Chapitre 5 : Programmation objet et événementielle

5.1 Introduction à la programmation orientée objet (POO)

- concepts fondamentaux de la POO
- initiation à la conception orientée objet (OOD,UML...)

5.2 Programmez objet avec Delphi

- Description générale de Delphi
- Modules dans Delphi
- Delphi et la POO
- Les propriétés en Delphi

5.3 Polymorphisme avec Delphi

- Polymorphisme d'objet
- Polymorphisme de méthodes
- Polymorphisme de classe abstraite
- Exercice traité sur le polymorphisme

5.4.Programmation événementielle et visuelle

- programmation visuelle basée sur les pictogrammes
- programmation orientée événements
- normalisation du graphe événementiel
- tableau des actions événementielles
- interfaces liées à un graphe événementiel
- avantage et modèle de développement RAD visuel
- Notice sur les interfaces de communication

5.5.les événements avec Delphi

- Pointeur de méthode
- Affecter un pointeur de méthode
- Un événement est un pointeur de méthode
- Quel est le code engendré
- Exercice-récapitulatif
- Notice méthodologique pour créer un nouvel événement
- Code pratique : une pile lifo événementielle

- Exemple traité : Un éditeur de texte

5.6.programmation défensive : les exceptions

- **Notions de défense et de protection**

  Outils participant à la programmation défensive
  Rôle et mode d'action d'une exception
  Gestion de la protection du code
    Fonctionnement sans incident
    Fonctionnement avec incident
    Effets dus à la position du bloc except...end
    Fonctionnement sans incident
    Fonctionnement avec incident
    Interception d'une exception d'une classe donnée
    Ordre dans l'interception d'une exception
    Interception dans l'ordre de la hiérarchie
    Interception dans l'ordre inverse

- **Traitement d'un exemple de protections**

  Le code de départ de l'unité
  Code de la version.1 (premier niveau de sécurité)
  Code de la version.2 (deuxième niveau de sécurité)

❖ Code pratique : une pile Lifo avec exception

5.1 Introduction à la programmation orientée objet

Plan du chapitre:

Introduction

Le lecteur qui connaît les fondements de ce chapitre peut l'ignorer et passer au chapitre suivant. Dans le cas contraire, **la programmation visuelle** étant intimement liée aux outils et aux concepts **objets**, ce chapitre est un minimum incontournable.

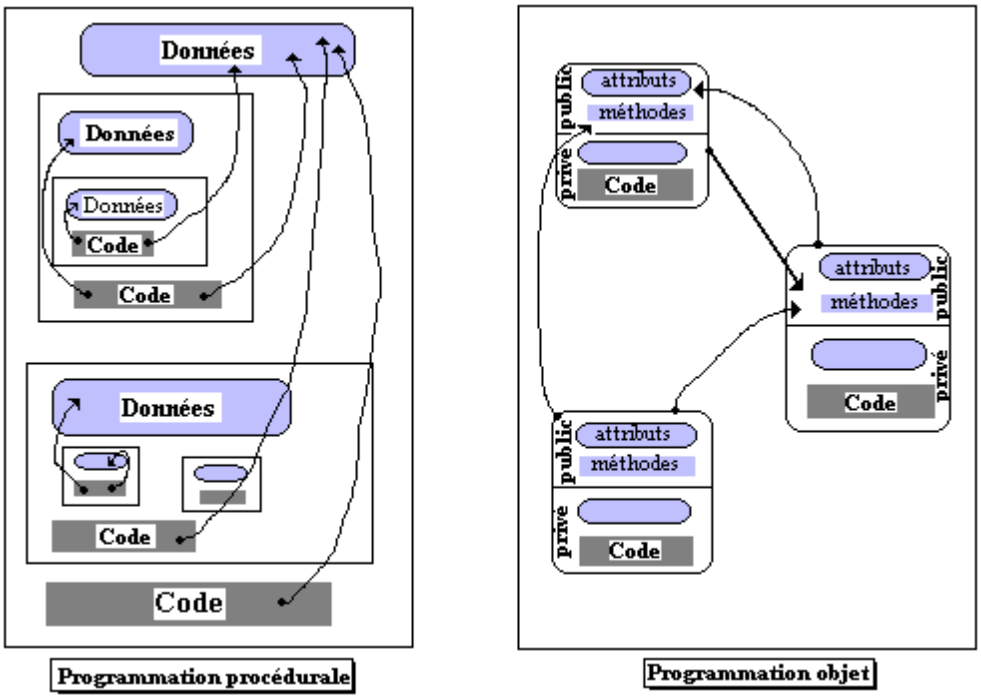
La programmation classique ou *procédurale* telle que le débutant peut la connaître à travers des langages de programmation comme Pascal, C etc... traite les programmes comme un ensemble de données sur lesquelles agissent des procédures. Les procédures sont les éléments actifs et importants, les données devenant des éléments passifs qui traversent *l'arborescence de programmation* procédurale en tant que flot d'information.

Cette manière de concevoir les programmes reste proche des machines de Von Neuman et consiste en dernier ressort à traiter indépendamment les données et les algorithmes (traduits par des procédures) sans tenir compte des relations qui les lient.

En introduisant la notion de modularité dans la programmation structurée descendante, l'approche diffère légèrement de l'approche habituelle de la programmation algorithmique classique. Nous avons défini des *machines abstraites* qui ont une autonomie relative et qui possèdent leurs propres structures de données; la conception d'un programme relevait dès lors essentiellement de la description des interactions que ces machines ont entre elles.

La programmation orientée objet relève d'une conception ascendante définie comme des "**messages**" échangés par des entité de base appelées objets.

comparaison des deux topologies de programmation

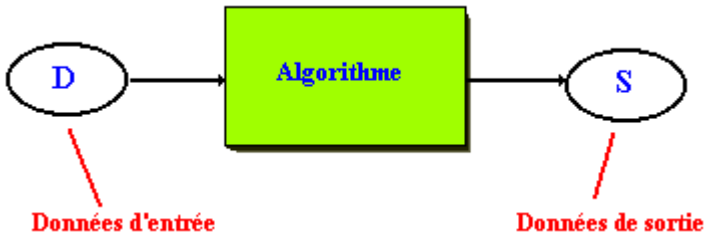


Les langages objets sont fondés sur la connaissance d'une seule catégorie d'entité informatique : l'objet. Dans un objet, traditionnellement ce sont les données qui deviennent

prépondérantes. On se pose d'abord la question : "de quoi parle-t-on ?" et non pas la question "que veut-on faire ?", comme en programmation algorithmique. C'est en ce sens que les machines abstraites de la programmation structurée modulaire peuvent être considérées comme des pré-objets.

En fait la notion de TAD est utilisée dans cet ouvrage comme spécification d'un objet, en ce sens nous nous préoccupons essentiellement des services offerts par un objet indépendamment de sa structure interne.

Programmation structurée :



1. Concepts fondamentaux de La P.O.O

Nous écrirons P.O.O pour : programmation orientée objet.

Voici trois concepts qui contribuent à la puissance de la P.O.O.

- Concept de **modélisation** à travers la notion de classe et **d'instanciation** de ces classes.
- Concept **d'action** à travers la notion d'envoi de messages et de méthodes à l'intérieur des objets.
- Concept de construction par **réutilisation et amélioration** par l'utilisation de la notion d'héritage.

1.1 les objets

Objet

Un **module** représente ***un objet ou une classe d'objet*** de l'espace du problème et non une étape principale du processus total, comme en programmation descendante.

Recenser les objets du monde réel

Lors de l'analyse du problème, on doit faire l'état de l'existant en recensant les objets du monde réel. On établit des classes d'objets et pour chaque objet on inventorie les connaissances que l'on a sur lui :

- Les connaissances déclaratives,
- les connaissances fonctionnelles,
- l'objet réel et les connaissances que l'on a sur lui sont regroupés dans une même entité.

On décrit alors les systèmes en classes d'objets plutôt qu'en terme de fonctions.

Par Exemple : **Une application de gestion bancaire est organisée sur les objets comptes, écritures, états.**

Les objets rassemblent une partie de la connaissance totale portant sur le problème. Cette connaissance est répartie sur tous les objets sous forme déclarative ou procédurale.

Les objets sont décrits selon le modèle des structures abstraites de données (TAD) : ils constituent des boîtes noires dissimulant leur implantation avec une interface publique pour les autres objets. Les interactions s'établissant à travers cette interface.

Vocabulaire objet

Encapsulation

c'est le fait de réunir à l'intérieur d'une même entité (*objet*) le code (*méthodes*) + données (*champs*).

Il est donc possible de masquer les informations d'un objet aux autres objets.

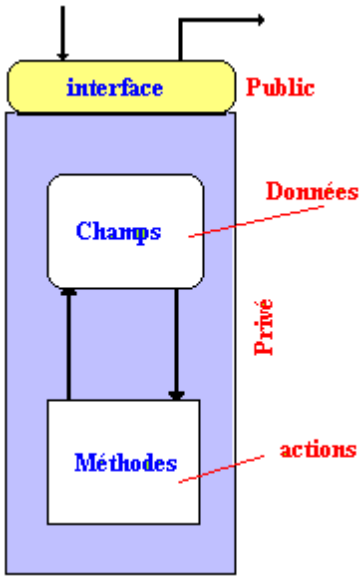
Deux niveaux d'encapsulation sont définis :

Privé

les **champs** et les **méthodes masqués** sont dans la partie privée de l'objet.

Public

les **champs** et les **méthodes visibles** sont dans la partie interface de l'objet.



Les notions de **privé** et de **public** comme dans un objet n'ont trait qu'à la communication entre deux objets, à l'intérieur d'un objet elles n'ont pas cours.

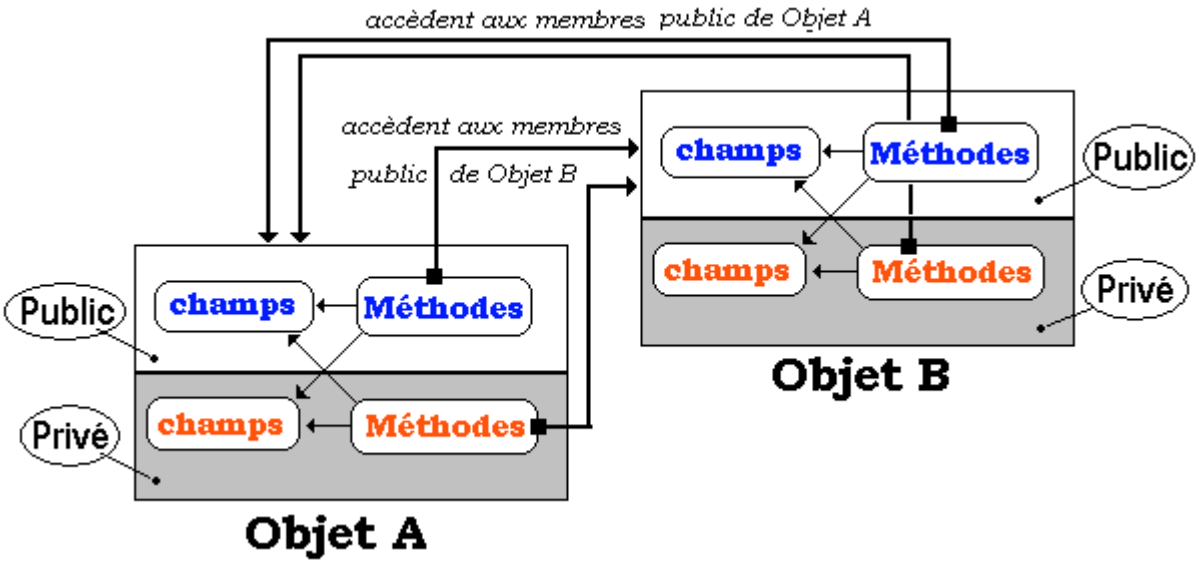


Figure sur la visibilité entre deux objets
Les méthodes de public ou privé de l'objet A accèdent et peuvent utiliser les méthodes et les champs public de B.
Les méthodes de public ou privé de l'objet B accèdent et peuvent utiliser les méthodes et les champs public de A.

1.2 les classes

Postulons une analogie entre les objets matériels de la vie courante et les objets informatiques. Un objet de tous les jours est souvent obtenu à partir d'un moule industriel servant de modèle pour en fabriquer des milliers. Il en est de même pour les objets informatiques.

Classe

Une **classe** est une sorte de ***moule ou de matrice*** à partir duquel sont engendrés les objets réels qui s'appellent des ***instances*** de la classe considérée.

Une classe contient

Des attributs (ou champs, ou variables d'instances).
Les attributs de la classe décrivent la structure de ses instances (les objets).

Des méthodes (ou opérations de la classe).
Les méthodes décrivent les opérations qui sont applicables aux instances de la classe.

Membres

les **attributs** et les **méthodes** d'une classe sont des **membres** de la classe.

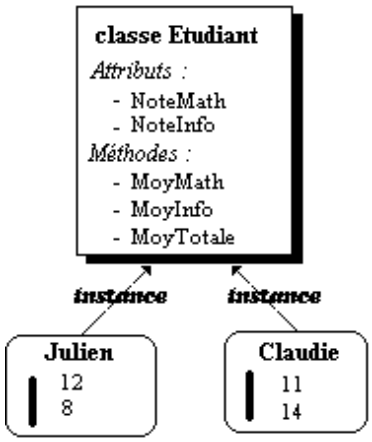
Instance

Un objet de classe A est appelé aussi une **instance** de classe A, l'opération de construction d'un objet s'appelle alors **l'instanciation**.

Remarque

En POO, programmer revient donc à décrire des classes d'objets, à caractériser leur structure et leur comportement, puis à instancier ces classes pour créer des objets réels. Un objet réel est matérialisé dans l'ordinateur par une zone de mémoire que les données et son code occupent.

Un exemple : des étudiants
Supposons que chaque étudiant soit caractérisé par sa note en mathématiques (NoteMath) et sa note en informatique (NoteInfo). Un étudiant doit pouvoir effectuer éventuellement des opérations de calcul de ses moyennes dans ces deux matières (MoyMath, MoyInfo)et connaître sa moyenne générale calculée à partir de ces deux notes (MoyTotale).



La classe Etudiant a été créée. Elle ne possède que les **attributs** *NoteMath* et *NoteInfo*. Les **méthodes** de cette classe sont par exemple *MoyMath*, *MoyInfo*, *MoyTotale*.

Nous avons créé deux **objets** étudiants de la classe Etudiant (deux **instances** : Julien et Claudie).

1.3 L'héritage

Dans un **LOO** (**L**angage **O**rienté **O**bjet), il existe une particularité dans la façon d'organiser ses classes : l'héritage de propriétés. L'objectif est de construire de nouvelles classes en **réutilisant** des attributs et des méthodes de classes déjà existantes.

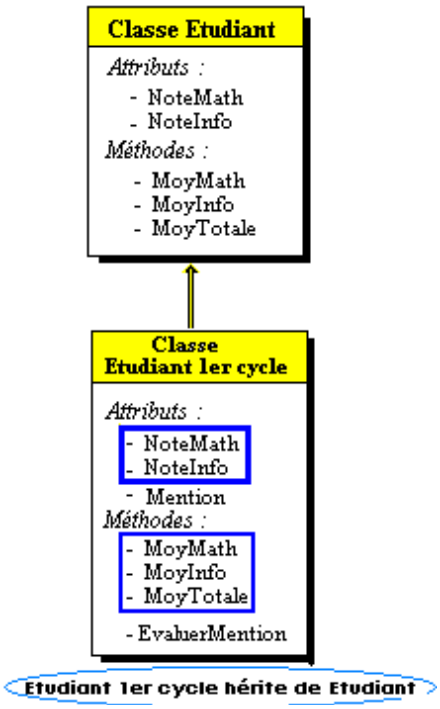
Héritage

C'est un mécanisme très puissant qui permet de décrire des structures génériques en transmettant depuis l'intérieur d'une même classe toutes les propriétés communes à toutes les " sous-classes " de cette classe. Par construction toutes les sous-classes d'une même classe possèdent toutes les attributs et les méthodes de leur classe parent.

Propriétés de l'héritage

- Les attributs et les méthodes peuvent être modifiés au niveau de la sous-classe qui hérite.
- Il peut y avoir des attributs et/ou des méthodes supplémentaires dans une sous-classe.
- Une classe A qui hérite d'une classe B dispose implicitement de tous les attributs et de toutes les méthodes définis dans la classe B.
- Les attributs et les méthodes définis dans A sont prioritaires par rapport aux attributs et aux méthodes de même nom définis dans

Exemple :
La classe " *Etudiant premier cycle*" héritant de la classe *Etudiant* précédemment définie.



Tout se passe comme si toute la classe *Etudiant* était recopiée dans la sous-classe *Etudiant premier cycle* (même si l'implémentation n'est pas faite ainsi).

La nouvelle classe dispose d'un attribut supplémentaire (Mention) et d'une méthode supplémentaire (EvaluerMention).

```
type Tmention=(Passable, Abien, Bien, Tbien);
Etudiant = class
    NoteMath : real;
    NoteInfo : real;
    procedure MoyMath(UneNote:real);
    procedure MoyInfo(UneNote:real);
    function MoyTotale : real ;
end;
 Etudiant1erCycle = class(Etudiant)
    Mention: Tmention ;
    function EvaluerMention: Tmention;
end;
```

Ceci est une implantation possible de la signature de la classe en :

Delphi

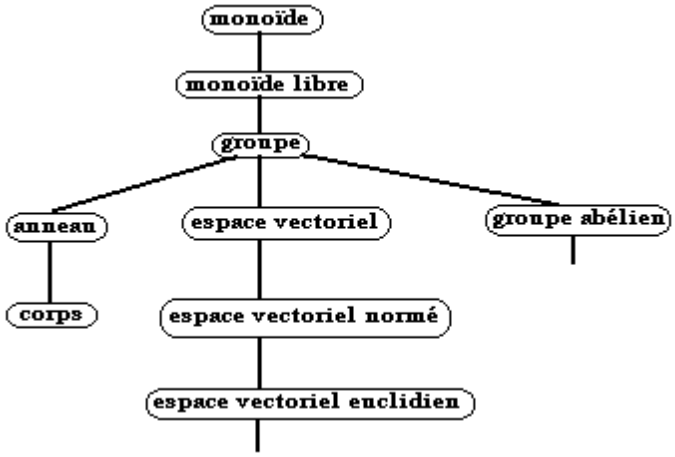
```
class Etudiant {
   public float NoteMath;
   public float NoteInfo;
   public void MoyMath(float UneNote){
   ...    }
   public void MoyInfo(float UneNote){
   ...    }
   public float MoyTotale(){
   ...    }
 }// fin classe Etudiant
public class Etudiant1erCycle extends Etudiant{
   public String Mention;
   public String EvaluerMention( ){
   ...    }
   public static void Main(String[ ] args){
   ...    }
}// fin
```

Ceci est une implantation possible de la signature de la classe en :

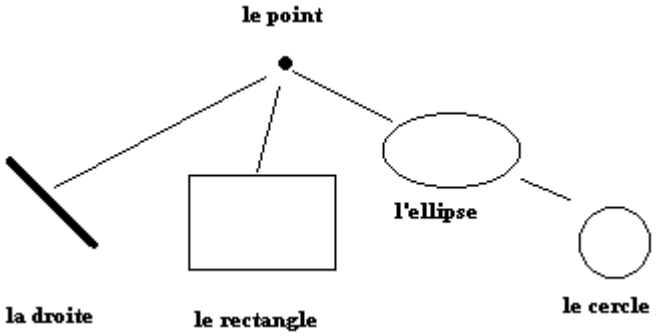
Java

Exemples d'héritage dans d'autres domaines :

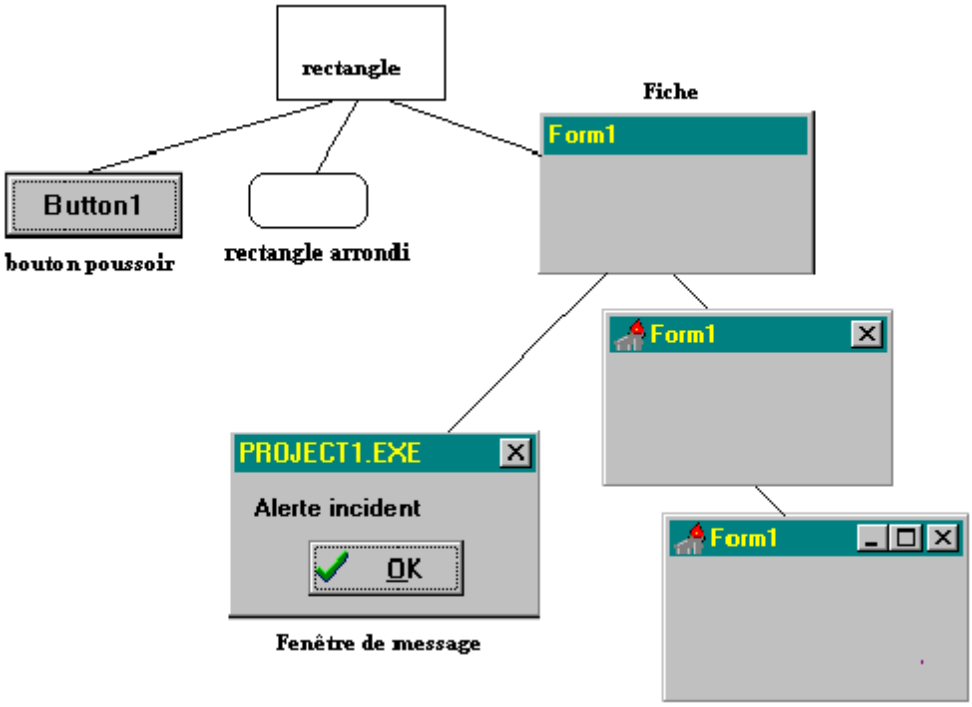
Héritage dans les structures mathématiques :



Héritage de figures de base en géométrie affine :



Héritage d'objets graphiques dans un système multi-fenêtré fictif :



2. Introduction à la conception orientée objet

L'attitude est ici résolument sous-tendue par un double souci : fournir des outils méthodologiques rationalisant l'effort de production du logiciel, sans que leur lourdeur rebute l'étudiant non professionnel et masque ainsi l'intérêt de leur utilisation. L'expérience d'enseignement de l'auteur avec des débutants a montré que si les étudiants sont appelés à développer sans outils méthodiques, ils pratiquent ce qu'appelle J.Arsac " la grande bidouille ". Mais dans le cas contraire, l'apprentissage détaillé de trop de méthodes strictes bien qu'efficaces (OOD, OMT, HOOD, UML,...)finit par ennuyer l'étudiant ou du moins par endormir son sens de l'intérêt. Dans ce dernier cas l'on retrouve " la grande bidouille " comme étape finale. Le chemin est donc étroit et il appartient à chaque enseignant de doser en fonction de l'auditoire l'utile et le superflu.

