantes des fiches. Ce genre de tableau sera répété pour chaque composant dans la suite du chapitre.

Fiche technique

Icône	(aucune)
Visibilité	Visible.
Conteneur	Oui

Les fiches ne se créent pas depuis la palette des composants, mais par une commande accessible par le menu "Fichier", choix "Nouveau..." puis "Fiche".. 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.

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

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).

- biSystemMenu affiche le menu système (en haut à gauche).
- biMinimize affiche le bouton de réduction en icône.
- biMaximize affiche le bouton d'agrandissement maximal.
- 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).

- bsDialog crée une bordure fixe standard. La fiche n'est pas redimensionnable.
- bsNone n'affiche pas de bordure. La fiche n'est pas redimensionnable.
- bsSingle affiche une bordure fine. La fiche n'est pas redimensionnable.
- bsSizeable affiche une bordure standard permettant de redimensionner la fiche.

Les deux choix suivants servent à créer des barres d'outils :

- bsSizeToolWin est similaire à bsSizeable, mais affiche une petite barre de titre.
- bsToolWindow est similaire à bsSingle, mais affiche une petite barre de titre.

Caption	Permet de fixer le texte écrit dans la barre de titre de la fenêtre.
FormStyle	Permet de fixer le style de la fiche. Ce style, normalement fixé à fsNormal, est utilisé principalement pour qu'une fiche reste affichée au premier plan (fsStayOnTop), ou pour créer une application MDI (fsMDIForm et fsMDIChild). Nous reparlerons de ce type d'application ultérieurement.
Icon	Permet de fixer une icône pour cette fiche. Utiliser le bouton  de  pour l'éditer et charger une icône (fichier .ICO). Cette icône est affichée dans sa barre de titre, à gauche, et lorsque la fenêtre est réduite en icône dans la barre des tâches. Note : il est aussi possible de fixer une icône pour l'application, qui est affichée dans la barre des tâches et pour les raccourcis vers les applications crées sous Delphi.
ModalResult	Utilisable depuis le code seulement, cette propriété, de type énuméré, permet de fermer une fiche montrée par la méthode ShowModal en lui attribuant une constante non nulle. Cette valeur est alors renvoyée par la méthode ShowModal. Le mécanisme des fenêtres modales est expliqué ci-dessous.
Position	Permet de fixer la position de la fiche. Utilisez une des valeurs proposées pour donner une position standard à la fiche. Trop de possibilités étant offertes pour être listées ici, vous pouvez les consulter dans l'aide en ligne en appuyant sur F1 après sélection de la propriété (ce qui est d'ailleurs valable avec toutes les autres propriétés, mais donne parfois des explications confuses ou imprécises).
Visible	A manipuler depuis le code source, cette propriété permet de rendre une fiche visible ou invisible. Préférez cependant l'utilisation des méthodes Show et ShowModal.
WindowState	Permet de fixer l'état de la fenêtre : o wsMaximized : donne la taille maximale à la fenêtre (même effet que le clic sur le bouton d'agrandissement). o wsMinimized : réduit la fenêtre en icône (même effet que le clic sur le bouton de réduction en icône). o wsNormal : redonne à la fenêtre son état normal, à savoir non maximisé et non icônisée.

Évènements

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-screen" lors d'un futur exemple.
OnCloseQuery	Se produit AVANT qu'une fiche soit fermée. Vous avez la possibilité de modifier un paramètre (CanClose) qui autorise ou non la fermeture de la fiche. On utilise souvent cet événement dans les formulaires pour vérifier la validité des informations entrées par l'utilisateur et éventuellement lui indiquer d'effectuer certaines corrections.
OnDeactivate	Contraire de OnActivate. Se produit lors d'une désactivation de la fiche.
OnHide	Se produit lorsque la fiche est cachée, c'est-à-dire lorsque sa propriété Visible passe de True à False.
OnResize	Se produit à chaque fois que les dimensions de la fiche changent. Cet événement permet éventuellement de mettre à jour certaines dimensions de composants pour maintenir un effet visuel.
OnShow	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.

3. Modales ou non ?

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 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".

B. 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 accéder 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 accédez à la même chose).

Les propriétés importantes du composant "MainMenu" sont décrites ci-dessous :

Propriétés

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.

C. Composant "TPopupMenu"

Fiche technique

Icône	
Visibilité	Invisible à la création, peut être visible à l'exécution
Conteneur	Non

Ce composant permet de créer un menu contextuel. Les menus contextuels ont 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").

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	Propriété objet. Permet l'accès à l'éditeur de menus.

Événements

OnPopup	Se produit juste avant que le menu contextuel soit montré. Utiliser une procédure répondant à cet évènement 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 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 proposent).

D. Composant "Label"

Fiche technique

Icône	
Visibilité	Visible
Conteneur	Non

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.

Remarque :

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 simp les étiquettes. Les composants "Label" sont très souvent utilisés et le seront donc dans les manipulations futures.

Propriétés

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 étiqueté.
Layout	Alter-ego vertical de "Alignment".
Visible	Permet de montrer ou de cacher le label. Cette propriété fait partie des grands classiques qui ne seront plus repris dans la suite.
WordWrap	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.

E. Composant "Edit"

Fiche technique

Icône	
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 réglée, 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

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'active la lecture seule de la zone d'édition. Lorsque "ReadOnly" vaut "true", la lecture est toujours possible mais l'écriture impossible.
Text	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.

Événements

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 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.

F. 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

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'accéder 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'utilis ateur à utiliser des tabulations à l'intérieur du mémo. Bien qu'intéressante, cette possibilité l'empêche de sortir du mémo en utilisant la touche Tab comme il le fait avec d'autres contrôles (je ne crois pas avoir auparavant expliqué qu'un composant est nommé "contrôle" lors de l'exécution) tels les boutons.
WordWrap	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.

Événements

OnChange	Analogue à l'événement OnChange des composants Edit.

G. 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 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

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 beaucoup de composant, Enabled décide si le bouton est utilisable ou non.
ModalResult	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.

Événements

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 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.

H. 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

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.

Événements

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.

XIII. Programmation événementielle et envoi de messages

A. Définition

On voit ci-dessus que les objets des différents composants (visuels ou non) réagissent lorsqu'une action extérieure intervient : par exemple appuis sur une touche du clavier, déplacement de la souris, top de l'horloge interne ou toute autre interruption matérielle ou non.

Lorsqu'un événement survient, un message est envoyé aux procédure de gestion d'événements avec les caractéristiques de l'expéditeur du message. Bien souvent, on ne se préoccupe pas de l'expéditeur, on indique dans l'inspecteur d'objets l'onglet évènement (par un double-clic) le nom de la méthode chargée de le gérer.

B. Exemple

Créons une fiche avec 2 boutons appelés (propriété name) respectivement Button1 et Button2 et comportant tous deux la même inscription : « Appuyez ici » (propriété caption). De plus la propriété visible de l'un est à true et de l'autre est false. On fera un double-clic indique dans l'inspecteur d'objets l'onglet évènement pour chaque bouton sur l'évènement ***OnMouseMove***

Le but du programme est de rendre invisible le bouton sur lequel l'utilisateur essaye de cliquer et de rendre l'autre visible :

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1MouseMove(Sender:
TObject; Shift: TShiftState; X,Y: Integer);
    procedure Button2MouseMove(Sender:
TObject; Shift: TShiftState; X,Y: Integer);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.Button1MouseMove(Sender:
TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  button1.visible := false;
  button2.visible := true
end;

procedure TForm1.Button2MouseMove(Sender:
TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  button2.visible := false;
  button1.visible := true
end;

end.
```

C. Réponse à un événement

Reprenons l'exemple (page 67)de création dynamique d'une forme et d'un bouton et ajoutons-lui une réaction à un click :

```
program tst_form;

uses   Forms, stdctrls, dialogs;

type bidon=class
    procedure repondre(expediteur : tobject);
  end;

  procedure bidon.repondre(expediteur :
tobject);
```

```
  begin
    showmessage ('ok')
  end;