Nous utilisons ici un de ces dosages pour montrer à l'étudiant comment écrire des programmes avec des objets sans être un grand spécialiste. Une aide irremplaçable à cet égard nous sera fournie par l'environnement de développement visuel Delphi.

2.1 La méthode de conception OOD simplifiée

La méthode O.O.D (object oriented design) de G.Booch propose 5 *étapes* dans l'établissement d'une conception orientée objet. Ces étapes n'ont pas obligatoirement à être enchaînées dans l'ordre dans lequel nous les citons dans le paragraphe suivant. C'est cette souplesse qui nous a fait choisir la démarche de G.Booch, car cette méthode est fondamentalement incrémentale et n'impose **pas un cadre trop précis et trop rigide** dans son application. Cette démarche se révèle être utile pour un débutant et lui permettra de fabriquer en particulier des prototypes avec efficacité sans trop surcharger sa mémoire.

Identifier les objets et leurs attributs

On cherchera à identifier les objets du monde réel que l'on voudra réaliser.

Conseil : *Il faut identifier les propriétés caractéristiques de l'objet (par l'expérience, l'intuition,...).On pourra s'aider d'une description textuelle (en langage naturel) du problème. La lecture de cette prose aidera à déduire les bons candidats pour les noms à utiliser dans cette description ainsi que les propriétés des adjectifs et des autres qualifiants.*

Identifier les opération

On cherchera ensuite à identifier les actions que l'objet subit de la part de son environnement et qu'il provoque sur son environnement.

Conseil : *Les verbes utilisés dans la description informelle (textuelle) fournissent de bons indices pour l'identification des opérations. Si nécessaire, c'est à cette étape que l'on pourra définir les conditions d'ordonnancement temporel des opérations (les événements ayant lieu).*

Etablir la visibilité

L'objet étant maintenant identifié par ses caractéristiques et ses opérations, on définira ses relations avec les autres objets.

Conseil : *On établira quels objets le voient et quels objets sont vus par lui (les spécialistes disent alors qu'on insère l'objet dans la topologie du projet). Il faudra prendre bien soin de définir ce qui est visible ou non, quitte à y revenir plus tard si un choix s'est révélé ne pas être judicieux.*

Etablir l'interface

Dès que la visibilité est acquise, on définit l'interface précise de l'objet avec le monde extérieur.

Conseil : *Cette interface définit exactement quelles fonctionnalités sont accessibles et sous quelles formes. Cette étape doit pouvoir être décrite sous notation formelle; les TAD sont l'outil que nous utiliserons à cette étape de conception.*

Implémenter les objets

La dernière étape consiste à implanter les objets en écrivant le code.

Conseil : *Cette étape peut donner lieu à la création de nouvelles classes correspondant par exemple à des nécessités d'implantation. Le code en général correspond aux spécifications concrètes effectuées avec les TAD, ou à la traduction des algorithmes développés par la méthode structurée. Lors de cette étape, on identifiera éventuellement de nouveaux objets de plus bas niveau d'abstraction qui ne pouvaient pas être analysés en première lecture(ceci provoquant l'itération de la méthode à ces niveaux plus bas).*

Nous n'opposons pas cette méthode de conception à la méthode structurée modulaire. Nous la considérons plutôt comme complémentaire (en appliquant à des débutants une idée contenue dans la méthode HOOD). La méthode structurée modulaire sert à élaborer des algorithmes classiques comme des actions sur des données. La COO permet de définir le monde de l'environnement de façon modulaire. Nous réutiliserons les algorithmes construits dans des objets afin de montrer la complémentarité des deux visions.

2.2 Notation UML de classes et d'objets

La notion d'un langage de modélisation standard pour tout ce qui concerne les développements objets a vu le jour en 1997 il s'agit d'UML (Unified Modeling Language).

UML n'est pas une méthode, mais une notation graphique et un métamodèle.

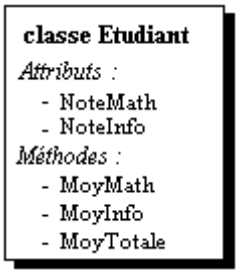
Nous allons fournir ici les principaux schémas d'UML permettant de décrire des démarches de conception objets simples. Le document de spécification de la version 1.4 d'UML par l'OMG (l'Object Management Group) représente 1400 pages de définitions sémantiques et de notations; il n'est donc pas question ici de développer l'ensemble de la notation UML (que d'ailleurs l'auteur ne possède pas lui-même).

Nous nous attacherons à détailler les diagrammes de base qui pourront être utilisés par la suite dans le reste du document.

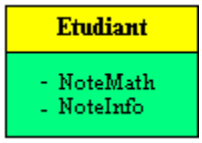
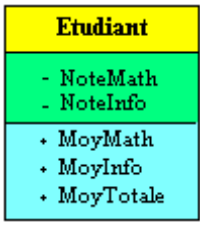
Nous nous limiterons aux notations relatives aux classes, aux objets, et à l'héritage.

SCHEMA UML DE CLASSE

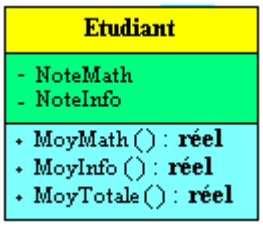
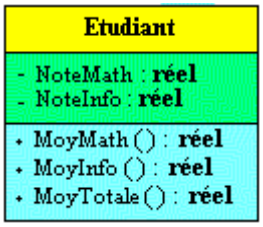
Notations UML possibles d'une classe :
Reprenons l'exemple précédent avec la classe étudiant :



trois notations UML possibles

Simplifiée	*Avec attributs*	*Attributs et méthodes*
		

Deux autres notations UML plus complète pour la même classe

Attributs et méthodes typées	*Attributs typés et méthodes typées*
	

Une classe abstraite est notée :

VISIBILITE DES ATTRIBUTS ET DES METHODES

Notation préfixée UML pour trois niveaux de visibilité (+, - , # , $) :

Pour les attributs :

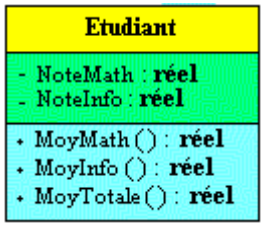
public	*privé*	*protégé*
+ Attribut1 : DeType1	- Attribut2 : DeType2	# Attribut3 : DeType3

Pour les méthodes :

public	*privé*	*protégé*
+ Methode1 ( ): DeType1	- Methode2 ( ) : DeType2	# Methode3 ( ) : DeType3

Méthode de classe
$ Methode4 ( ) : DeType4

Explicitation dans la classe Etudiant :



- Dans la classe **étudiant** les deux attributs **NoteMath** et **NoteInfo** sont de type réel et sont **privés** (préfixe -).
- Les trois méthodes de calcul de moyennes sont **publiques** (préfixe +).

SCHEMA UML D'UN OBJET

Notations UML pour deux objets étudiants instanciés à partir de la classe Etudiant :

Schéma UML simplifié :

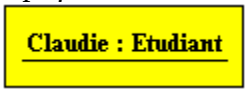
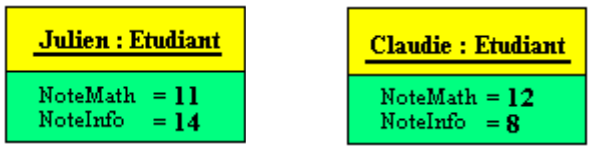
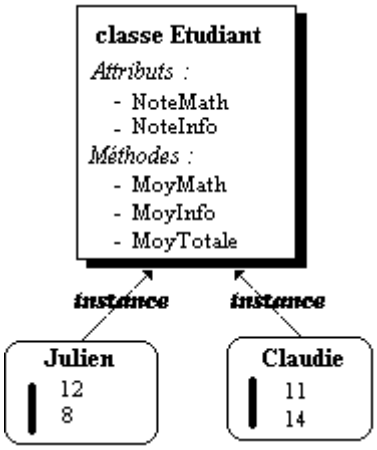


Schéma UML avec valeur des attributs :

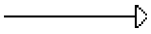


Ces notations correspondent à l'exemple ci-dessous :

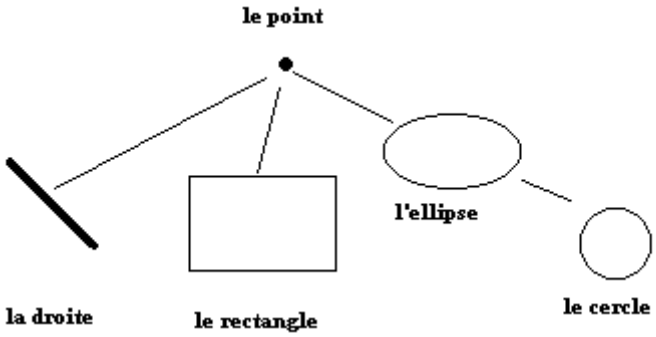


SCHEMA UML DE L'HERITAGE

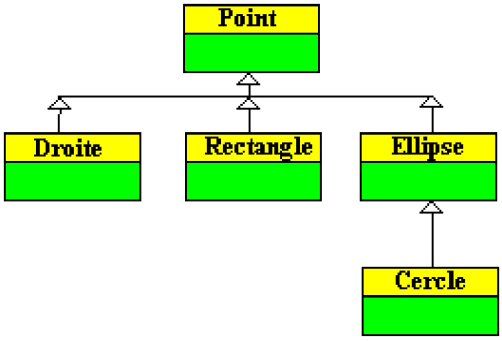
Notation UML de l'héritage :



Soit pour l'exemple de hiérarchie de classes fictives ci-dessous :

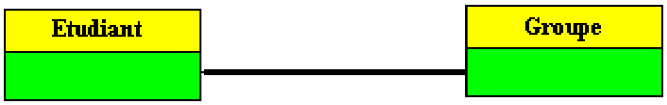


La notation UML suivante :

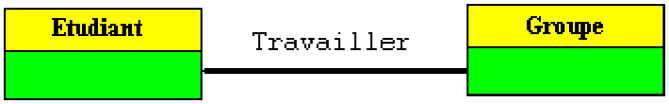


SCHEMA UML DES ASSOCIATIONS

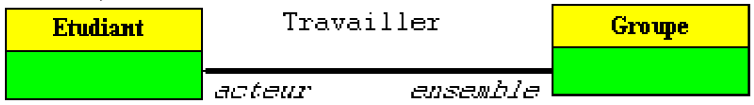
Une association binaire (ou plus généralement n-aire), représente un lien conceptuel entre deux classes. Par exemple un étudiant travaille dans un groupe (association entre la classe **Etudiant** et la classe **Groupe**).



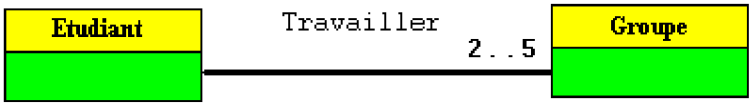
Une association peut être dénotée par une expression appelée nom d'association (nommé **Travailler** ci-dessous) :



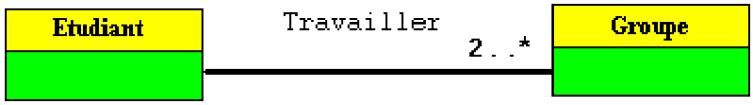
Chaque association possède donc deux extrémités appelées aussi rôles, il est possible de nommer les extrémités (nom de rôles, ci-dessous un étudiant est un acteur travaillant dans un groupe qui est un ensemble) :



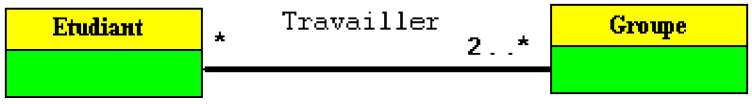
Une association peut posséder une multiplicité qui représente sous forme d'un intervalle de nombres entiers a..b, le nombre d'objets de la classe d'arrivée qui peut être mis en association avec un objet de la classe de départ. Supposons qu'un étudiant doive s'inscrire à au moins 2 groupes de travail et au plus à 5 groupes, nous aurons le schéma UML suivant :



La présence d'une étoile dans la multiplicité indiquant un nombre quelquonque (par exemple un étudiant peut s'inscrire à au moins 2 groupes sans limite supérieure):



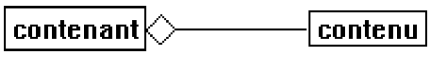
Par exemple pour dénoter en UML le fait qu'un nombre quelconque d'étudiants doit travailler dans au moins deux groupes nous écrirons:



UNE ASSOCIATION PARTICULIERE : L'AGREGATION

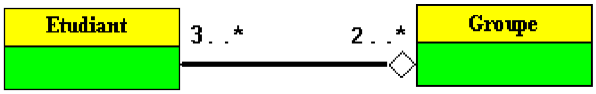
Une agrégation est une association correspondant à une relation qui lorsqu'elle est lue dans un sens signifie "est une partie de" et lorsqu'elle est lue dans l'autre sens elle signifie "est composé de". UML disposant de plusieurs raffinements possibles nous utiliserons l'agrégation comme composition par valeur en ce sens que la construction du tout implique la construction automatique de toutes les parties et que la destruction du tout entraîne en cascade la destruction de chacune de ses parties.

Notation UML de l'agrégation



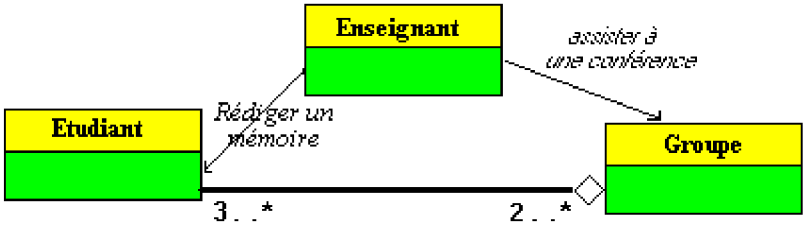
Exemple :

un groupe contient au moins 3 étudiants et chaque étudiant doit s'inscrire à au moins 2 groupes :



NOTATION UML DES MESSAGES

Un message envoyé par une classe à une autre classe est représenté par une flèche vers la classe à qui s'adresse le message, le nommage de la flèche indique le message à exécuter :



2.3 Attitudes et outils méthodologiques

Afin d'utiliser une méthodologie pratique et rationnelle, nous énumérons au lecteur les outils que nous utilisons selon les besoins, dans le processus d'écriture d'un logiciel.

En tout premier la notion de module : C'est la décomposition d'un logiciel en sous-ensembles que l'on peut changer comme des pièces d'un patchwork.
La notion de cycle de vie du logiciel : Développer un logiciel ce n'est pas seulement écrire du Pascal, de l'Ada etc...
Utiliser des TAD : Un type abstrait de données correspond très exactement à l'interface d'un module. Il renforce la méthodologie modulaire.
La programmation structurée par machines abstraites : On se sert d'une méthode de conception descendante et modulaire des algorithmes utiles pour certaines actions dans le logiciel.
La conception et la programmation orientées objet : On utilise une version simplifiée de la COO de G.Booch pour définir les classes et leurs relations en attendant une simplification pédagogique d'UML.

5.2 Programmez objet avec Delphi

Plan du chapitre:

Introduction

Delphi™ de Borland est un RAD visuel fondé sur une extension orientée objet, visuelle et événementielle de Pascal, il fonctionne depuis 2004 sous le système Windows toutes versions, sous Linux et sous l'architecture .Net.

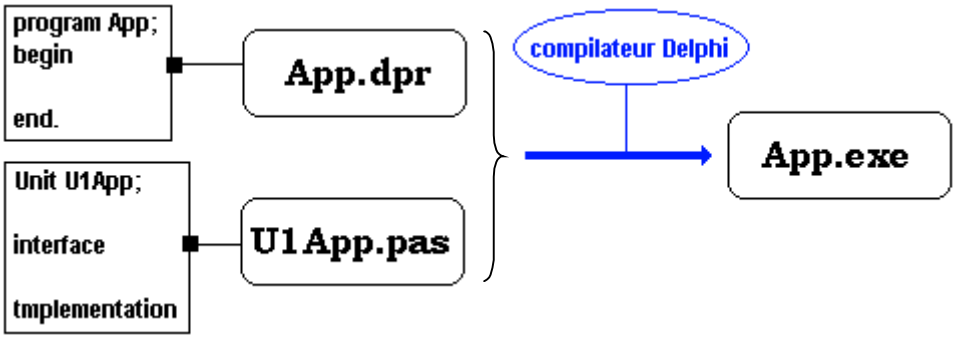
Pascal est le langage utilisé pour l'initiation dans de très nombreux établissements d'enseignement européens. Le RAD Delphi est un prolongement intéressant de ce langage. Nous allons explorer certains aspects les plus importants et significatifs du pascal objet de Delphi. Le langage pascal de base étant supposé connu par le lecteur, nous souhaitons utiliser ce RAD visuel en réutilisant du savoir faire pascal tout en y rajoutant les possibilités objet offertes par Delphi.

1. Description minimale de Delphi ...

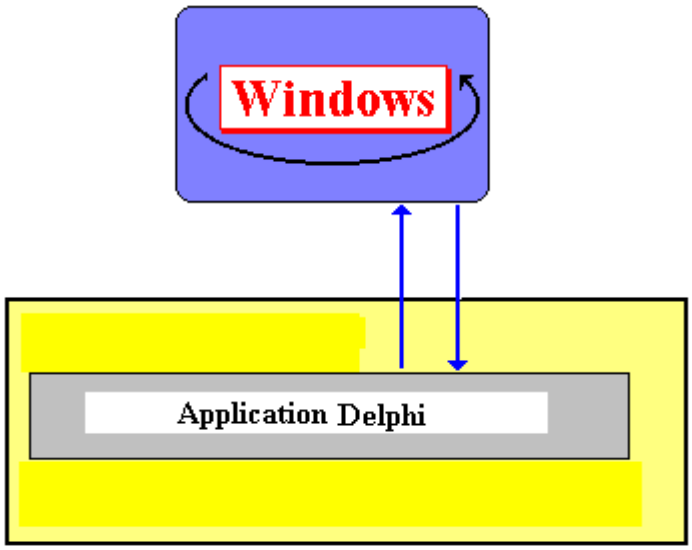
La version utilisée pour écrire les exemples est la dernière disponible sur Windows, mais tous les exemples sont écrits avec les fonctionnalités générales de Delphi ce qui permet de les compiler sur n'importe quelle version de Delphi depuis la version 5.