var
  maForm:tform;
  gamelle: bidon;
begin
  Application.Initialize;
  Application.CreateForm(TForm, maForm);
  maForm.visible := true;
  maForm.top:=30;
  maForm.left:=100;
  maForm.height:=300;
  maForm.width:=400;
  maForm.caption := 'nouvelle fiche';
  with TButton.Create(maForm)do begin
    Parent := maForm;
    top := 20;
    left := 20;
    height := 20;
    width := 200;
    caption := 'nouveau bouton';
    onclick := gamelle.repondre;
  end;
  Application.Run;
end.
```

La gestion d'un événement se fait par une méthode (et non une simple procédure) ce qui explique la création d'une classe bidon instanciée par l'objet gamelle . En Réalité, la méthode fait partie de la classe créée spécialement et qui réagit à l'événement qui lui correspond !

D. Retour sur les fichiers DFM

L'exemple précédent, s'il est correct, n'est pas « propre » : fabriquer une clase bidon pour traiter un événement n'est pas des plus lisible. Il vaut mieux créer une classe englobant la fiche, le bouton et le traitement de l'événement comme suit :

```
program Project2;

uses
  Forms, stdctrls, dialogs;

type TmaForm=class(tform)
    monbouton : TButton;
    procedure repondre(expediteur : tobject);
  end;

  procedure TmaForm.repondre(expediteur :
tobject);
  begin
    showmessage ('ok')
  end;

var
  maForm:TmaForm;
{$R prj.dfm}

begin
  Application.Initialize;
  Application.CreateForm(TmaForm, maForm);
  maForm.visible := true;
  maForm.top:=30;
  maForm.left:=100;
  maForm.height:=300;
  maForm.width:=400;
  maForm.caption := 'nouvelle fiche';
  maForm.monbouton:=
TButton.Create(maForm);
  with maForm.monbouton do begin
    Parent := maForm;
    top := 20;
    left := 20;
    height := 20;
    width := 200;
    caption := 'nouveau bouton';
    onclick := maForm.repondre;
  end;
  Application.Run;
end.
```

On remarquera la présence de *{$R prj.dfm}* (ou de *{$R *.dfm}* s'il n'y a pas de confusion possible). Le fichier ***prj.dfm*** est un simple fichier texte comportant :

```
object maForm: TmaForm  end
```

Il contient presque rien, mais il est **Obligatoire** !ceci est à mettre en relation avec le chapitre sur « Constitution des fichiers DPR, DMF, PAS »

E. Exemple pratique : bouton à double fonctionnalité

Le but est de du programme est de changer la fonctionnalité d'un bouton : par exemple, *on* souhaite mettre à jour une liste quelconque et l'on dispose d'un bouton intitulé (*caption*) ***ajouter*** sur lequel on peut cliquer. Le même bouton permet de valider la mise à jour son intitulé se transformant en ***valider.*** Bien entendu à chaque intitulé, correspond une fonctionnalité différente entraînant un traitement différent. Entre autre, 3 solutions s'offrent à nous :

1. 1 Bouton ➔ 1 évènement

Dans l'exemple qui suit, on a un bouton, et l'on teste son intitulé : Attention, il faut faire attention à la chasse (Majuscule / Minuscule) des ***captions***. Selon son titre, le traitement s'effectue selon le cas 1 ou 2.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender:
TObject);
begin

  with button1 do begin
    if caption='Ajouter' then begin
      caption:='Valider';
  //traitement cas 1
    end else begin
      caption:= 'Ajouter';
  //traitement cas 2
    end;
  end;
end;
end.
```

Il n'y a qu'un seul bouton, l'utilisateur ne voit qu'un bouton dont l'intitulé change et qui réagit en conséquence.

2. 2 Boutons ➔ 2 évènements

Ici, on utilise 2 boutons que l'on cache (propriété ***visible***) alternativement.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender:
TObject);
begin
  button2.Width:=button1.Width;
  button2.Top:=button1.Top;
  button2.Left :=button1.Left;
  button2.Height:=button1.Height;
  button2.Visible:=false;
end;

procedure TForm1.Button1Click(Sender:
TObject);
begin
  button2.Visible:=True;
  button1.Visible:=false;
  //traitement cas 1
end;

procedure TForm1.Button2Click(Sender:
TObject);
begin
  button1.Visible:=True;
  button2.Visible:=false;
```

```
  //traitement cas 2
end;
```

```
end.
```

L'utilisateur à toujours l'impression de ne voir qu'un seul bouton, car le programmeur, qui a placé le second n'importe où sur la fiche, a pris soin dans l'évènement ***FormCreate*** de donner au ***button2*** les mêmes caractéristiques que ***button1***. Et donc l'utilisateur le voit apparaître au même endroit et croit que c'est le même bouton dont seul le titre a changé.

L'avantage de cette deuxième solution est d'éviter de faire un test pour savoir dans quel cas l'on se trouve. L'inconvénient est d'avoir un objet (bouton) supplémentaire.

3. 1 Bouton ➔ 2 événements

Dans cette troisième solution, on ne manipule qu'un seul bouton, mais chaque click entraîne un changement de gestionnaire d'évènement ***OnClick*** du ***button1***

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click1(Sender: TObject);
    procedure Button1Click2(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click1(Sender:
TObject);
begin
 with button1 do begin
   OnClick:=Button1Click2;
   caption:= 'Valider';
 end;
  //traitement cas 1
end;

procedure TForm1.Button1Click2(Sender:
TObject);
begin
 with button1 do begin
   OnClick:=Button1Click1;
   caption:= 'Ajouter';
 end;
  //traitement cas 2
end;

end.
```

4. Conclusion

Les 3 solutions se valent, seules les techniques diffèrent. La première est la plus compacte, mais souffre de 2 inconvénients dus au ***if*** :

- Les 2 traitements sont dans la même méthode ➔ séparation des taches moins évidente donc lisibilité moindre.

Le test sur les ***caption*** doit absolument respecter la chasse des caractères : par exemple si la **caption** vaut ***'Valider'*** et que le test se fait sur ***'valider'***, celui-ci vaudra ***false*** !.

XIV. Événements et Envoi de messages

Un événement est une action ou une occurrence détectée par un programme. La plupart des applications modernes sont dites pilotées par événements, car elles sont conçues pour répondre à des événements. Dans un programme, le programmeur n'a aucun moyen de prévoir la séquence exacte des actions que va entreprendre l'utilisateur. Il peut choisir un élément de menu, cliquer sur un bouton ou sélectionner du texte. Vous allez donc écrire le code qui gère chacun des événements qui vous intéressent au lieu d'écrire du code s'exécutant toujours selon le même ordre.

A. Evénements utilisateur

Les événements utilisateur sont des actions initiées par l'utilisateur. Les événements utilisateur sont, par exemple, *OnClick* (l'utilisateur a cliqué avec la souris), *OnKeyPress* (l'utilisateur a appuyé sur une touche du clavier) et *OnDblClick*

(l'utilisateur a double-cliqué sur un bouton de la souris). Ces événements sont toujours rattachés à une action de l'utilisateur.

B. Evénements système

Ce sont des événements que le système d'exploitation déclenche pour vous. Par exemple, l'événement OnTimer (le composant Timer déclenche l'un de ces événements lorsqu'un intervalle prédéfini s'est écoulé), l'événement OnCreate (le composant est en train d'être créé), l'événement OnPaint (un composant ou une fenêtre a besoin d'être redessiné), etc. En règle générale, ces événements ne sont pas directement déclenchés par des actions de l'utilisateur..

Toutes les classes Delphi disposent d'un mécanisme intégré pour gérer les messages : ce sont les méthodes de gestion des messages ou gestionnaires de messages.

L'idée sous-jacente aux gestionnaires de messages est la suivante : un objet reçoit des messages qu'il répartit selon le **message** en appelant une méthode choisie dans un ensemble de méthodes spécifiques. Un gestionnaire par défaut est appelé si aucune méthode n'est définie pour le **message**.

Le diagramme suivant illustre le fonctionnement du système de répartition de **message** :

Evénement→MainWndProc→WndProc→Dispatch→Gestionnaire

Exemple 1

La procedure AppMessage définie ci-dessous capture tous les évènements OnMessage de l'application. Attention cette procédure est appelée pour chaque message si vous n'y prenez garde, elle peut ralentir votre application les messages arrivent même s'ils sont destinés à une autre fenêtre que la fenêtre principale (essayez dans la fenêtre du ShowMessage) }

```
…
Type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    procedure AppMessage(var Msg: TMsg; var Handled: Boolean);
  end;
Var   Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
{-----------------------------------------------------
affectation de AppMessage en temps que procédure déclenchée par OnMessage
-----------------------------------------------------}
begin
  Application.OnMessage := AppMessage;
end;
procedure TForm1.AppMessage(var Msg: TMsg;  var Handled: Boolean);
{ procédure lancée par chaque événement
Affectez la valeur True à Handled si le message a été complètement  géré afin d'empêcher le
traitement normal du message. }
begin
   if (Msg.message =WM_KEYDOWN) then
      ShowMessage('message WM_KEYDOWN  dans AppMessage')
   else
      if (Msg.message =WM_RBUTTONDOWN) then
           ShowMessage('message WM_RBUTTONDOWN dans AppMessage');
   { handled=true; aurait empéché la suite du traitement normal du message:
il n'aurait donc pas été traité par WindProc}
end;
```