1.1 L'application Delphi

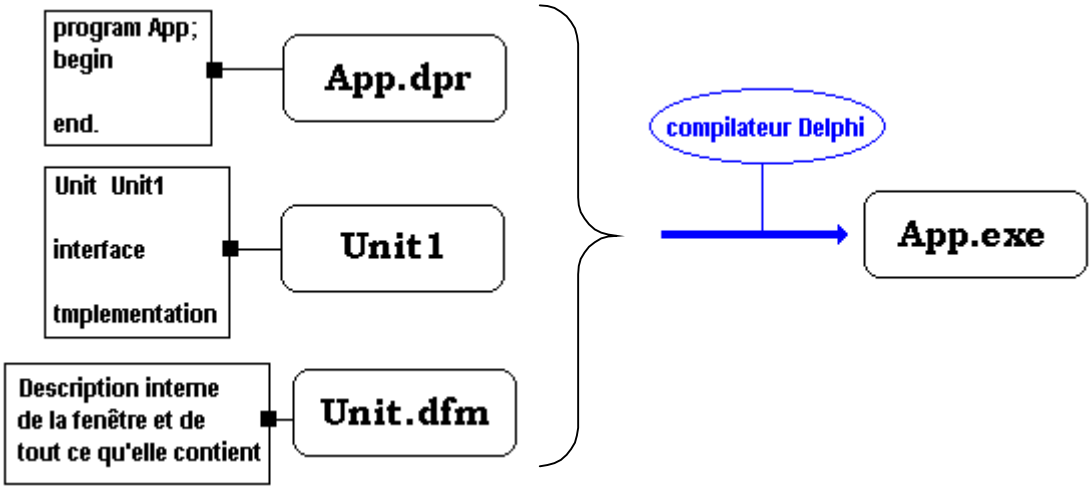
Une application console (non fenêtrée) Delphi se compose d'un projet "xxx.**dpr**" et d'au minimum un fichier d'Unit "xxx.**pas**" pour le code source. Lors de la compilation d'un projet Delphi engendre un code "xxx.**exe**" directement exécutable.

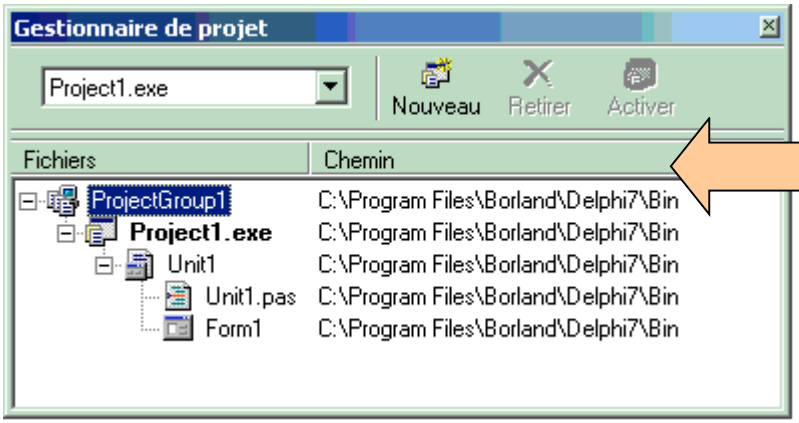


Tous les projets Delphi s'exécutent sous windows sans aucune Dll supplémentaire autre que celles que vous programmerez vous-même :



Une application fenêtrée Delphi se compose d'un projet "xxx.**dpr**" et d'au minimum deux fichiers de fiche "xxx.**dfm**" pour la description et "xxx.**pas**" et d'Unit pour le code source.





Ci-contre un projet minimal Delphi comportant une fiche principale.

Le projet se dénomme **Project1.dpr**

La fiche principale **Form1** et le code source de l'application sont rangés dans la Unit **Unit1.pas**.

La description de la fiche **Form1** et de ce qu'elle contient se trouve dans un fichier nommé **Unit1.dfm**.

A quoi sert une fiche ?

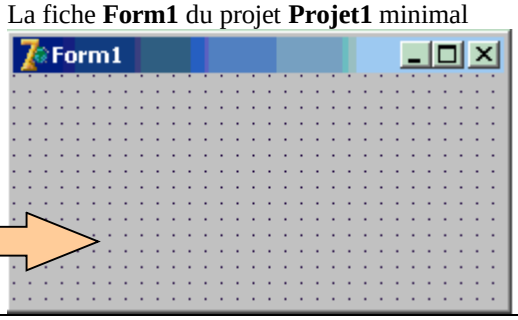
Les systèmes d'exploitation actuels sont dit fenêtrés au sens où ils fournissent un mode de communication avec l'homme fondé sur la notion de fenêtre, Windows en est un exemple.

La première action à entreprendre lors du développement d'une application **Interface Homme Machine** (IHM) avec un langage de programmation, est la création de l'interface de l'application et ensuite les interactions avec l'utilisateur. Le langage de programmation doit donc permettre de construire au moins une fenêtre

En Delphi pour créer une IHM, il faut utiliser des **fiches** (ou fenêtres) et des **contrôles**.

Ces fiches sont des objets au sens informatique de la POO, mais elles possèdent une **représentation visuelle**.

La fiche **Form1** du projet **Projet1** minimal

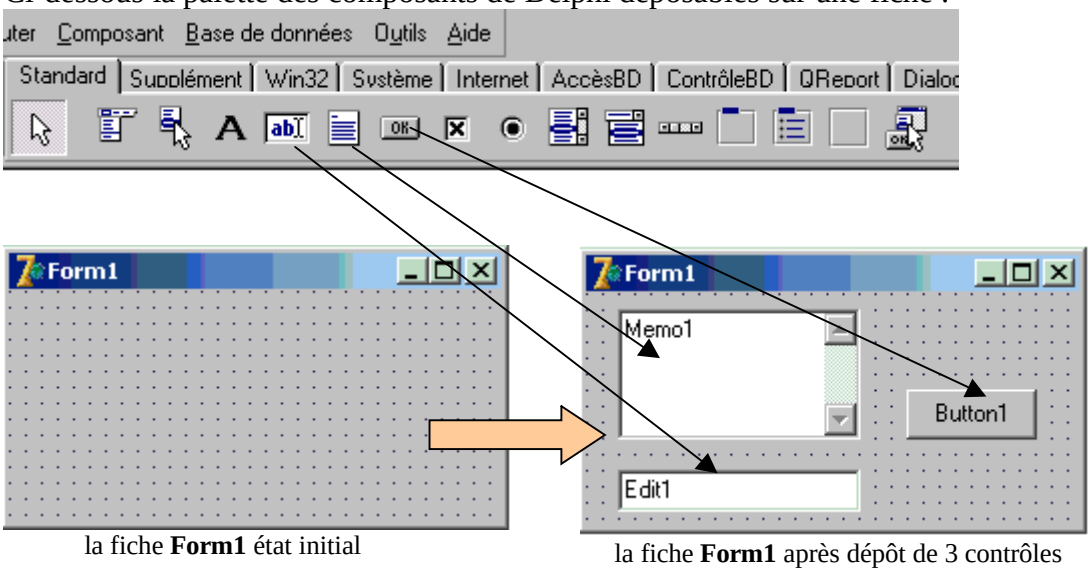


1.2 Les objets de fiches et les objets contrôles

Chaque fiche est en fait un objet instancié à partir de la classe interne des **TForm** de Delphi. Cette classe possède des propriétés(attributs) qui décrivent l'apparence de la fiche, des méthodes qui décrivent le comportement de la fiche et enfin des gestionnaires d'événements (pointeurs de méthodes) qui permettent de programmer la réaction de la fiche aux événements auxquels elles est sensible.

Sur une fiche vous déposez des **contrôles** qui sont eux aussi d'autres classes d'objets visuels, mais qui sont contenus dans la fiche.

Ci-dessous la palette des composants de Delphi déposables sur une fiche :



la fiche **Form1** état initial

la fiche **Form1** après dépôt de 3 contrôles

Que se passe-t-il lors du dépôt des contrôles ?

code dans la fiche **Form1** avant dépôt	code dans la fiche **Form1** après dépôt
unit Unit1; **interface** **uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs; **type** TForm1 = **class**(TForm) **private** *{ Déclarations privées }* **public** *{ Déclarations publiques }* **end**; **var** Form1: TForm1; **implementation** *{$R *.DFM}* **end**.	**unit** Unit1; **interface** **uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs; **type** TForm1 = **class**(TForm) Memo1: TMemo; Button1: TButton; Edit1: TEdit; **private** *{ Déclarations privées }* **public** *{ Déclarations publiques }* **end**; **var** Form1: TForm1; **implementation** *{$R *.DFM}* **end**.

Il existe dans Delphi une notion de classe conteneur, la fiche (classe **TForm**) en est le principal représentant. Delphi est un générateur automatique de programme source Pascal objet et dès l'ouverture du projet, il définit une classe conteneur **TForm1** qui hérite de la classe **TForm** et qui au départ ne contient rien :

```
TForm1 = class(TForm)
 private
   { Déclarations privées }
 public
   { Déclarations publiques }
 end;
```

Lorsque nous déposons les 3 contrôles Button1, Edit1, Memo1, ce sont des champs objets qui sont automatiquement ajoutés par Delphi dans le code source de la classe :

```
TForm1 = class(TForm)
   Memo1: T Memo;
   Button1: TButton;
   Edit1: TEdit;
 private
   { Déclarations privées }
 public
   { Déclarations publiques }
 end;
```

Donc l'environnement Delphi, n'est pas seulement un langage de programmation, mais aussi un générateur de programme à partir de dépôt de composants visuels, c'est la fonction d'un système RAD (Rapid Application Developpement).

Pour une utilisation de Delphi, nous renvoyons le lecteur à la documentation du constructeur, nous attachons par la suite à faire ressortir et à utiliser les éléments de Delphi qui concernent la programmation modulaire et la programmation objet

2. Les modules dans Delphi

Nous avons déjà vu au chapitre sur les TAD (types abstraits de données) que le Pascal de Delphi permettait de traduire sous une première forme la notion de type abstrait : la **Unit**. Une **Unit** Delphi permet aussi de représenter la notion de module.

- Une **Unit** est une unité compilable séparément de tout programme et stockable en bibliothèque.

- Une **Unit** comporte une partie " public " et une partie " privé ". Elle implante donc l'idée de module et étend la notion de bloc (procédure ou fonction) en Pascal. Elle peut contenir des descriptions de code simple ou de classe.

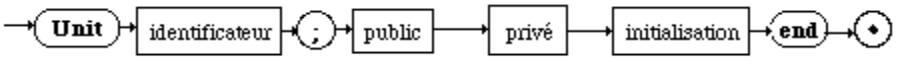
Chaque unité **Unit** est stockée dans un fichier **xxx.pas** distinct et compilée séparément ; les unités compilées (les fichiers xxx.**DCU**) sont liées pour créer une application.

Les unités en Delphi permettent aussi :

- De diviser de grands programmes en modules qui peuvent être modifiés séparément.
- De créer des bibliothèques qui peuvent être partagées par plusieurs programmes.
- De distribuer des bibliothèques à d'autres développeurs sans leur donner le code source.

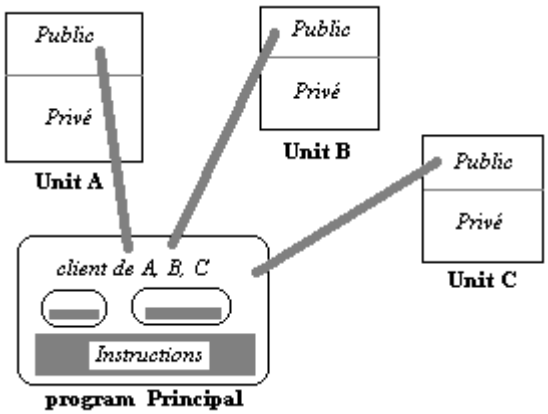
Pour générer un code éxécutable à partir d'un projet comportant plusieurs unités, le compilateur Delphi doit disposer, pour chaque unité, soit du fichier source xxx.**PAS**, soit du fichier XXX.**DCU** résultant d'une compilation antérieure. Cette dernière possibilité permet de fournir des unités compilées à d'autres personnes sans leur fournir le code source.

Syntaxe d'une unité :



Unit Truc;
<partie public >
<partie privée >
<initialisation >
end.

Utilisation d'une unité :



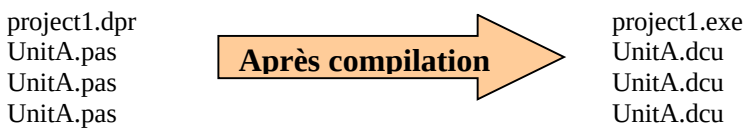
Le programme principal se nomme le projet, il est rangé dans le fichier "xxx.**dpr".**

Ici le programme principal utilise 3 Unit : UnitA , UnitB et UnitC

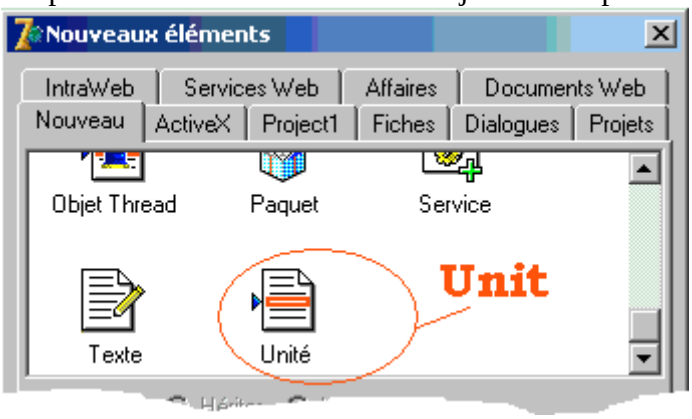
Squelette du code associé au schéma précédent :

unit UnitA; **interface** **implementation** **end**.	**unit** UnitB; **interface** **implementation** **end**.	**unit** UnitC; **interface** **implementation** **end**.	**Program** project1; **Uses** UnitA, UnitB, UnitC; **Begin** **end**.

Fichiers disque (sources et compilés) associés au code précédent :



Pour ajouter au projet une nouvelle **Unité** :

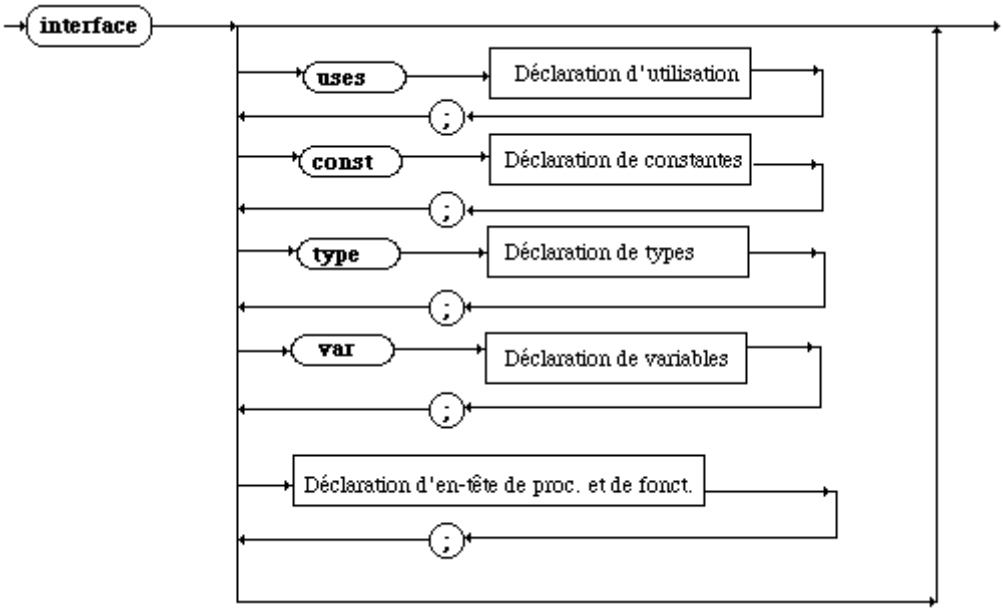
On peut utiliser l'environnement d'ajout de Delphi :	On peut écrire soi-même un fichier texte contenant la unit :
 qui crée par exemple un fichier Unit2.pas contenant le squelette de la Unit2 et rajoute automatiquement cette **unit** au programme principal.	**unit** Unit2; **interface** **implementation** **end**. Et rajouter soi-même au texte source du programme principal la **unit** : **Program** project1; **Uses** Unit2 ; **Begin** **end**.

2.1 Partie " public " d'une UNIT : " Interface "

Cette partie d'une **unit**, correspond exactement à la partie publique du module représenté par la UNIT. Cette partie décrit les en-têtes des procédures et des fonctions publiques et utilisables par les clients. Les clients peuvent être soit d'autres procédures Delphi, des programmes Delphi ou d'autres Unit.

La clause **Uses** XXX dans un programme Delphi, permet d'indiquer la référence à la **Unit** XXX et autorise l'accès aux procédures et fonctions publiques de l'interface dans tout le programme.

Syntaxe de l'interface :

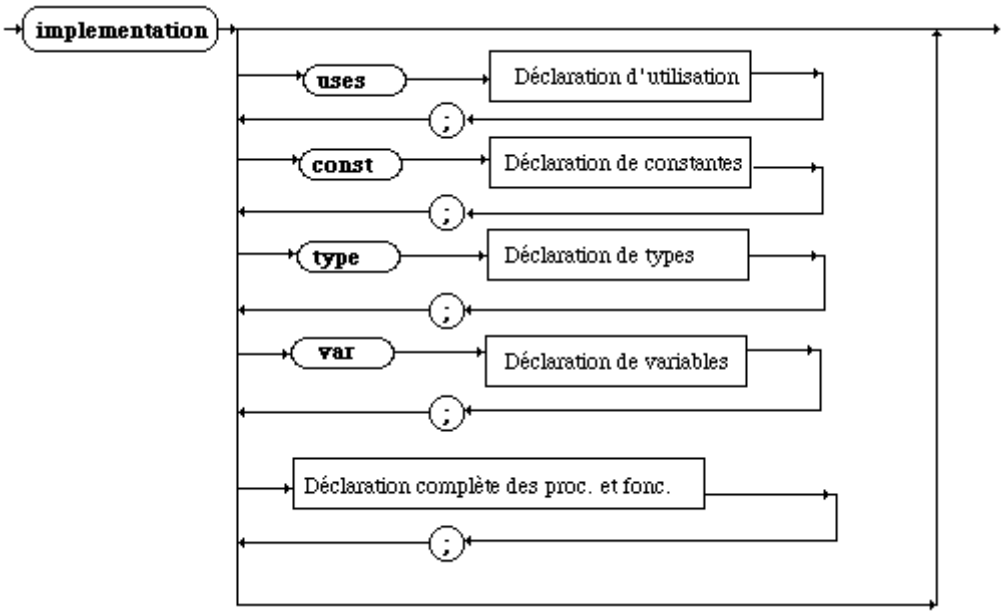


2.2 Partie " privée " d'une UNIT : " Implementation "

Correspond à la partie privée du module représenté par la UNIT. Cette partie intimement liée à l'interface, contient le code interne du module.

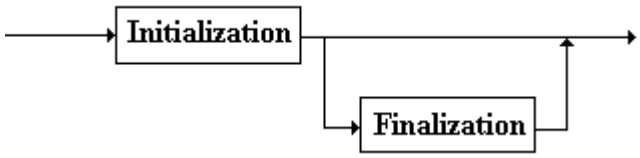
Elle contient deux sortes d'éléments : les déclarations complètes des procédures et des fonctions privées ainsi que les structures de données privées. Elle contient aussi les déclarations complètes des fonctions et procédures publiques dont les en-têtes sont présentes dans l'interface.

Syntaxe de l'implementation :



2.3 Initialisation et finalisation d'une UNIT

Syntaxe de l'initialisation :

La partie initialisation d'une unité en Delphi comporte deux sous parties **Initialization** et **Finalization** la seconde étant optionnelle:	

Initialization	Finalization
Il est possible d'initialiser des variables et d'exécuter des instructions au lancement de l'UNIT. Elles correspondent à des instructions classiques Pascal sur des données publiques ou privées de la Unit (initialisation de tableaux, mise à zéro de divers indicateurs, chargement de fichiers etc...).	Une fois que le code d'initialisation d'une unité a commencé à s'exécuter, la section de finalisation correspondante si elle existe, s'exécute obligatoirement à l'arrêt de l'application (libération de mémoire, de fichiers, récupération d'incidents etc...).