Déclaration d'une méthode spécifique pour le traitement du messageu envoyé lors d'un click droit sur la souris

```
Type
  TForm1 = class(TForm)
    Panel1: TPanel;
```

```
  private
    procedure SourisCliqueDroit( var msg:TMessage); message WM_RBUTTONDOWN;
end;
Var   Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.SourisCliqueDroit( var msg:TMessage);
begin
  ShowMessage('Message détecté dans SourisCliqueDroit');
  inherited; // si on veut continuer à propager le message
end;
```

Le message est envoyé par la Tform1 et non par les objets qui lui appartiennent

Un clic droit dans le panel1 n'en envoie pas !

```
...
Const
WM_MESSAGEPERSO = WM_USER + 1;
{ le 1 peut être remplacé par le nombre de votre choix  ne pas utiliser le même nombre  pour un
autre message }
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    procedure WMMessagePerso(var Msg : TMessage);
          message WM_MESSAGEPERSO;
// procédure destinée à recevoir notre messageend;
Var   Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  SendMessage(Form1.Handle,          WM_MESSAGEPERSO,0,0);
 //envoi de notre message à  Form1
end;
procedure TForm1.WMMessagePerso(var Msg : TMessage);
begin
  ShowMessage('Message Perso reçu par WMMessagePerso');
end;
```

Sur une forme, un bouton et un panel

```
procedure TForm1.Button1Click(Sender: TObject);
begin
      Panel1.Color:=clRed;
      sleep(3000);                  // delays 3000 msec
      Panel1.Color:=clGreen;
end;
```

On pourrait espérer que le clic sur le bouton entraîne une coloration rouge du panel pendant 3 secondes et qu'ensuite, il devienne vert.
Il n'en est rien car la méthode repaint ne sera appelée qu'en sortie de procédure : le message envoyé est capturé et traité par cette procédure qui ne rend la main que lorsqu'elle a fini.
Et donc le vert vient recouvrir immédiatement le rouge que l'œil n'a pas le temps de percevoir

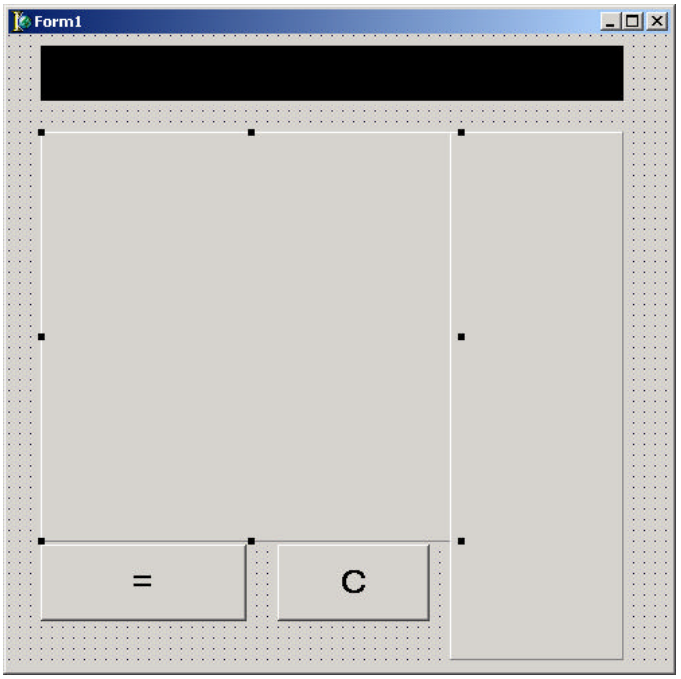
La solution consiste à laisser passer les messages

```
procedure TForm1.Button1Click(Sender: TObject);
begin
      Panel1.Color:=clRed;
      Application.ProcessMessages;
```

```
    sleep(3000); // delays 3000 msec
    Panel1.Color:=clGreen;
End
```