Exemple de programme console modulaire

Unit Delphi Uratio du TAD rationnel	Programme principal Delphi utilisant Uratio

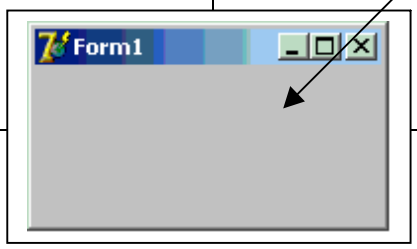
```
unit Uratio;
{unité de rationnels spécification classique ZxZ/R}
interface
  type
    rationnel =
    record
      num: integer;
      denom: integer
    end;
    procedure reduire (var r: rationnel);
    procedure addratio (a, b: rationnel; var s:
rationnel);
    procedure divratio (a, b: rationnel; var s:
rationnel);
    procedure mulratio (a, b: rationnel; var s:
rationnel);
    procedure affectQ(var s: rationnel; b: rationnel);
    procedure opposeQ(x:rationnel;var s:rationnel);

implementation
    procedure reduire ….
    procedure addratio ….
    Procedure divratio ….
    procedure mulratio ….
    procedure affectQ….
    procedure opposeQ….
end.
```

```
program essaiRatio;
  {programme de test de la unit Uratio }

{$APPTYPE CONSOLE}

  uses SysUtils , Uratio;
  var
    r1, r2, r3, r4, r5: rationnel;

  begin
    r1.num :=18;
    r1.denom := 15;
    r2.num := 7;
    r2.denom := 12;
    addratio(r1, r2, r3);
    writeln('18/15 + 7/12 = ', r3.num, '/', r3.denom);
    mulratio(r1, r2, r4);
    writeln('18/15 * 7/12 = ', r4.num, '/', r4.denom);
    divratio(r1, r2, r5);
    writeln('18/15 / 7/12 = ', r5.num, '/', r5.denom);
    r1.num := 72;
    r1.denom := 60;
    affectQ(r3,r1);
    reduire(r1);
    writeln('72/60 = ', r1.num, '/', r1.denom);
    writeln('avant réduction ', r3.num, '/', r3.denom);
  end.
```

Exemple fiche : le programme lançant une fiche vierge

```
unit Unit1;
    interface
    uses
      Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs;
    type
      TForm1 = class(TForm)
      private
        { Déclarations privées }
      public
        { Déclarations publiques }
      end;
    var
      Form1: TForm1;
    implementation
    {$R *.DFM}
end.
```

```
program Project1;

  uses  Forms,
    Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

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



3. Delphi et la POO ...

Notre objectif est d'apprendre comment Delphi implante les notions contenues dans la P.O.O. et d'utiliser ces outils dans nos programmes.

Delphi est un langage orienté objet, dans ce domaine il possède des fonctionnalités équivalentes à C++ et java, avec sa version .Net, il possède les fonctionnalités équivalentes à celles de C# de microsoft.

3.1 Les éléments de base

Sachons que dans Delphi tout est objet, des contrôles de la **VCL** (Visual Component Library commportant plus de 600 classes dont environ 75 sont visuelles, les autres étant non visuelles ou concernant des objets de service).

Delphi possède de par son fondement sur Object Pascal tous les types prédéfinis du pascal et les constructeurs de types (supposés connus du lecteur), plus des types prédéfinis étendus spécifique à Delphi. Ce qui signifie qu'il est tout à fait possible de réutiliser en Delphi sans effort de conversion ni d'adaptation tout programme pascal ISO ou UCSD déjà écrit.

Les types **integer**, **real** et **string** sont des types génériques c'est à dire qu'ils s'adaptent à tous les types d'**entiers**, de **réels** et de **chaînes** de Delphi.

Par exemple les entiers de base de Delphi suivants :

Shortint	-128..127	8 bits signé
Smallint	-32768..32767	16 bits signé
Longint	-2147483648..2147483647	32 bits signé
Byte	0..255	8 bits non signé
Word	0..65535	16 bits non signé
Longword	0..4294967295	32 bits non signé

peuvent être affectés à une variable x : **integer** car c'est un type générique.

var x : **integer** ;
a : Longint; b : Longword; c: Byte; d: Word; e: Smallint; f: Shortint;
x := a ; x := b ; x := c ; x := d ; x := e ; x := f ;

Delphi dispose d'un type générique **variant polymorphe** sur les types de données prédéfinis de Delphi. Un **variant** peut s'adapter ou se changer en pratiquement n'importe quel type de Delphi	**Var** x : variant; **begin** x:='abcdefgh'; *// x est une string* x:=123.45; *// x est un real* x:=true; *// x est un booléen* x:=5876 *// x est un integer* etc... 

Les passages des paramètres s'effectuent principalement selon les deux modes classiques du pascal : le passage par valeur (pas de mot clé), le passage par référence (mot clé **Var**). Par défaut si rien n'est spécifié le passage est par valeur.

Améliorations de ces passages de paramètres :

- Delphi autorise un mode de passage dit des paramètres constants permettant une sécurité accrue au passage par valeur (mot clé **Const**).

- Delphi dispose d'une autre amélioration concernant le passage par référence : le passage par sortie (mot clé **out**), qui indique simplement à la procédure où placer la valeur en sortie sans spécifier de valeur en entrée, qui si elle existe n'est pas utilisée.

Exemple d'utilisation des variant dans un tri récursif sur un tableau de variant :

(les paramètres sont tous passés par référence)

```
const n=100;
type Tableau =array[0..n] of variant;

procedure quicksort (var G,D:integer; var
tabl:Tableau);
var i , j : Integer;
   x , w : variant;
begin
 i := G;
 j := D;
 x := tabl[(i + j) div 2];
 repeat
   While tabl[i] < x do  i := i + 1;
   While tabl[j] x do  j := j - 1;
   If i <= j Then
   begin
      w := tabl[i];
      tabl[i] := tabl[j];
      tabl[j] := w;
      i := i + 1;
      j := j - 1
   End;
 Until i j;
 If G < j Then quicksort(G, j, tabl);
 If D i Then quicksort(i, D, tabl)
End;
```

La procédure générique quicksort permet de trier un tableau de **variant**.

Grâce à son pouvoir polymorphe un **variant** peut devenir au choix soit:

- Entier long
- Entier court
- Un réel simple ou double
- Une chaîne de caractères.

Ce qui revient à dire que la procédure générique quicksort permet de trier un tableau de :

- Entiers longs
- Entiers courts
- Réels simples ou doubles
- Chaînes de caractères.

Dans le cas où le type **variant** n'existe pas, il faut au moins 3 procédures différentes quicksortXXX qui diffèrent par le type de leur paramètres, mais avec le même code :

procedure quicksortEntier (sur un tableau d'**integer**)
procedure quicksortReel (sur un tableau de **real**)
procedure quicksortString (sur un tableau de **string**)

Remarque :
D'autres modes de passage des paramètres sont possibles en Delphi, nous n'en parlons pas ici.

3.2 Fonctionnalités du pascal objet de Delphi

Delphi est un langage à structure de blocs complet. Sa gestion est identique à celle du langage pascal.

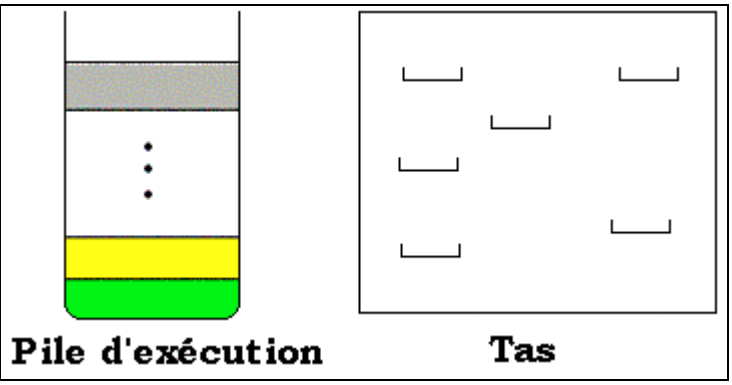
- La visibilité est identique à celle qui est inhérente à tout langage de bloc Algol-like : variables globales, variables locales, masquage des identificateurs.

- Les entités gérées dynamiquement le sont selon un modèle classique de **tas** et de **pile d'exécution**.

- Les entités gérées d'une façon permanente (données globales)sont persistantes durant toute la durée de l'exécution du programme, elles existent dans le **segment de données** alloué à l'application.

Dans un langage à structure de bloc comme Delphi, la mémoire centrale contient deux entités fondamentales : le **tas** et de **pile d'exécution**.

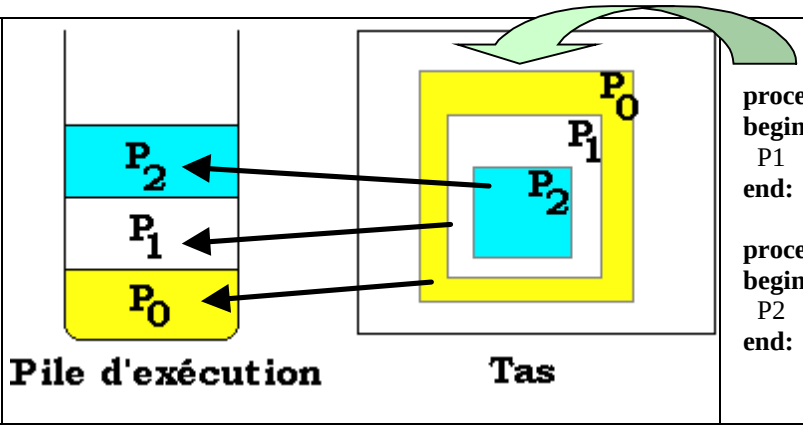
- Le **tas** est une structure de données déjà étudiée (arbre parfait partiellement ordonné géré dans un tableau).
- La **pile d'exécution** est une pile LIFO.



La pile d'exécution de Delphi a une taille paramétrable par le programmeur (maximum=2 147 483 647 octets), elle permet toutes les récursivités utiles.

Ci-dessous l'illustration du fonctionnement du **tas** et de la **pile** dans le cas de l'appel d'une procédure P0 qui appelle une procédure P1 qui appelle elle-même une procédure P2:

- Le **tas** contient les structures dynamiques et les codes.
- La **pile d'exécution** contient les contextes des procédures appelées.



procedure P0;
begin
  P1
end:

procedure P1;
begin
  P2
end:

Delphi est un langage acceptant la récursisvité

La récursivité d'une procédure ou d'une fonction est la capacité que cette fonction/procédure a de s'appeler elle-même.

Soit par exemple la somme des n premiers entiers définie par la suite récurrente S_n :

$$S_n = S_{n-1} + n$$
$$S_0 = 0$$

Ci-dessous une fonction de calcul récursif de la somme des **n** premiers entiers :

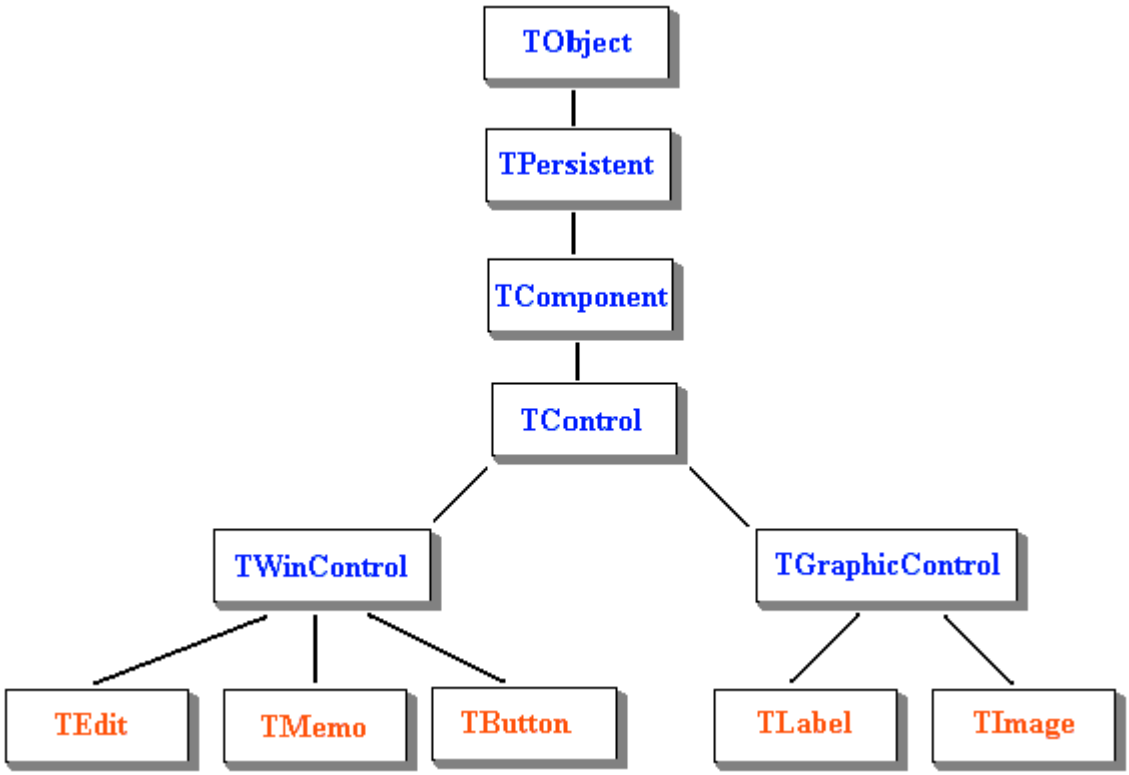
```
Function S(n : integer) : integer;
begin
 If n = 0 Then
    result:= 0
 Else
    result := n + S(n - 1)
End;
```

Delphi affiche un message d'erreur de pile pleine sur l'appel **S(129937)**, donc une profondeur de 129936 appels recursifs a été atteinte sur ce programme avec **le paramètrage standard fournit de 1 Mo comme taille maximum de la pile**.

3.3 Les classes

La notion de *classe* est essentielle dans Delphi en mode application visuelle. Les classes sont déclarées comme des types et contiennent des champs, des méthodes et des propriétés. Il existe une classe d'objet primitive appelée **TObject** , qui est l'ancêtre de toutes les autres classes de Delphi.

Voici un extrait de la hiérarchie des classes visuelles VCL (Visual Component Library) dans Delphi :



Comment déclarer une classe en Delphi

Un type classe doit être déclaré et nommé avant de pouvoir être instancié. Une classe est considéré par Delphi comme un nouveau type et doit donc être déclaré là où en Pascal l'on construit les nouveaux types : au paragraphe des déclarations à l'alinéa de déclaration des types.

Les types classes peuvent être déclarés pratiquement partout où le mot clé type est autorisé à l'exception suivante : ils ne doivent pas être déclarés à l'intérieur d'une procédure ou d'une fonction.

Les deux déclarations ci-dessous sont équivalentes :

```
Type
 ma_classe = class
   {déclarations de champs }
   {spécification méthodes }
   {spécification propriétés }
 end;
```

```
Type
 ma_classe = class(Tobject)
   {déclarations de champs }
   {spécification méthodes }
   {spécification propriétés }
 end;
```

Méta-classe

Delphi autorise la construction de méta-classes. Une Méta-classe est un générateur de classe. En Delphi les méta-classes sont utilisées afin de pouvoir passer des paramètres dont la valeur est une classe dans des procédures ou des fonction.

Les méta-classes en Delphi sont représentées par une référence de classe

Une méta-classe	Une variable de méta-classe
Type TMetaWinClasse = **class of** TWinControl;	**var** x : TMetaWinClasse;

La variable x de classe TMetaWinClasse, peut contenir une référence sur n'importe quelle classe de TWinControl : x :=Tmemo; x :=TEdit; x :=TButton; …

Exemple d'utilisation de la notion de méta-classe pour tester le type réel du paramètre effectif lors de l'appel de TwinEssai.	**procedure** TwinEssai (UnWinControl : TmetaWinClasse); **begin** **if** UnWinControl =TEdit **then …** **else if** UnWinControl=TMemo **then …** ....etc **end**;
A comparer avec l'utilisation de l'opérateur is sur le même problème	**procedure** TwinEssai (UnWinControl : TWinControl); **begin** **if** UnWinControl **is** TEdit **then …** **else if** UnWinContro **is** TMemo **then …** ....etc **end**;

En passant à un constructeur un paramètre de méta-classe on peut alors construire des objets dont la classe ne sera connue que lors de l'exécution.

Où déclarer une classe en Delphi ?

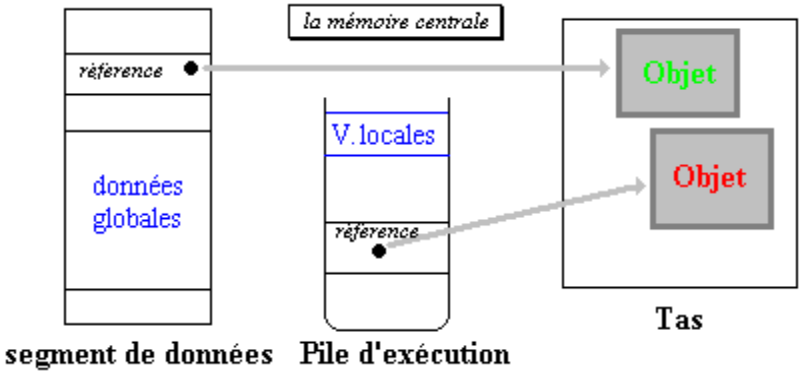
- Les classes sont déclarées dans des **unit**.
- Toutes les classes déclarées dans la même **unit** sont **amies**, c'est à dire que chacune d'elle a accès aux membres **privés** de toutes les autres classes declarées dans cette **unit**.
- Si l'on souhaite avoir **n** classes **non amies**, il faut les déclarer chacune dans une **unit** séparée.

3.4 Le modèle objet de Delphi

Le modèle physique choisi pour Delphi est celui de la référence :

- Chaque objet est caractérisé par un couple (référence, bloc de données).
- Si la variable de référence est locale à une procédure,elle est alors située physiquement dans la **pile d'exécution** avec les autres variables locales.
- Si la variable de référence est globale, elle est située physiquement dans le **segment de données** (permanent durant toutes la durée de l'exécution). Dans les deux cas, le bloc de données (l'objet effectif) est alloué dans le **tas**.

Ci-dessous une carte mémoire fictive d'un processus (une application Delphi classique à un seul thread):



La simplicité du modèle (semblable aux variables dynamiques ou pointeurs du pascal) permet de dire qu'en Delphi les pointeurs sont sous-jacents mais entièrement encapsulés.

Ce modèle impose une contrainte d'écriture en découplant la déclaration d'objet et sa création.

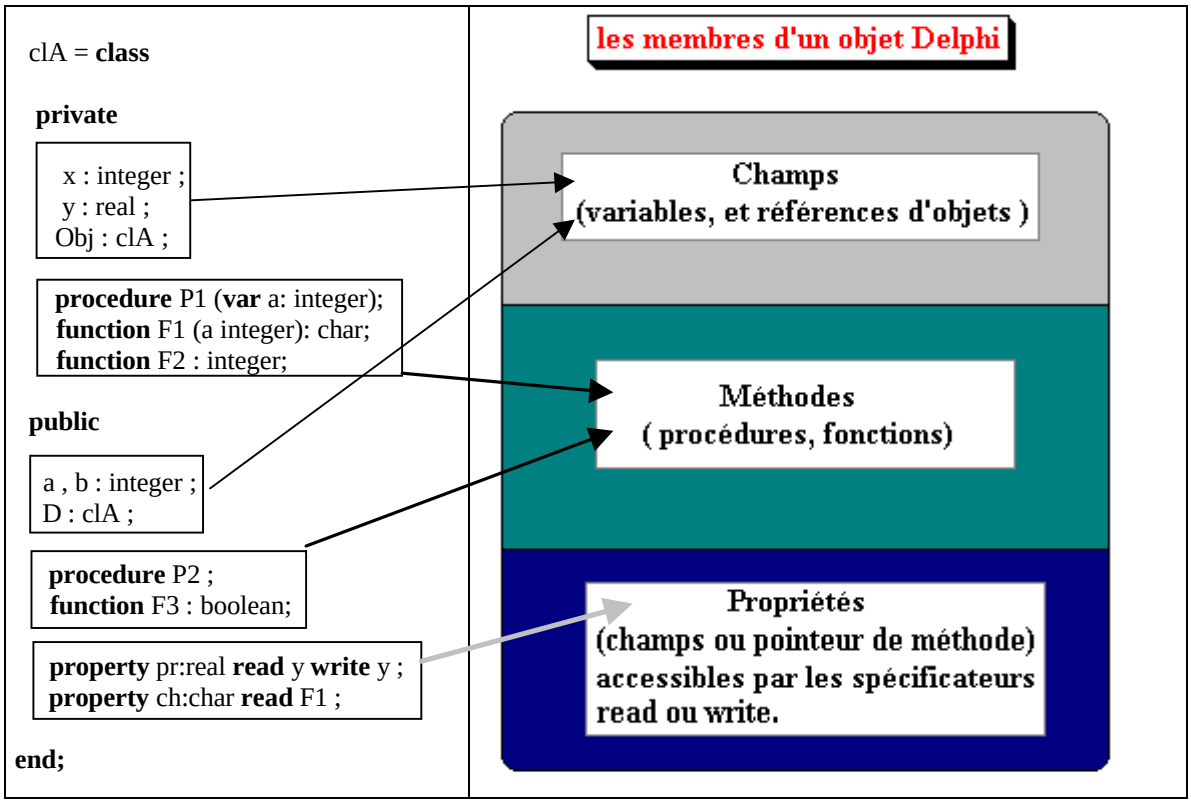
3.5 Les objets en Delphi

Un *objet* Delphi suit très exactement les définitions générales d'objets.

Un *objet* Delphi est une *instance* d'une classe.
Un *objet* Delphi contient des membres qui sont :

- des variables pascal (*champs ou attributs*)
- des procédures et des fonctions (*méthodes*)
- des **propriétés**.

Nous verrons plus loin que ces propriétés peuvent être de différentes catégories, en première lecture nous dirons que ce sont des champs possédant des spécificateurs d'accès.



Les classes et les objets sont déclarés dans les unités (dans la partie déclaration de l'**interface** ou de l'**implementation** de l'unité), le code des méthodes est défini uniquement dans la partie **implementation** de l'unité.

Constructeur / Destructeurs d'objets

En Delphi, chaque variable d'instance (*objet instancié*) doit obligatoirement être initialisée et peut être détruite lorsqu'elle n'est plus utile. Tout objet doit être d'abord construit avant son utilisation. Ceci se fait à l'aide d'une méthode spécifique déclarée par le mot clef **constructor**.

La destruction d'une instance s'effectue par une méthode aux propriétés identiques à un **constructor** et elle se dénomme un **destructor**.

- Par défaut les classes disposent (provenant de la classe mère **TObject**) d'une méthode permettant la construction des objets de la classe : cette méthode de type **constructor** est dénommée **Create**. Ce genre de méthode assure la création de l'objet physique en mémoire et sa liaison avec l'identificateur déclaré dans le code (l'identificateur contient après l'action du Create, l'adresse physique de l'objet).

- Il en est de même pour la désallocation des objets de vos classes (leur destruction), celles-ci disposent d'un destructeur d'objets dénommé **Destroy**. Cette méthode de type **destructor** rend au processus, tout l'espace mémoire utilisé par l'objet physique, mais attention elle ne dé-référence pas l'identificateur, si nécessaire cette opération est à la charge du programmeur (grâce à l'instruction : **identificateur:=nil**).

Unit Uclasses contenant une classe clA

```
Unit Uclasses ;
Interface type
clA = class
  private
     x : integer ;
     y : real ;
    Obj : clA ;
    procedure P1 (var a: integer);
    function F1 (a integer): char;
    function F2 : integer;
 public
   a ;  b : integer ;
  D : clA ;
   procedure P2 ;
   function F3 : integer;
   property prop1: integer  read x  write  x ;
   property prop2 : char read F1 ;
end;

implementation

  procedure clA.P1 (var a: integer); …
  function clA.F1 (a integer): char; …
  function clA.F2 : integer; …
  procedure clA.P2 ; …
  function clA.F3 : boolean; …

end.
```

Déclaration

code des méthodes

Programme principal utilisant Uclasses

```
Program principal ;

  Uses Uclasses ;

  procedure utilise (ObjA : clA);
  begin
    if  ObjA.a  > 5 then
    begin

    ObjA.P2 ;

     ObjA.b :=  ObjA.F3   - ObjA.a +1;

    end
    else

    prop1  := ObjA.a -1

  end;

  var refA : clA ;

  begin
   refA : = clA.Create ;
   utilise ( refA ) ;
   refA.destroy;
  end.
```

Déclaration d'une référence de type clA.

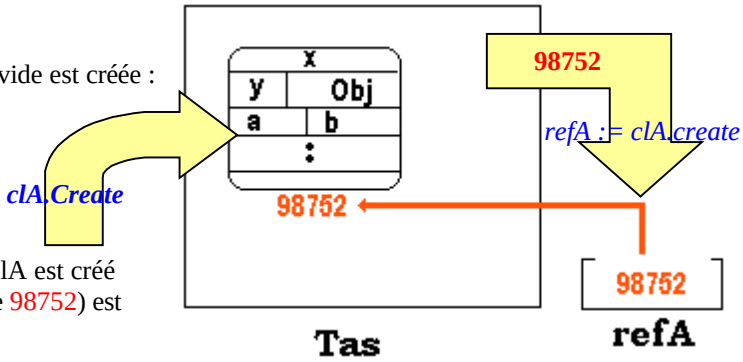
Instanciation d'un objet de type clA.

Destruction de l'objet de type clA.

var refA : clA ;

Lors de la déclaration une variable **refA** vide est créée :

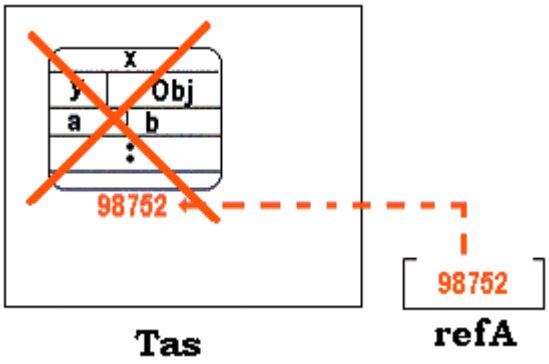
refA : = clA.Create ;



Lors de l'instanciation, un objet de type clA est créé dans le tas, puis sa référence (son adresse 98752) est mise dans la variable **refA** :

refA.destroy;

Lors de la destruction , l'objet du tas est détruit, la mémoire qu'il occupait est rendu au tas et la variable refA ne référence plus un objet.

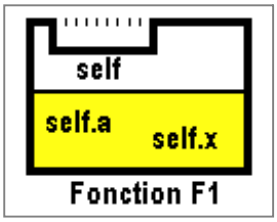


Le paramètre implicite self

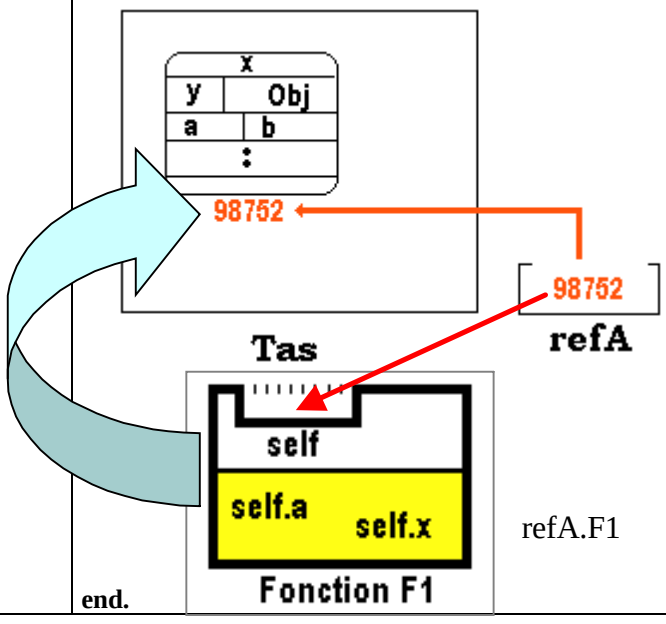
- Etant donnée une classe quelconque, chaque méthode de cette classe possède systématiquement dans son implémentation un paramètre implicite dénommé **Self**, que vous pouvez utiliser.
- Cet identificateur **Self** désigne l'objet (lorsqu'il sera instancié)dans lequel la méthode est appelée.

Exemple:

```
Unit Uclasses;
Interface
type
clA = class
   private
      Obj : clA ;
  public
     x , y : integer ;
     a , b : integer ;
    function F1 : integer;
end;

implementation

function clA.F1 : integer;
begin
  self.a := self.x +2;
  result := self.a;
end;

end.
```




```
Program principal ;

    Uses Uclasses ;
var refA : clA ;
       n : integer;

    begin
     refA : = clA.Create ; ….
```




```
end.
```

Lors de l'instanciation (refA : = clA.Create) la valeur 98752 de la référence, est passée automatiquement dans la variable implicite **self** de chaque méthode de l'objet **refA**, ce qui a pour résultat que dans la méthode F1, les expressions formelles **self.a** et **self.x** prennent une valeur effective et désignent les valeurs effectives des champs **a** et **x** de l'objet n° **98752** du **tas**.

3.6 Encapsulation

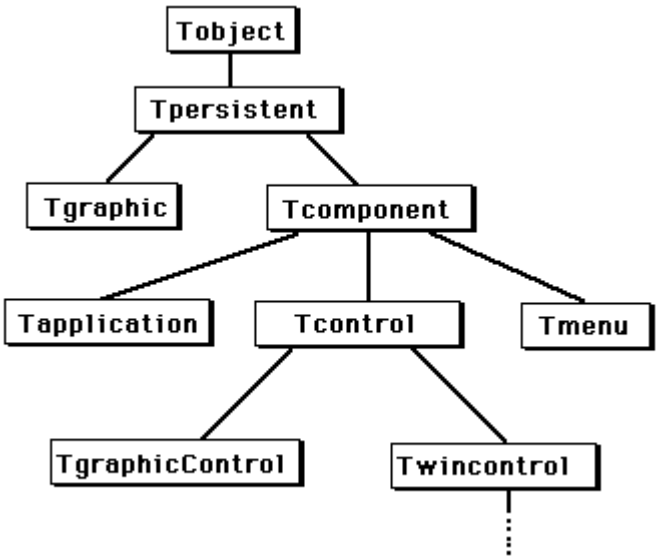
Rappelons que pratiquement, en POO l'encapsulation permet de masquer les informations et les opérations d'un objet aux autres objets. Contrairement à certains autres langages orientés objet, dans Delphi par défaut, s'il n'y a pas de descripteur d'encapsulation, tout est visible (donc **public**).

Delphi possède au moins quatre niveaux d'encapsulation des informations dans un objet qui sont matérialisés par des descripteurs :

published : Les informations sont accessibles par toutes les instances de toutes les classes (les *clients*) + accessibles à l'inspecteur d'objet de Delphi.
public : Les informations sont accessibles par toutes les instances de toutes les classes (les *clients*).
protected : Les informations ne sont accessibles qu'à toutes les instances de la classe elle-même et à toutes celles qui en *héritent* (ses descendants).
private : Les informations ne sont accessibles qu'à toutes les instances de la classe elle-même.

3.7 Héritage

Il s'agit en Delphi de *l'héritage simple* (graphe arborescent) dans lequel une famille dérive d'une seule classe de base. Voici une partie du graphe d'héritage simple de certaines classes de Delphi :



Syntaxe de la déclaration d'héritage :

Type classe_fille = **class**(classe_ancetre)

Type classe_ancetre = **class** { champs } { méthodes } **end**;	**Type** classe_fille = **class**(classe_ancetre) { champs } { méthodes } **end**;

Nous indiquons donc ainsi en Delphi que la classe_fille hérite de la classe_ancetre.

Nous avons vu que les deux déclarations ci-dessous étaient équivalentes :

Type ma_classe = **class** {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;	**Type** ma_classe = **class**(TObject) {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;

L'écriture de gauche indique en fait que toutes les classes déclarées sans qualificatif d'héritage héritent **automatiquement** de la classe TObject. La VCL de Delphi est entièrement construite par héritage (une hiérarchie objet complète) à partir de TObject.

*Ci-dessous la déclaration de la classe **TObject** dans Delphi :*

```
TObject = class
  constructor Create;
  procedure Free;
  class function InitInstance(Instance: Pointer): TObject;
  procedure CleanupInstance;
  function ClassType: TClass;
  class function ClassName: ShortString;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: Pointer;
  class function InstanceSize: Longint;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress(const Name: ShortString): Pointer;
  class function MethodName(Address: Pointer): ShortString;
  function FieldAddress(const Name: ShortString): Pointer;
  function GetInterface(const IID: TGUID; out Obj): Boolean;
  class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
  class function GetInterfaceTable: PInterfaceTable;
  function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HResult; virtual;
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
  class function NewInstance: TObject; virtual;
  procedure FreeInstance; virtual;
  destructor Destroy; virtual;
end;
```

*La classe Tobject utilise ici la notion de **méthode de classe** :*

- Une méthode de classe est une méthode (autre qu'un constructeur) qui **agit sur des classes** et non sur des objets.

- La définition d'une méthode de classe doit commencer par le mot réservé **class.**

Par exemple dans Tobject :

```
TObject = class
  ...
  class function ClassName: ShortString;
  class function ClassParent: TClass; ...
```

- La déclaration de définition d'une méthode de classe doit également commencer par class :

```
class function TObject.ClassName: ShortString;
begin
  //...le paramètre self est ici la classe Tobject
end;
etc...
```

Outre la classe TObject, Delphi fournit le type de méta-classe (référence de classe) générale **TClass** :

```
TClass = class of TObject;
```

3.8 surcharge de méthode

Signature d'une méthode -- définition C'est son nom , le nombre et le type de ses paramètres

On dit qu'une méthode est surchargée dans sa classe si l'on peut trouver dans la classe plusieurs signatures différentes de la même méthode.

En Delphi, les en-têtes des méthodes surchargées de la même classe, doivent être suivies du qualificateur **overload** afin d'indiquer au compilateur qu'il s'agit d'une autre signature de la même méthode.

Dans la classe classeA nous avons déclaré 3 surcharges de la même méthode Prix, qui pourrait évaluer un prix en fonction du montant rentré selon trois types de données exprimées en yen, en euro ou en dollar.	**Type** classeA = **class** **public** **function** Prix(x:yen) : real;**overload**; **function** Prix(x:euro): real;**overload**; **function** Prix(x:dollar) : real;**overload**; **end**;

Comment appeler une surcharge de méthode ?

Soit le programme de droite qui utilise la classeA définie à gauche avec 3 surcharges de la méthode Prix

```
Unit Uclasses ;
interface

Type
yen = class … end;
euro = class … end;
dollar = class … end;
classeA = class
    public
        function Prix(x:yen) : real;overload;
        function Prix(x:euro): real;overload;
        function Prix(x:dollar) : real;overload;
 end;

implementation

function classeA.Prix(x:yen) : real; // signature n°1
begin
  …
end;

function classeA.Prix(x:euro): real;
begin
  …
end;

function classeA.Prix(x:dollar) : real;
begin
  …
end;
```

```
Program principal ;
Uses Uclasses ;

var refA : classeA;
      a : yen ; b : euro ; c : dollar ;

Procedure Calcul1 (ObjA : classeA; valeur : yen );
begin
  ObjA.Prix ( valeur)
end;

Procedure Calcul2 (ObjA : classeA;  valeur : euro);
begin
  ObjA.Prix ( valeur )
end;

Procedure Calcul3 (ObjA : classeA;  valeur : dollar);
Var ObjA : classeA ;
begin
  ObjA.Prix ( valeur)
end;

begin
 refA:= classeA.Create ;
 a := yen.Create ;
 b := euro.Create ;
 c := dollar.Create ;
 Calcul1 ( refA , a );
 Calcul2 ( refA , b );
 Calcul3 ( refA , c );
End.
```

Le compilateur connaît le type du second paramètre, lorsqu'il appelle :
ObjA.Prix (valeur)

Il va alors chercher s'il existe une signature correspondant à ce type et va exécuter le code de la surcharge adéquate

- L'appel ObjA.Prix(valeur) dans Calcul1(refA , a) sur a de type yen provoque la recherche de la signature suivante **Prix** (type **yen**), c'est la *signature n°1* qui est trouvée et exécutée.
- L'appel ObjA.Prix(valeur) dans Calcul2(refA , b) sur a de type euro provoque la recherche de la signature suivante **Prix** (type **euro**)), c'est la *signature n° 2* qui est trouvée et exécutée.
- L'appel ObjA.Prix(valeur) dans Calcul3(refA , c) sur a de type dollar provoque la recherche de la signature suivante **Prix** (type **dollar**)), c'est la *signature n°3* qui est trouvée et exécutée.

4. Les propriétés en Delphi

Une propriété définie dans une classe permet d'accéder à certaines informations contenues dans les objets instanciés à partir de cette classe. Une propriété a la même syntaxe de définition et d'utilisation que celle d'un champ d'objet (elle possède un type de déclaration), mais en fait elle peut invoquer une ou deux méthodes internes pour fonctionner ou se référer directement à un champ. Les méthodes internes sont déclarées à l'intérieur d'un bloc de défintion de la propriété.

Nous nous limiterons aux propriétés non tableau, car cette notion est commune à d'autres langages (C# en particulier)

4.1. Définition

Comme un champ, une propriété définit un attribut d'un objet.

Mais un champ n'est qu'un emplacement de stockage dont le contenu peut être consulté et modifié, tandis qu'une propriété peut associer des actions spécifiques à la lecture et lors de la modification de ses données : une propriété peut être utilisée comme un attribut.

Les propriétés proposent un moyen de contrôler l'accès aux attributs d'un objet et autorisent ou non le calcul sur les attributs.

Syntaxe :
property nomPropriété[indices] : type [index constanteEntière] spécificateurs ;

Remarque :

il faut au minimum un descripteur (ou spécificateur) d'accès pour une propriété :

- soit lecture seule (spécificateur **read**),
- soit écriture seule (spécificateur **write**),
- soit lecture et écriture **read …. write** .

Attention :

Une propriété **ne peut pas être transmise comme paramètre** référence dans une procédure ou une méthode !

Exemple de syntaxe d'écriture de propriétés :

```
classeA = class
   public
      property prop1 : integer  read y write z ; // lecture et écriture
      property prop2 : char read F1 ; // lecture seule
      property prop3 : string write t ; // écriture seule
 end;
```

4.2. Accès par read/write aux données d'une propriété

Après les spécificateurs **read** et **write**, il obligatoire de préciser le moyen d'accès à la propriété : Ce moyen peut être un **attribut**, ou une **méthode**.

Accès à une propriété par un attribut

property *propriété1: type* **read** *Fpropriété1* ;

La **property** *propriété1* fait référence à l'attribut *Fpropriété1* et en permet l'accès en lecture seule.

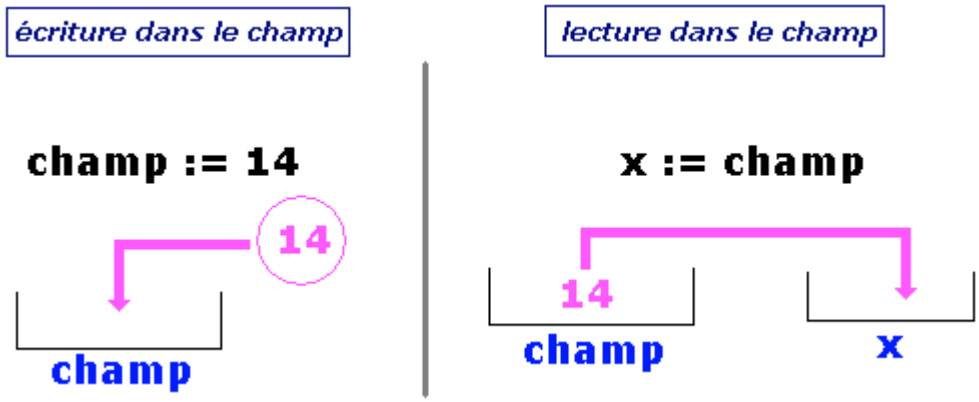
Pour avoir un intérêt, la **property** *propriété1* doit être déclarée en **public**, tandis que l'attribut *Fpropriété1* est déclaré en **private**.

Lorsque la propriété est définie, il faut que le champ auquel elle se réfère ait déjà été défini dans la classe, ou dans une classe ancêtre.

D'une façon générale on peut comparer la lecture et l'écriture dans un champ et dans une propriété comme ci-dessous :

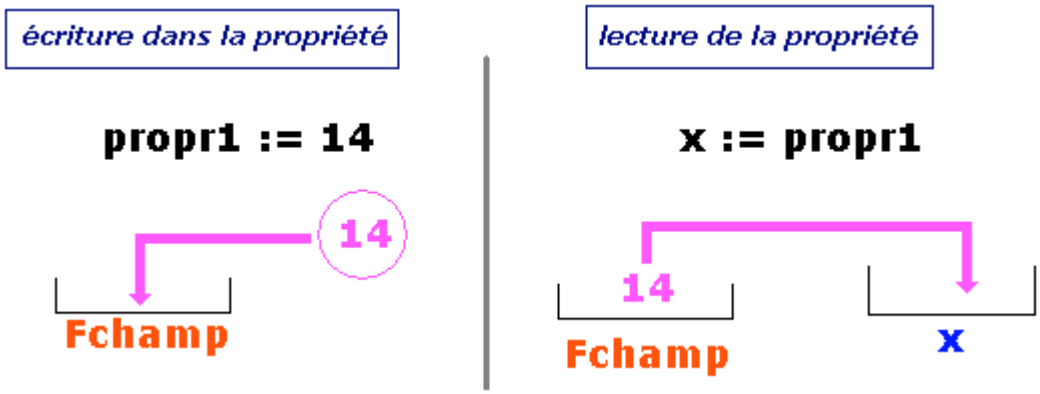
Soit la classe classeA :