XV. Exemple récapitulatif

Il s'agit de réaliser un programme simulant une calculatrice conforme aux spécifications du TP2, mais il ne s 'agit pas d'un corrigé dans la mesure ou le cours n'était pas suffisemment avancé pour fournir une réponse de cette nature .




```
unit Unit1;

interface

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

type
   tbbout = class
      private
        nl,nc :byte;
        proprio:tcomponent;
      public
        tb : array of array of TButton;
        constructor create(nl,nc:byte;proprio:tcomponent;tt:TNotifyEvent;  kp:TKeyPressEvent);
        procedure dimensionne;
      end;

  Calculatrice = class
      private
       Op1,Op2 : Extended;//Operandes de l'entree en cours
       Operation : char;//stocke l'operation demandée parmi {+,-,/,*}
       entree_en_cours : string;
       nbop:byte;
      public
       constructor create;
       Function Gere_chif(Chaine:string) : string;
       Function Gere_oper(Chaine:string) : string;
       Procedure Clear;//Efface tous les champs après operation
       Function ExecOperation : Extended;
      end;

  TForm1 = class(TForm)
    Button_egal: TButton;
    Panel_chif: TPanel;
    Panel_oper: TPanel;
    Button_clear: TButton;
    afficheur: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
    procedure FormResize(Sender: TObject);
    procedure Button_egal_Click(Sender: TObject);
    procedure Button_clear_Click(Sender: TObject);
    procedure FormKeyPress(Sender: TObject; var Key: Char);
  private
    Calculette:Calculatrice;
    boutons_oper,boutons_chif : tbbout;
    old_width : integer;
    procedure traite_chif(Sender: TObject);
    procedure traite_oper(Sender: TObject);
  end;

var
  Form1: TForm1;

Implementation {$R *.dfm}

const MINI=378;

{---------- classe calculatrice ----------}
constructor Calculatrice.create;
begin
  inherited;
  clear;
  nbop:=0;
end;

procedure Calculatrice.Clear;
begin
  Op1:=0;
  Op2:=0;
  Operation:=#0;
  entree_en_cours:='';

end;

function Calculatrice.Gere_oper(chaine:string):string ;
begin
  inc(nbop);
  if entree_en_cours='' then  entree_en_cours:= '0';
  case nbop of
  2 :  op1:=ExecOperation;
  1 :  Op1:=StrtoFloat(entree_en_cours);
  else op1:=0;
  end;
  Operation:=Chaine[1];
  result:=FloatToStr(op1);
  entree_en_cours:='';
end;

Function Calculatrice.Gere_chif(chaine:string) : string;
begin
  if chaine=',' then begin
    if entree_en_cours='' then  entree_en_cours:='0,';
    result:=entree_en_cours;
    if ansipos(',',entree_en_cours) <> 0 then Exit;//éviter de mettre 2 virgules!
  end;
  if chaine='+/-'  then begin // il faut juste prendre l'opposé
```

```
   if (entree_en_cours<>'') then
     entree_en_cours:=FloattoStr(-StrtoFloat(entree_en_cours))
  end else
   if entree_en_cours='0' then
     entree_en_cours:=Chaine
   else
     entree_en_cours:=entree_en_cours+Chaine;
  result:=entree_en_cours;
end;

function Calculatrice.ExecOperation: Extended;
begin
   op2:=StrtoFloat(entree_en_cours);
   if nbop>0 then dec(nbop);
    Case Operation of
         '+' : Result:=Op1+Op2;
         '-' : Result:=Op1-Op2;
         '*' : Result:=Op1*Op2;
         '/' : begin
               if op2=0 then Exit;
               Result:=Op1/Op2;
               end;
          else result:=0;
    end;
  entree_en_cours:=floatToStr(result);
  Operation:=#0;
end;

{---------- classe tbbout ----------}

constructor tbbout.create(nl,nc:byte;proprio:tcomponent;tt:TNotifyEvent;kp:TKeyPressEvent);
var i,j:byte;
begin
  inherited create;
  self.nl:=nl;
  self.nc:=nc;
  self.proprio:=proprio;
  setlength(tb,nl,nc);