```
classeA = class
    public
      champ : integer ;
 end;
```



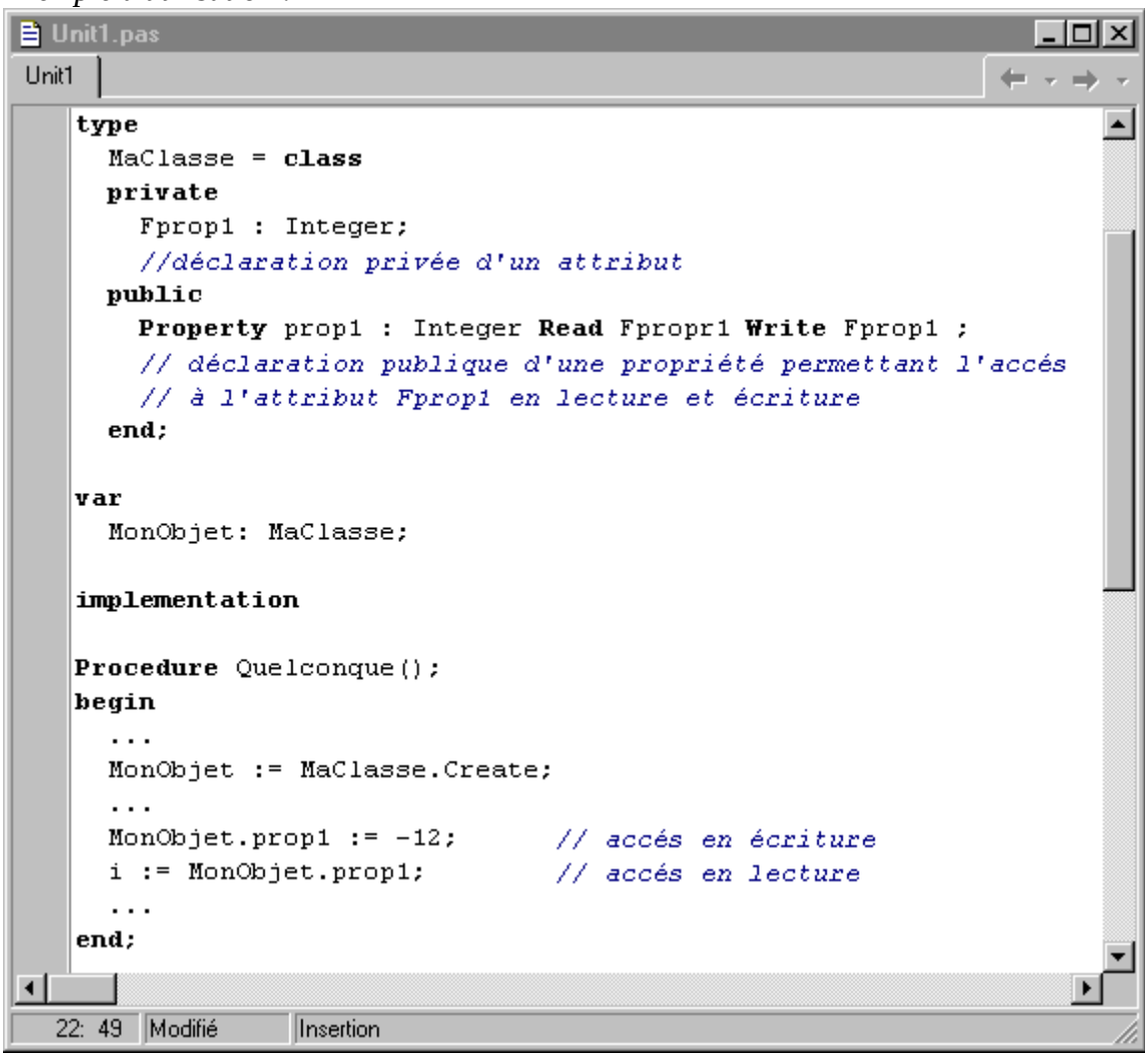
Soit une autre version de la classe classeA :

```
classeA = class
    private
      Fchamp : integer ;
    public
      property propr1 : integer read Fchamp write Fchamp ;
 end;
```



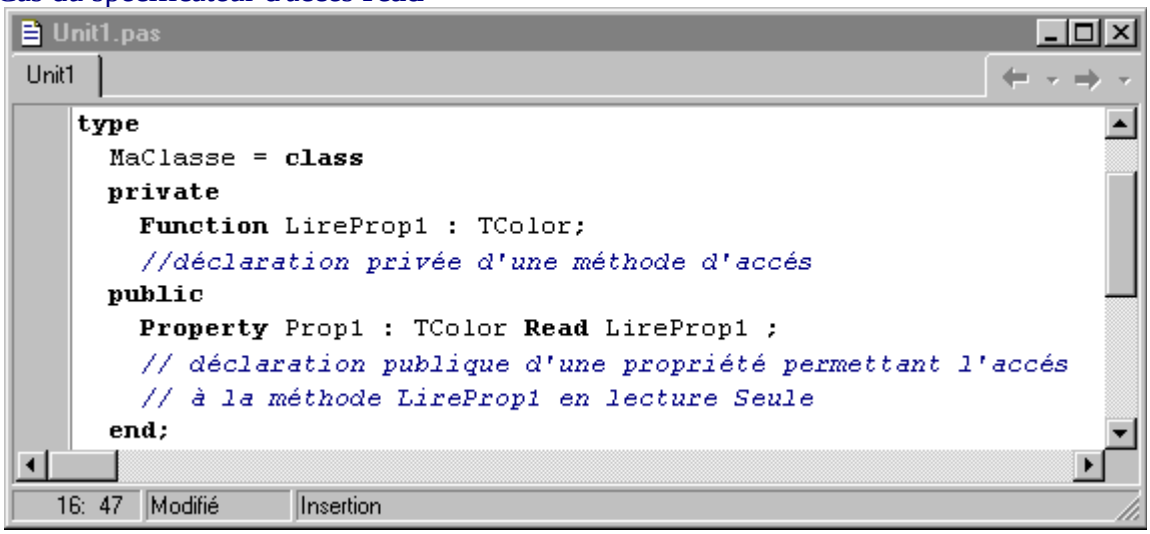
On peut donc assimiler la propriété **propr1** à une clef ouvrant une porte sur un champ privé de l'objet. et autorisant la lecture et/ou l'écriture dans ce champ.

Exemple d'utilisation :



Accès à une propriété par une méthode

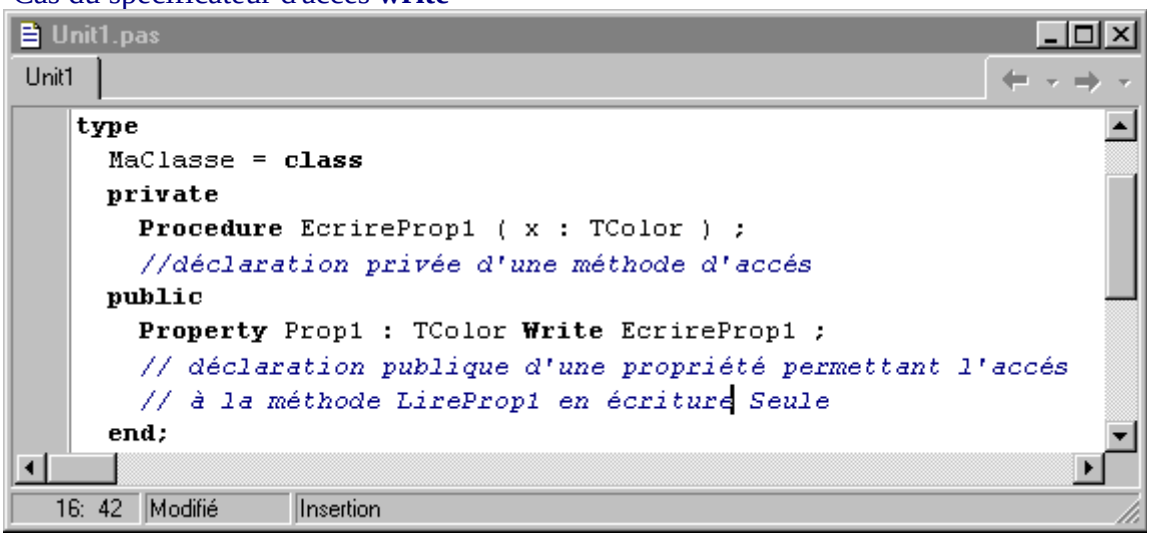
Cas du spécificateur d'accès **read**



La méthode d'accès en lecture **LirеProp1** doit :

- être une fonction,
- être sans paramètres,
- renvoyer un résultat du même type que celui de la propriété.

Cas du spécificateur d'accès **write**



La méthode d'accès en écriture **EcrirеProp1** doit :

- être une procédure,
- n'avoir qu'un seul paramètre formel passé par constante ou par valeur (**pas par référence**),
- ce paramètre doit être du même type que celui de la propriété.

4.4 Surcharge de propriétés

Surcharge permettant d'augmenter la visibilité

Lorsqu'une propriété est déclarée dans une classe, on peut la surcharger dans les classes dérivées en augmentant son niveau de visibilité.

```
MaClasse = class
   private
     Fchamp : integer ;
   protected
     property propr1 : integer read Fchamp
                               write Fchamp ;
 end;
```

```
MaFille = class (MaClasse)
   private
     Fchamp : integer ;
   public
     property propr1 : integer read Fchamp
                               write Fchamp ;
 end;
```

Surcharge permettant de redéfinir un spécificateur existant

On ne peut pas supprimer de spécificateur.
Par contre on peut modifier le/les spécificateur(s) existant(s) ou ajouter le spécificateur manquant.

```
MaClasse = class
    private
      Fchamp : integer ;

    public
      property propr1 : integer read Fchamp ;
      property propr2 : integer read Fchamp ;
 end;
```

```
MaFille = class (MaClasse)
    private
      Fchamp : integer ;
      function lirePropr2 : integer ;
    public
      property propr1 : integer read Fchamp write Fchamp ;
      property propr2 : integer read lirePropr2 ;

 end;
```

Redéfinition et masquage d'une propriété

Une propriété redéfinie dans une classe remplace l'ancienne avec ses nouveaux attributs, son nouveau type.

```
MaClasse = class
    private
      Fchamp : integer ;
    protected
      property propr1 : integer read Fchamp
                                write Fchamp ;
 end;
```

```
MaFille = class (MaClasse)
    private
      FNom : string ;
    public
      property propr1 : string  read Fnom ;

 end;
```

Dés qu'une redéclaration de propriété contient une redéfinition de type, elle masque automatiquement la propriété parent héritée et la remplace entièrement. Elle doit donc être redéfinie avec au moins un spécificateur d'accès.

5.3 Polymorphisme avec Delphi

Plan de ce chapitre:

Polymorphisme d'objet

- Introduction
- Instanciation et utilisation dans le même type
- Instanciation et utilisation dans un type différent
- Polymorphisme implicite
- Instanciation dans un type descendant
- Polymorphisme explicite par transtypage
- Utilisation pratique du polymorphisme d'objet
- instanciation dans un type ascendant

Polymorphisme de méthode

- Introduction
- Vocabulaire et concepts
- Surcharge dans la même classe
- Surcharge dans une classe dérivée
- Surcharge dynamique dans une classe dérivée
- Répartition des méthodes en Delphi
- Réutilisation de méthodes avec inherited

Polymorphisme de classe abstraite

- Introduction
- Vocabulaire et concepts

Exercice traité sur le polymorphisme

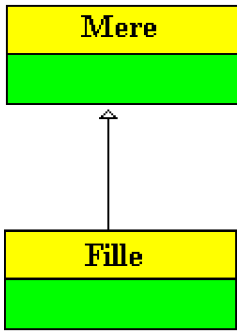
1. Polymorphisme d'objet

Conversion de références d'objet entre classe et classe dérivée

Il existe un concept essentiel en POO désignant la capacité d'une hiérarchie de classes à fournir différentes implémentations de méthodes portant le même nom et par corollaire la capacité qu'ont des objets enfants de modifier les comportements hérités de leur parents. Ce concept d'adaptation à différentes "situations" se dénomme le **polymorphisme** qui peut être implémenté de différentes manières.

Polymorphisme d'objet - *définition générale*
C'est une interchangeabilité entre variables d'objets de classes de la même hiérarchie sous certaines conditions, que dénommons le polymorphisme d'objet.

Soit une classe **Mere** et une **Fille** héritant de la classe **Mere** :



Les objets peuvent avoir des comportements polymorphes (s'adapter et se comporter différemment selon leur utilisation) licites et des comportements polymorphes dangereux selon les langages.

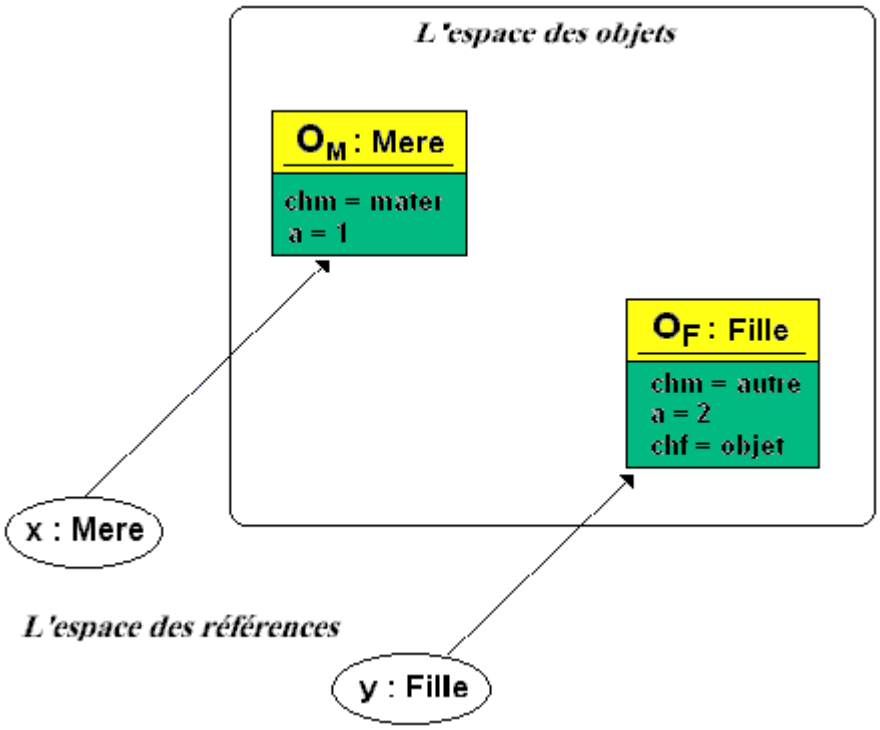
Dans un langage dont le modèle objet est la référence (un objet est un couple : référence, bloc mémoire) comme **C++, C#, Delphi ou Java**, il y a découplage entre les actions statiques du compilateur et les actions dynamiques du système d'exécution selon le langage utilisé le compilateur protège ou non statiquement des actions dynamiques sur les objets une fois créés. C'est la déclaration et l'utilisation des variables de références qui autorise ou non les actions licites grâce à la compilation.

Supposons que nous ayons déclaré deux variables de référence, l'une de classe **Mere**, l'autre de classe **Fille**, une question qui se pose est la suivante : au cours du programme quel genre d'affectation et d'instanciation est-on autorisé à effectuer sur chacune de ces variables. L'héritage permet une variabilité entre variables d'objets de classes de la même hiérarchie, c'est cette variabilité que dénommons le **polymorphisme d'objet**.

Nous allons dès lors envisager toutes les situations possibles et les évaluer, les exemples appuyant le texte sont présentés en Delphi.

instanciation dans le type initial et utilisation dans le même type

Il s'agit ici d'une utilisation la plus classique qui soit, dans laquelle une variable est utilisée dans son type de définition initial.

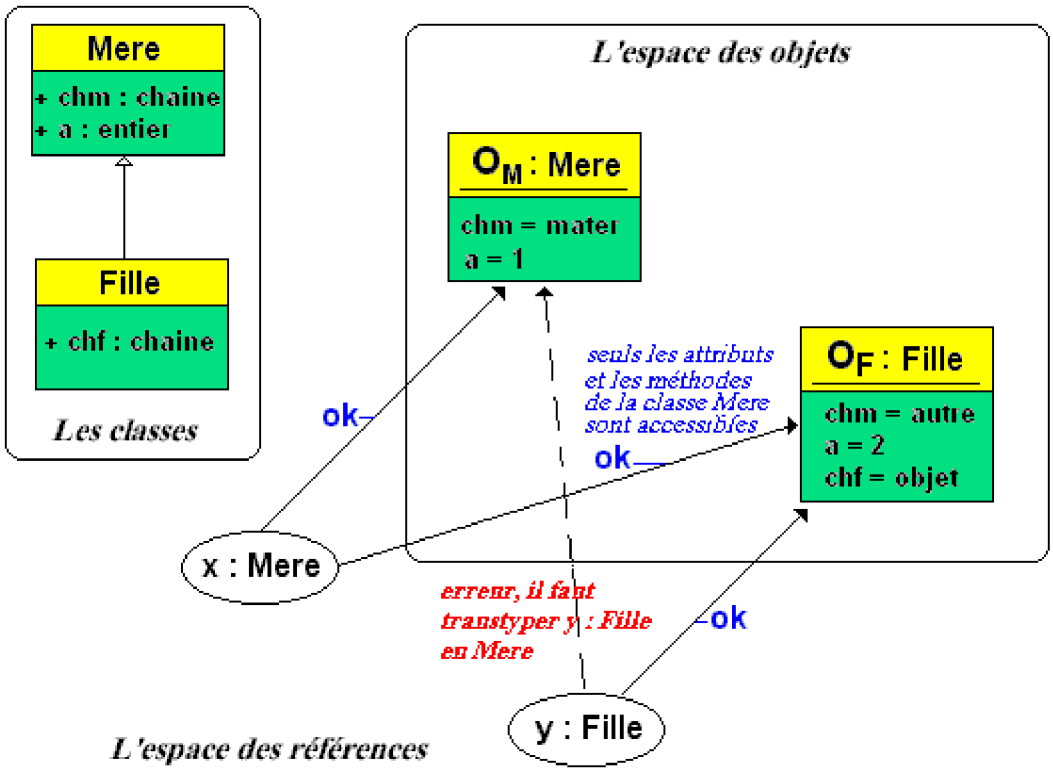


Delphi

```
var
   x,u : Mere;
   y,v : Fille ;
.....
x : = Mere.Create ; // instanciation dans le type initial
u := x; // affectation de références du même type
y : = Fille.Create ;  // instanciation dans le type initial
v := y; // affectation de références du même type
```

instanciation dans le type initial et utilisation différente

Il s'agit ici de l'utilisation licite commune à tous les langages cités plus haut, nous illustrons le discours en explicitant deux champs de la classe Mere (chm:chaine et a:entier) et un champ supplémentaire (chf:chaine) dans la classe Fille. il existe 3 possibilités différentes, la figure ci-dessous indique les affectations possibles :




```
var
  x , ObjM : Mere;
  y , ObjF : Fille;
```

```
ObjM := Mere.Create; // instanciation dans le type initial
ObjF := Fille.Create; // instanciation dans le type initial
 x := ObjM; // affectation de références du même type
 x := ObjF; // affectation de références du type descendant implicite
```

```
 y := ObjF; // affectation de références du même type
 y := Fille(ObjM); // affectation de références du type ascendant explicite mais dangereux si ObjM est uniquement Mere
```

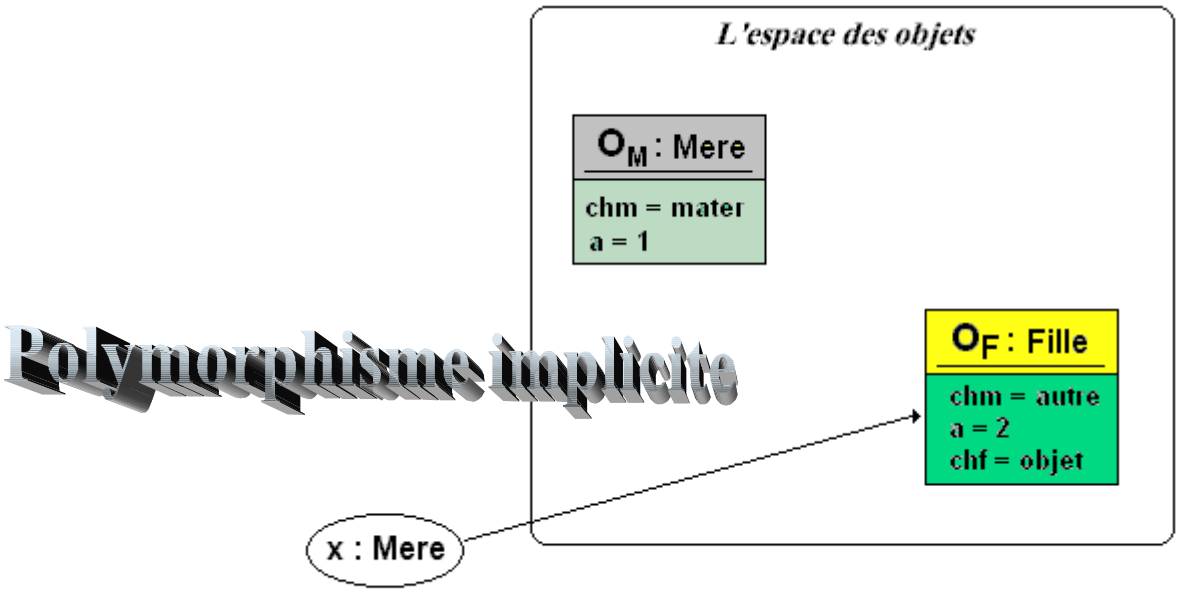
Les trois possibilités sont :

- L'instanciation et l'utilisation de références dans le même type
- L'affectation de références : polymorphisme implicite
- L'affectation de références : polymorphisme par transtypage d'objet

La dernière de ces possibilités pose un problème d'exécution lorsqu'elle est mal employée !

Polymorphisme d'objet implicite

Dans l'exemple précédent le compilateur accepte le transtypage 'y :=Fille(ObjM)' car il autorise un polymorphisme d'objet de classe ascendante vers une classe descendante (c'est à dire que ObjM peut se référer implicitement à tout objet de classe **Mere** ou de toute classe **descendante** de la classe Mere).



Polymorphisme implicite

Nous pouvons en effet dire que **x** peut se référer implicitement à tout objet de classe **Mere** ou de **toute classe héritant** de la classe **Mere** :

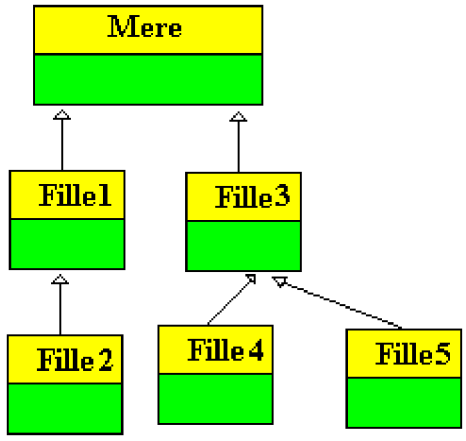


fig - 1

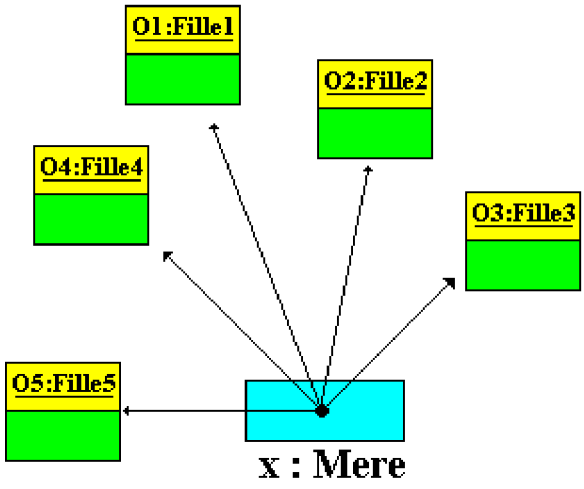


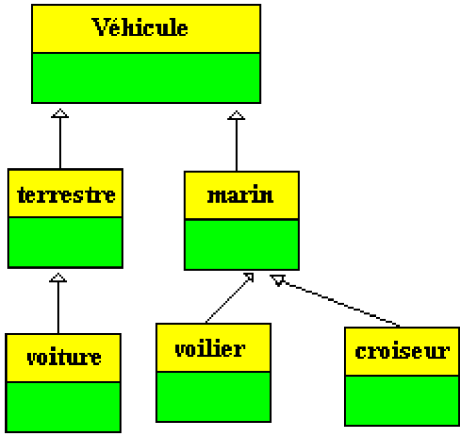
fig - 2

Dans la figure fig-1 ci-dessus, une hiérarchie de classes decendant toutes de la classe **Mere**.

Dans la figure fig-2 ci-dessus le schéma montre une référence de type **Mere** qui peut '**pointer**' vers n'importe quel objet de classe descendante (polymorphisme d'objet).

Exemple pratique tiré du schéma précédent

Le polymorphisme d'objet est typiquement fait pour représenter des situations pratiques figurées ci-dessous : (Mere=vehicule, Fille1=terrestre, Fille2=voiture, Fille3=marin, Fille4=voilier, Fille5=croiseur)



Une hiérarchie de classes de véhicules descendant toutes de la classe mère **Vehicule**.

Déclaration de cette hiérarchie en Delphi

```
Vehicule = class
  .........
end;

terrestre = class (Vehicule)
 .........
end;

voiture = class (terrestre)
  .........
end;
```

```
Marin = class (Vehicule)
.........
end;

voilier = class (marin)
 .........
end;

croiseur = class (marin)
.........
end;
```

On peut énoncer le fait qu'un véhicule peut être de plusieurs sortes : soit un **croiseur**, soit une **voiture**, soit un véhicule **terrestre** etc...

En traduisant cette phrase en termes informatiques :

Si l'on déclare une référence de type véhicule **(var** x : **vehicule**) elle pourra pointer vers n'importe quel objet d'une des classe filles de la classe **vehicule**.

Polymorphisme implicite = création d'objet de classe descendante référencé par une variable parent

Quand peut-on écrire x := y sur des objets ?

D'une façon générale vous pourrez toujours écrire des affectations entre deux références d'objets :

```
var
  x : Classe1
  y : Classe2
   .........
  x := y;
```

Delphi

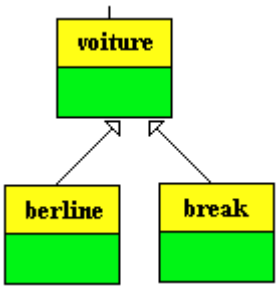
si et seulement si Classe2 est une classe descendante de Classe1.

instanciation dans un type descendant

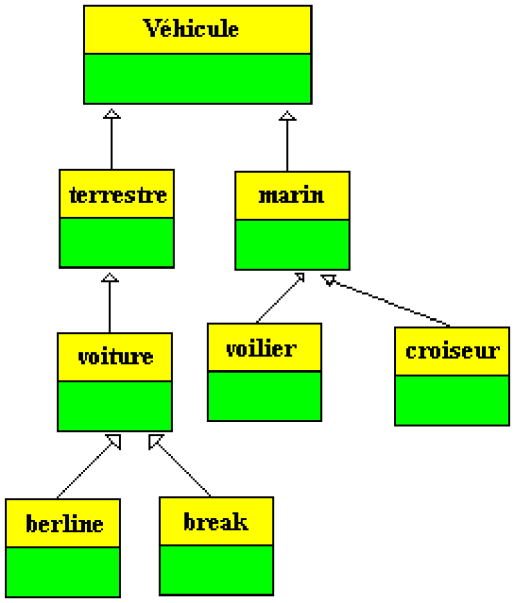
Polymorphisme par création d'objet de classe descendante

Dans ce paragraphe nous signalons qu'il est tout à fait possible, du fait du transtypage implicite, de créer un objet de classe descendante référencé par une variable de classe parent.

Ajoutons 2 classes à la hiérarchie des véhicules :



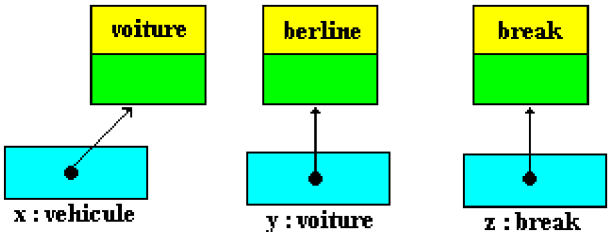
La nouvelle hiérarchie est la suivante :



Ensuite nous déclarons 3 références de type **x:vehicule**, **y:voiture** et **z:break , puis** nous créons 3 objets de classe **voiture**, **berline** et **break**, il est possible de créer directement un objet de classe descendante à partir d'une référence de classe mère :

- on crée une **voiture** référencée par la variable x de classe **vehicule**,
- on crée une **berline** référencée par la variable y de classe **voiture**,
- enfin on crée un **break** référencé par la variable z de classe **break**.

Réécrivons ces phrases afin de comprendre à quel genre de situation pratique cette opération correspond :

on crée un **véhicule** du type **voiture** on crée une **voiture** de type **berline** enfin on crée un **break** de type **break**	

```
var
 x : vehicule;
 y : voiture;
 z : break;
...
x := voiture.Create; // objet de classe enfant voiture référencé par x de classe parent vehicule
y := berline.Create; // objet de classe enfant berline référencé par x de classe parent voiture
z := break.Create; // instanciation dans le type initial
```

Polymorphisme d'objet explicite par transtypage

Reprenons le code précédent en extrayant la partie qui nous intéresse :

```
var
  x , ObjM : Mere;
  y : Fille;

ObjM := Mere.Create; // instanciation dans le type initial
 x := ObjM; // affectation de références du même type
 y := Fille(ObjM); // affectation de références du type ascendant explicite licite
```

Nous avons signalé que l'affectation y := Fille(ObjM) pouvait être dangereuse si ObjM pointe vers un objet purement de type Mere. Voyons ce qu'il en est.

Nous avons vu plus haut qu'une référence de type parent peut '**pointer**' vers n'importe quel objet de classe descendante. Si l'on sait qu'une reference **x** de classe parent, pointe vers un objet de classe enfant, on peut en toute sureté procéder à une affectation de cette référence à une autre reference **y** définie comme reference classe enfant (opération y := x).

La situation informatique est la suivante :

- on déclare une variable x de type Mere,
- on déclare une variable y de type Fille héritant de Mere,
- on instancie la variable x dans le type descendant Fille (polymorphisme implicite).

Il est alors possible de faire "pointer" la variable y (de type Fille) vers l'objet (de type Fille) auquel se réfère x en effectuant une affectation de références.

Toutefois le compilateur refusera l'écriture y := x, il suffit de lui indiquer qu'il faut

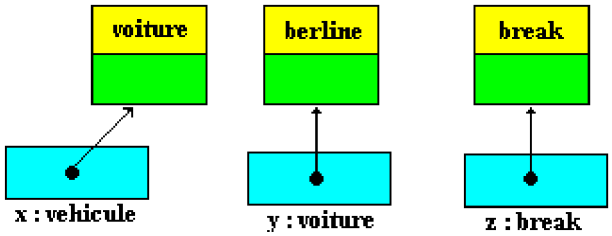
transtyper la variable de référence x et la considérer dans cette instruction comme une reférence sur un enfant

```
var
  x : Mere;
  y  : Fille;
```

Delphi

```
x := Mere.Create; // instanciation dans le type initial
y := Fille(ObjM); // affectation de références du type ascendant explicite licite
```

Dans la dernière instruction, la **reférence ObjM est transtypée** en type Fille, de telle manière que le compilateur puisse faire pointer y vers l'objet déjà pointé par ObjM. En reprenant l'exemple pratique de la hiérarchie des véhicules :

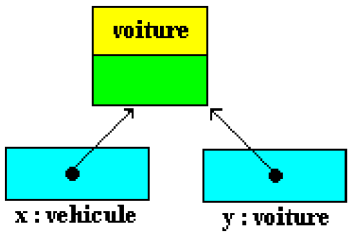


Puisque x pointe vers un objet de type **voiture** toute variable de référence voiture acceptera de pointer vers cet objet, en particulier la variable y :voiture après transtypage de la référence de x.

```
var
 x : vehicule;
 y : voiture;
...
x := voiture.Create; // objet de classe enfant voiture référencé par x de classe parent vehicule
y := voiture ( x ); // transtypage
```

En Delphi l'affectation s'écrit par application de l'opérateur de transtypage :

y := voiture (x);



ATTENTION

- La validité du transtypage n'est pas vérifiée statiquement par le compilateur, donc si votre variable de référence pointe vers un objet qui n'a pas la même nature que l'opérateur de transtypage, c'est de l'exécution qu'il y aura production d'un message d'erreur indiquant le transtypage impossible.
- Il est donc impératif de tester l'appartenance à la bonne classe de l'objet à transtyper avant de le transtyper, les langages C#, Delphi et Java disposent d'un opérateur permettant de tester cette appartenance ou plutôt l'appartenance à une hiérarchie.

L'opérateur "is" en

L'opérateur **is**, qui effectue une vérification de type dynamique, est utilisé pour vérifier quelle est effectivement la classe d'un objet à l'exécution.

L'expression : objet **is** classeT

renvoie True si objet est une instance de la classe désignée par **classeT** ou de l'**un de ses descendants**, et False sinon. Si objet a la valeur nil, le résultat est False.

L'opérateur "as" en 

L'opérateur **as** est un opérateur de transtypage de référence d'objet semblable à l'opérateur ( ).
L'opérateur **as** fournit la valeur **null** en cas d'échec de conversion alors que l'opérateur ( ) lève une exception.

(objet **as** classeT) renvoie une référence de type classeT

Exemple d'utilisation des deux opérateurs :

```
var x : classeT;
if objet is classeT then
  x := objet as classeT ;
```

Utilisation pratique du polymorphisme d'objet

Le polymorphisme d'objet associé au transtypage est très utile dans les paramètres des méthodes.

Lorsque vous déclarez une méthode P avec un paramètre formel de type ClasseT :

```
procedure P( x : ClasseT );
begin
   ........
end;
```

Vous pouvez utiliser lors de l'appel de la procédure P n'importe quel paramètre effectif de ClasseT ou bien d'une quelconque classe descendant de ClasseT et ensuite à l'intérieur de la procédure vous transtypez le paramètre. Cet aspect est abondamment utilisé en Delphi lors de la création de gestionnaires d'événements communs à plusieurs objets :

```
procedure P1( Sender : Tobject );
begin
   if Sender is TEdit then
     TEdit(Sender).text := 'ok'
   else
    if Sender is TButton then
     TButton(Sender).caption := 'oui'
  ............
end;
```

Autre exemple avec une méthode P2 personnelle sur la hiérarchie des véhicules définies plus haut :

```
procedure P2( Sender : vehicule );
begin
    if  Sender is voiture then
      voiture(Sender).  .....
   else
     if Sender is voilier then
      voilier(Sender). .....
  .....
end;
```

2. Polymorphisme de méthode

Introduction

Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires nouvelles peuvent être implémentées dans la classe enfant mais aussi des méthodes des parents redéfinies pour obtenir des implémentations différentes.

Une classe dérivée hérite de tous les membres de sa classe parent ; c'est-à-dire que tous les membres du parent sont disponibles pour l'enfant, rappelons qu'une méthode est un membre qualifiant un comportement d'un objet de la classe. En POO on distingue deux catégories de méthodes selon les besoins des applications et du polymorphisme : **les méthodes statiques et les méthodes dynamiques**.

2.1 Vocabulaire et concepts généraux :

- L'action qui consiste à donner le même nom à plusieurs méthodes dans la même classe ou d'une classe parent à une classe enfant, se dénomme d'une manière générale la **surcharge de nom de méthode** (avec ou non la même signature).
- Le vocabulaire n'étant pas stabilisé selon les auteurs (surcharge, redéfinition, substitution,...) nous employerons les mots **redéfinition, surcharge dynamique ou substitution dans le même sens**, en précisant lorsque cela s'avérera nécessaire de quel genre de laison il s'agit.

> Les actions des méthodes héritées du parent peuvent être modifiés par l'enfant de deux manières, selon le type de liaison du code utilisé pour la méthode (**la liaison statique ou précoce** ou bien **la liaison dynamique ou retardée**).

Les deux modes de liaison du code d'une méthode

La liaison statique ou précoce (early-binding) :

- Lorsqu'une méthode à liaison statique est invoquée dans le corps d'un programme, le compilateur établit immédiatement dans le code appelant l'adresse précise et connue du code de la méthode à invoquer. Lors de l'exécution c'est donc toujours le même code invoqué.

La liaison dynamique ou retardée (lazy-binding) :

- Lorsqu'une méthode à liaison dynamique est invoquée dans le corps d'un programme, le compilateur **n**'établit **pas** immédiatement dans le code appelant l'adresse de la méthode à invoquer. Le compilateur met en place un mécanisme de reférence (référence vide lors de la compilation) qui, lors de l'exécution, désignera (pointera vers) le code que l'on voudra invoquer; on pourra donc invoquer des codes différents.

2.2 Surcharge dans la même classe :

Dans une classe donnée, plusieurs méthodes peuvent avoir le même nom, mais les signatures des méthodes ainsi surchargées doivent **obligatoirement** être différentes et peuvent **éventuellement** avoir des niveaux de visibilité différents.

Nous avons déjà vu les bases de ce type de surcharge lors de l'étude de Delphi et la POO. Soit par exemple, la classe ClasseA ci-dessous, ayant 3 méthodes de même nom P, elles sont surchargées dans la classe selon 3 signatures différentes :

```
Classe A
      public methode P(x,y);
      privé methode P(a,b,c);
      protégé methode P( );
finClasse A
```

La première surcharge de P dispose de 2 paramètres, la seconde de 3 paramètres, la dernière enfin n'a pas de paramètres. C'est le compilateur du langage qui devra faire le choix pour sélectionner le code de la bonne méthode à utiliser. Pour indiquer ce genre de surcharge, en Delphi il faut utiliser un qualificateur particulier dénoté **overload**.

Syntaxe de l'exemple en Delphi, en Java et en C# :

Delphi	Java - C#

ClasseA = **class**
public
procedure P(x,y : integer);**overload**;
private
procedure P(a,b,c : string); **overload**;
protected
procedure P;**overload**;
end;
	class ClasseA {
	public void P(**int** x,y){ }
	private void P(**String** a,b,c){ }
	protected void P( ){ }
	}

Utilisation pratique : permettre à une méthode d'accepter plusieurs types de paramètres en conservant le même nom, comme dans le cas d'opérateur arithmétique travaillant sur les entiers, les réels,...

Exemple de code Delphi :

```
ClasseA = class
  public
     procedure P(x,y : integer);overload;
     procedure P(a,b,c : string);overload;
     procedure P;overload;
end;
   var Obj:ClasseA;
   .....
   Obj := ClasseA.create;
   Obj.P( 10, 5 );
   Obj.P( 'abc', 'ef', 'ghi' );
   Obj.P;
```

2.3 Surcharge statique dans une classe dérivée :

D'une manière générale, Delphi et C# disposent **par défaut** de la notion de méthode statique, Java n'en dispose pas sauf dans le cas des méthodes de classes. Dans l'exemple ci-dessous en Delphi et en C#, les trois méthodes P,Q et R sont à liaison statique dans leur déclaration par défaut sans utiliser de qualificateur spécial.

Delphi	C#

ClasseA = **class**
public
procedure P(x,y : integer);
private
procedure Q(a,b,c : string);
protected
procedure R;
end;
	class ClasseA {
	public void P(**int** x,y){ }
	private void Q(**String** a,b,c){ }
	protected void R( ){ }
	}

Une classe dérivée peut **masquer** une méthode à liaison statique héritée en définissant une nouvelle méthode avec **le même nom**.

Si vous déclarez dans une **classe dérivée**, une méthode ayant le même nom qu'une méthode à liaison statique d'une **classe ancêtre**, la nouvelle méthode **remplace** simplement la méthode héritée **dans la classe dérivée**.

Dans ce cas nous employerons aussi le mot de **masquage** qui semble être utilisé par beaucoup d'auteurs pour dénommer ce remplacement, car il correspond bien à l'idée d'un masquage "local" dans la classe fille du code de la méthode de la classe parent par le code de la méthode fille.

Ci-dessous un exemple de hiérarchie de classes et de masquages successifs licites de méthodes à liaison statiques dans certaines classes dérivées avec ou sans modification de visibilité :

Classe A public statique methode P; privé statique methode Q; protégé statique methode R; **finClasse A**	**Classe B hérite de Classe A** public statique methode P; privé statique methode Q; protégé statique methode R; **finClasse B**	**Classe C hérite de Classe B** protégé statique methode P; privé statique methode Q; **finClasse C**
Classe D hérite de Classe C public statique methode P; **finClasse D**	**Classe E hérite de Classe D** protégé statique methode P; **finClasse E**	**Classe F hérite de Classe E** privé statique methode P; public statique methode R; **finClasse F**

Dans le code d'implémentation de la Classe F :

La **méthode P** utilisée est celle qui définie dans la Classe F et elle masque la **méthode P** de la Classe E.
La **méthode Q** utilisée est celle qui définie dans la Classe C.
La **méthode R** utilisée est celle qui définie dans la Classe F et elle masque la **méthode R** de la Classe B

Soit en Delphi l'écriture des classes ClasseA et ClasseB de la hiérarchie ci-haut :

Delphi	Explications
ClasseA = **class** **public** **procedure** P(x,y : integer); **private** **procedure** Q(a,b,c : string); **protected** **procedure** R; **end;**	Dans la classe ClasseB : La méthode **procedure** P(u : char) surcharge statiquement (masque) avec une autre signature, la méthode héritée de sa classe parent **procedure** P(x,y : integer).