  for i:= 0 to nl-1 do
    for j:=0 to nc-1 do begin
      tb[i,j]:= TButton.create(proprio);
      with tb[i,j] do begin
        parent:= proprio as TWinControl;
        caption := intTostr(i*nc+j+1);
        Font.Charset := DEFAULT_CHARSET;
        Font.Color := clWindowText ;
        Font.Height := -24 ;
        Font.Name := 'MS Sans Serif' ;
        Font.Style := [fsBold]  ;
        onclick := tt;
        onkeypress:=kp;
      end;
    end;
end;

procedure tbbout.dimensionne;
var i,j:byte;
w,h,l,t : integer;
```

```
begin
  w:=((proprio as TWinControl).width-30) div nc -20;
  h:=((proprio as TWinControl).height-30) div nl -20;
  t:=-h;
  for i:= 0 to nl-1 do begin
    t:=t+h+20;
    l:=-w;
    for j:=0 to nc-1 do begin
      l:=l+w+20;
      tb[i,j].width:=w;
      tb[i,j].height:=h;
      tb[i,j].top:=t;
      tb[i,j].left:=l;
    end;
  end;
end;

{-------------------- classe Tform1 ----------------------------}

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  calculette.Destroy;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    width:=height;
    self.ChangeScale(width,old_width);
    old_width:=width;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  calculette:=Calculatrice.Create;
  Panel_oper.BevelInner := bvNone;  Panel_oper.BevelOuter := bvNone;
  Panel_chif.BevelInner := bvNone;  Panel_chif.BevelOuter := bvNone;
  boutons_chif := tbbout.create(4,3,panel_chif, traite_chif,FormKeyPress);
  boutons_oper := tbbout.create(4,1,panel_oper,traite_oper,FormKeyPress);
  boutons_chif.tb[3,0].caption:=',';
  boutons_chif.tb[3,1].caption:='0';
  boutons_chif.tb[3,2].caption:='+/-';
  boutons_oper.tb[0,0].caption:='+';
  boutons_oper.tb[1,0].caption:='-';
  boutons_oper.tb[2,0].caption:='*';
  boutons_oper.tb[3,0].caption:='/';
  boutons_chif.dimensionne;
  boutons_oper.dimensionne;
  self.Constraints.MinHeight := MINI;
  self.Constraints.MinWidth  := MINI;
  old_width:=width;
  calculette.Clear;
  afficheur.caption:='0';
end;

procedure TForm1.traite_oper(Sender: TObject);
begin
    afficheur.caption:=calculette.Gere_oper(string((Sender as TButton).Caption));
    Button_egal.SetFocus;
```

```
end;

procedure TForm1.traite_chif(Sender: TObject);
begin
   afficheur.caption:=calculette.Gere_chif(string((Sender as TButton).Caption));
   Button_egal.SetFocus;
end;

procedure TForm1.Button_egal_Click(Sender: TObject);
begin
  afficheur.caption:=floatToStr(calculette.ExecOperation);
end;

procedure TForm1.Button_clear_Click(Sender: TObject);
begin
  calculette.Clear;
  afficheur.caption:='0';
  Button_egal.SetFocus;
end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
   if key='.' then key:=',';
   case upcase(key) of
     '=' :  afficheur.caption:=floatToStr(calculette.ExecOperation);
     'C' :  Button_clear_Click(Sender);
     '0'..'9',',': afficheur.caption:=calculette.Gere_chif(key);
     '+','-','*','/': afficheur.caption:=calculette.Gere_oper(key);
   end;
   Button_egal.SetFocus;
end;

end.
```

XVI. Transtypage des objets : l'opérateur As et is

1. Exemple 1

Reprenons l'exemple ci dessus en n'écrivant qu'une procédure ButtonMouseMove au lieu de 2. Dans l'inspecteur d'objet, indiquons que l'événement onMouseMove de chaque bouton est géré par ButtonMouseMove (au lieu de Button1mouseMove et Button2ouseMove). Il faut en contre-partie identifier l'auteur du message qui a déclanché l'événement :

```
unit Unit1;
interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure ButtonMouseMove(Sender:
TObject; Shift: TShiftState; X, Y: Integer);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
```