```
ClasseB = class ( ClasseA )
  public
     procedure P(u : char);
  private
     procedure Q(a,b,c : string);
  protected
     procedure R(x,y : real);
end;
```

La méthode **procedure** Q(a,b,c : string) surcharge statiquement (masque) avec la même signature, la méthode héritée de sa classe parent **procedure** Q(a,b,c : string).

La méthode **procedure** R(x,y : real) surcharge statiquement (masque) avec une autre signature, la méthode héritée de sa classe parent **procedure** R.

Utilisation pratique : Possibilité notamment de définir un nouveau comportement lié à la classe descendante et éventuellement de changer le niveau de visibilité de la méthode.

Exemple de code Delphi :

```
ClasseA = class
  public
     procedure P(x,y : integer);
     procedure Q(a,b,c : string);
     procedure R;
end;

ClasseB = class ( ClasseA )
  public
     procedure P(u : char);
     procedure Q(a,b,c : string);
     procedure R(x,y : real);
end;
.......
```

```
.........
    var ObjA:ClasseA;
        ObjB:ClasseB;
    .....
    ObjA := ClasseA.create;
    ObjA.P( 10, 5 );
    ObjA.Q( 'abc', 'ef', 'ghi' );
    ObjA.R;
    .........
    ObjB := ClasseB.create;
    ObjB.P( 'g' );
    ObjB.Q( 'abc', 'ef', 'ghi' );
    ObjB.R( 1.2, -5.36 );
```

2.4 Surcharge dynamique dans une classe dérivée :

Un type dérivé peut **redéfinir** (**surcharger dynamiquement**) une **méthode à liaison dynamique héritée**. On appelle aussi **virtuelle** une telle méthode à liaison dynamique, nous utiliserons donc souvent ce raccourci de notation pour désigner une méthode surchargeable dynamiquement.

L'action de redéfinition fournit une **nouvelle définition de la méthode** qui sera appelée en fonction du type de l'objet **au moment de l'exécution** et **non** du type de la variable de référence connue **au moment de la compilation**.

Ci-dessous un exemple de hiérarchie de classes et de redéfinitions (surcharges dynamiques) successives fictives de méthodes à liaison dynamique dans certaines classes dérivées, pour les modifications de visibilité il faut étudier le manuel de chaque langage :

Classe A public dynamique methode P; privé dynamique methode Q; protégé dynamique methode R; **finClasse A**	**Classe B hérite de Classe A** public dynamique methode P; privé dynamique methode Q; protégé dynamique methode R; **finClasse B**	**Classe C hérite de Classe B** protégé dynamique methode P; privé dynamique methode Q; **finClasse C**
Classe D hérite de Classe C public dynamique methode P; **finClasse D**	**Classe E hérite de Classe D** protégé dynamique methode P; **finClasse E**	**Classe F hérite de Classe E** privé dynamique methode P; public dynamique methode R; **finClasse F**

Remarque pratique :

Une méthode redéfinissant une méthode virtuelle peut selon les langages changer le niveau de visibilité (**il est conseillé de laisser la nouvelle méthode redéfinie au moins aussi visible que la méthode virtuelle parent**).

Tableau comparatif liaison dynamique-statique

Liaison statique	Liaison dynamique
Lors d'un appel pendant l'exécution leur liaison est très rapide car le compilateur a généré l'adresse précise du code de la méthode lors de la compilation.	Lors d'un appel pendant l'exécution leur liaison plus lente car l'adresse précise du code de la méthode est obtenu par un processus de recherche dans une structure de données.
Une telle méthode fonctionne comme une procédure ou fonction d'un langage non orienté objet et ne permet pas le polymorphisme. Car lors d'un appel pendant l'exécution c'est toujours le même code qui est exécuté quel que soit le type de l'objet qui l'invoque.	Une telle méthode autorise le polymorphisme, car bien que portant le même nom dans une hiérarchie de classe, lors d'un appel pendant l'exécution c'est toujours le type de l'objet qui l'invoque qui déclenche le mécanisme de recherche du code adéquat.

2.5 La répartition des méthodes en Delphi

Le terme de **répartition** des méthodes est synonyme de **liaison** et fait référence à la façon dont un programme détermine où il doit rechercher le code d'une méthode lorsqu'il rencontre un appel à cette méthode.

En Delphi, il existe trois modes de répartition des méthodes qui peuvent être :

Statiques ,
Virtuelles,
Dynamiques.

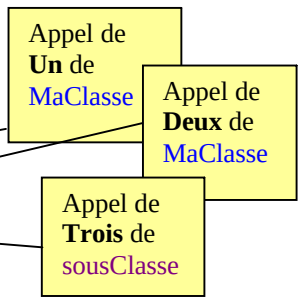
Méthodes statiques en Delphi

Les méthodes statiques de Delphi sont des **méthodes à liaison précoce**.

```
type
 MaClasse=class
    Etat:string;
    procedure Un;     //statique
    procedure Deux;    //statique
 end;

  SousClasse=class(MaClasse)
    procedure Trois;    //statique
 end;
```

Dans SousClas nous avons 3 méthodes statiques :
Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille).

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```



Voyons ce qui se passe lorsque dans la classe fille on tente de "redéfinir" une méthode héritée de la classe mère. Ci-après nous tentons de "redéfinir" la méthode Deux :

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
end;

 SousClasse=class(MaClasse)
   procedure Deux;   //statique
   procedure Trois;  //statique
end;
```

Dans SousClas nous avons 3 méthodes statiques :
Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille).

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```

Appel de **Un** de MaClasse

Appel de **Deux** de MaClasse

Appel de **Trois** de sousClasse

Lors de l'exécution de l'appel **Obj.Deux**, rien n'a changé. En effet lors de la compilation la variable **Obj** est déclarée de type **MaClasse**, c'est donc l'adresse du code de la méthode **Deux** de la classe **MaClasse** qui est liée. Le type **SousClasse** de l'objet réel vers lequel pointe **Obj** pendant l'exécution n'a aucune influence sur le mode de répartition :

Important

- Cela signifie qu'il est impossible de redéfinir une méthode statique P; **c'est toujours le même code** qui est exécuté, quelque soit la classe dans laquelle P est appelée.

- Si l'on déclare dans une classe dérivée une méthode portant le même nom qu'une méthode statique de la classe mère avec la même signature ou bien avec une signature différente, la nouvelle méthode remplace simplement la méthode héritée dans la classe dérivée, nous dirons qu'elle **masque** la méthode mère.

Masquage avec la même signature

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
end;

 SousClasse=class(MaClasse)
   procedure Deux;  // masque la méthode mère
   procedure Trois;  //statique
end;
```

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```

Appel de **Deux** de MaClasse

Masquage avec une signature différente

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
end;

 SousClasse=class(MaClasse)
   procedure Deux ( x : byte ) ;  // masque la méthode mère
   procedure Trois;  //statique
end;
```

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
Obj.Deux ( 49 ) ;
```

Appel de **Deux** de MaClasse

Erreur de compilation ! car Obj est de type MaClasse et la signature **Deux (byte)** n'existe pas dans MaClasse.

Méthodes virtuelles en Delphi

Les méthodes virtuelles utilisent un mécanisme de répartition nécessitant une recherche contrairement aux méthodes statiques. Une méthode virtuelle peut être redéfinie dans les classes descendantes sans masquer ses différentes versions dans les classes ancêtres.

A l'opposé d'une méthode statique l'adresse du code de la méthode virtuelle n'est pas déterminée lors de la compilation, mais seulement lors de l'exécution et en fonction du type de l'objet qui l'appelle.

Les méthodes virtuelles de Delphi sont des **méthodes à liaison tardive**.

Pour déclarer une méthode virtuelle, il faut ajouter le qualificateur **virtual** à la fin de la déclaration de l'en-tête de la méthode :

```
procedure P(x,y : integer); virtual ;
```

Comment se passe la liaison dynamique avec Delphi ?

Delphi implante d'une façon classique le mécanisme de liaison dynamique :

- Lors de la compilation, Delphi rajoute à chaque classe une **T**able des **M**éthodes **V**irtuelles (**TMV**). Cette table contient en principal, pour chaque méthode déclarée avec le qualificateur **virtual** :
  - un pointeur sur le code de la méthode,
  - la taille de l'objet lui-même
- Donc chaque objet possède sa propre **TMV** , et elle est unique.
- La **TMV** d'un objet est créée avec des pointeurs vides lors de la compilation, elle est remplie lors de l'exécution du programme (plus précisément lors de l'instanciation de l'objet) car c'est à l'exécution que l'adresse du code de la méthode est connue et donc stockée dans la **TMV**.
- En fait c'est le constructeur de l'objet lors de son instanciation qui lance le stockage dans la **TMV** des adresses de **toutes** les méthodes virtuelles de l'objet, à la fois les méthodes **héritées** et les méthodes **nouvelles**.

Lorsque l'on construit une nouvelle classe héritant d'une autre classe mère, la nouvelle classe récupère dans sa **TMV** toutes les entrées de la **TMV** de sa classe mère, plus les nouvelles entrées correspondant aux méthodes virtuelles déclarées dans la nouvelle classe. Une **TMV** est donc une structure de données qui grossit au cours de l'héritage de classe et peut être assez volumineuse pour des objets de classes situées en fin de hiérarchie.

Redéfinition de méthode virtuelle avec Delphi

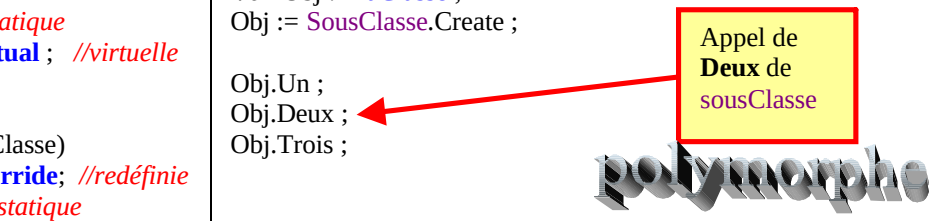
Pour redéfinir une méthode virtuelle dans les classes descendantes sans masquer ses différentes versions dans les classes ancêtres, il faut ajouter à la fin de sa déclaration d'en-tête le qualificateur **override**.

Reprenons l'exemple précédent et tentons de "redéfinir" la méthode **Deux** cette fois en la déclarant virtuelle dans la classe mère et redéfinie dans la classe fille, puis comparons le comportement polymorphique de la méthode **Deux** selon qu'elle est virtuelle ou statique :

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual ;   //virtuelle
end;

 SousClasse=class(MaClasse)
   procedure Deux; override;  //redéfinie
   procedure Trois;    //statique
end;
```

Dans SousClas nous avons 2 méthodes statiques : Un , Trois et une méthode virtuelle redéfinie : Deux

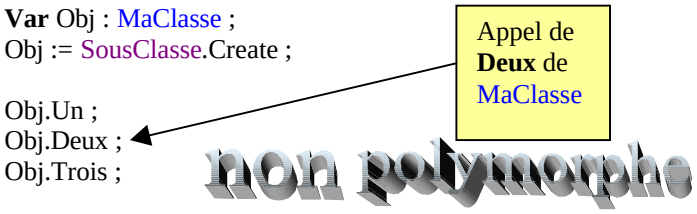
```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```




```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;    //statique
end;

 SousClasse=class(MaClasse)
   procedure Deux;    //statique
   procedure Trois;    //statique
end;
```

Dans SousClas nous avons 3 méthodes statiques :
Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille).

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```



Méthodes dynamiques en Delphi

Les méthodes dynamiques sont des méthodes virtuelles avec un mécanisme de répartition différent, donc les méthodes dynamiques de Delphi sont des **méthodes à liaison tardive**.

technique

Au lieu d'être stockées dans la **T**able des **M**éthodes **V**irtuelles, les méthodes dynamiques sont ajoutées dans une **structure de données de liste** spécifique pour chaque objet (la liste des méthodes dynamiques).

Seules les adresses des méthodes **nouvelles** ou **redéfinies** d'une classe sont rangées dans sa liste.

La liaison précode d'une méthode dynamique héritée s'effectue en recherchant dans la liste des méthodes dynamiques de chaque ancêtre, en remontant la hiérarchie de l'héritage.

Pour déclarer une méthode dynamique, il faut ajouter le qualificateur **dynamic** à la fin de la déclaration de l'en-tête de la méthode :

```
procedure P(x,y : integer); dynamic ;
```

Remarques de Borland

- **Toute méthode** sans qualification **particulière est considérée comme statique** par défaut.
- Comme les méthodes dynamiques ne disposent pas d'entrées dans la table des méthodes virtuelles de l'objet, elles réduisent la quantité de mémoire occupée par les objets.
- Si une méthode est appelée fréquemment, ou si le temps d'exécution est un paramètre important, il vaut mieux déclarer une méthode virtuelle plutôt que dynamique.

Masquage de méthode virtuelle avec Delphi

Pour masquer une méthode virtuelle dans une de ses classes, il suffit de **ne pas** ajouter à la fin de sa déclaration d'en-tête le qualificateur **override**.

Ceci peut se faire de deux façons soit en masquant par une méthode statique, soit en masquant par une méthode dynamique.

Masquage par une méthode dynamique

Dans le code suivant, la méthode Deux déclarée virtuelle dans la classe mère, est masquée par une méthode Deux ayant la même signature et déclarée elle aussi virtuelle dans la classe fille :

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual;     //virtuelle
end;
 SousClasse=class(MaClasse)
   procedure Un ;     //statique, masque la méthode ancêtre
   procedure Deux;virtual;    // virtuelle, masque la méthode ancêtre, mais elle-même est redéfinissable
end;
```

comparons le comportement polymorphique de la méthode Deux selon qu'elle est redéfinie, masquée par une méthode statique ou masquée par une méthode virtuelle :

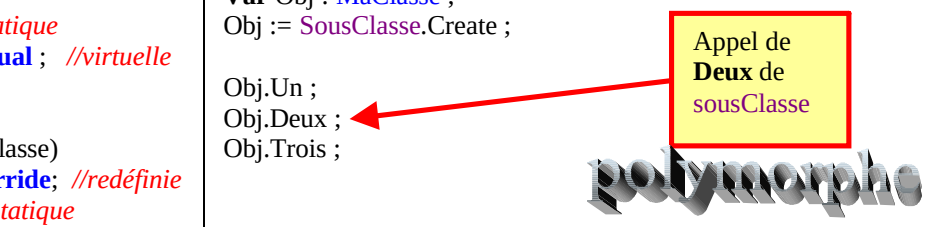
<table>
<tr>
<td>

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual ;   //virtuelle
end;

 SousClasse=class(MaClasse)
   procedure Deux; override;  //redéfinie
   procedure Trois;    //statique
end;
```

</td>
<td>
Dans SousClas nous avons 2 méthodes statiques : Un , Trois
et une méthode virtuelle redéfinie : Deux

Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;



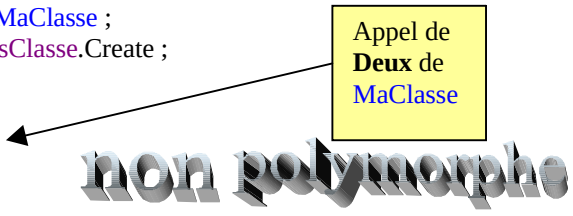
</td>
</tr>
<tr>
<td>

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual ;   //virtuelle
end;

 SousClasse=class(MaClasse)
   procedure Deux;  //masquage
   procedure Trois;    //statique
end;
```

</td>
<td>
Dans SousClas nous avons 2 méthodes statiques : Un , Trois
et une méthode virtuelle masquée statiquement : Deux

Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;



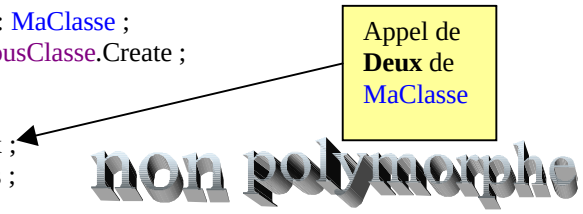
</td>
</tr>
<tr>
<td>

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual ;   //virtuelle
end;

 SousClasse=class(MaClasse)
   procedure Deux; virtual ;  //masquage
   procedure Trois;    //statique
end;
```

</td>
<td>
Dans SousClas nous avons 2 méthodes statiques : Un , Trois
et une méthode virtuelle masquée virtuellemnt: Deux

Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;



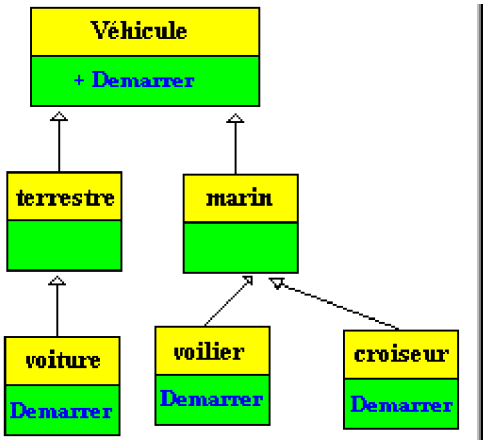
</td>
</tr>
</table>

Nous pouvons conclure de ce tableau de comparaison que le masquage (par une méthode statique ou virtuelle) ne permet jamais le polymorphisme.

Exemple pratique

Le polymorphisme de méthode offre la possibilité de conserver **le même nom de méthode** dans une hiérarchie de classe, afin de ne pas "surcharger" le cerveau de l'utilisateur. Les comportements seront différents selon le type d'objet utilisé.

Par exemple l'action de **Démarrer** dans une hiérarchie de véhicules :



Nous voyons bien que sémantiquement parlant on peut dire qu'une voiture démarre, qu'un voilier démarre, qu'un croiseur démarre, toutefois les actions internes permettant le comportement démarrage ne sont pas les mêmes.

- **Démarrer** dans la classe **voiture** : tourner la clef de contact, engager une vitesse,...
- **Démarrer** dans la classe **voilier** : hisser les voiles, dégager la barre,...
- **Démarrer** dans la classe **croiseur** : lancer les moteurs, modifier la barre,...

L'action Démarrer est polymorphe (car elle s'adapte au type de véhicule qui l'exécute).

Pour traduire en Delphi, ce comportement polymorphe de l'action Démarrer, nous allons utiliser une **méthode virtuelle** que nous **redéfinissons** dans toutes les classes dérivées.

Soit en Delphi l'écriture de l'exemple de la hiérarchie précédente :

Delphi

```
Vehicule = class
  public
     procedure Demarrer; virtual; //virtuelle
end;

Terrestre = class ( Vehicule )
  .....
end;

Voiture = class ( Terrestre )
  public
     procedure Demarrer; override; //redéfinie
end;
```

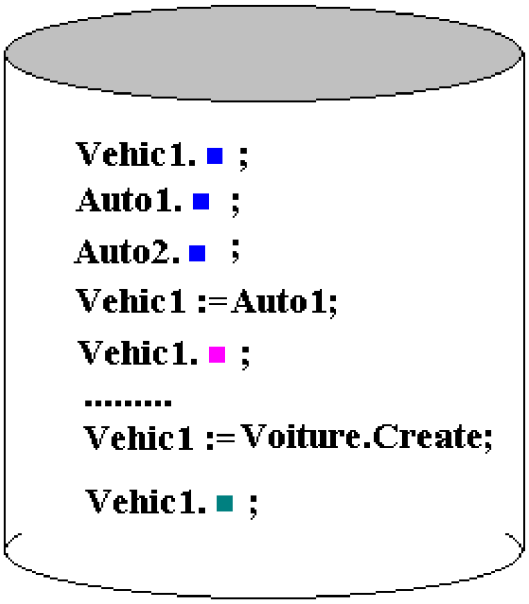
```
Marin = class ( Vehicule )
  .....
end;

Voilier = class ( Marin )
  public
     procedure Demarrer; override; //redéfinie
end;

Croiseur = class ( Marin )
  public
     procedure Demarrer; override; //redéfinie
end;
```

Exemple de code d'utilisation :

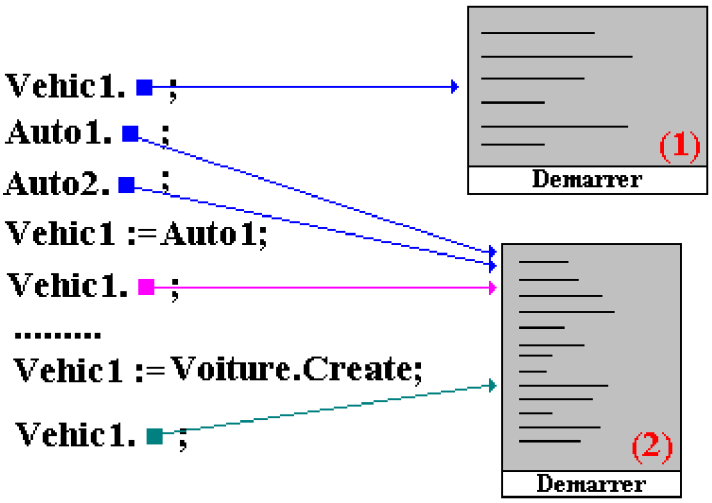
```
Vehicule = class
  public
     procedure Demarrer; virtual; (1)
end;

Voiture = class ( Terrestre )
  public
     procedure Demarrer; override; (2)
end;
.......
var Vehic1 : Vehicule;
      Auto1, Auto2 : Voiture;
```

```
.....
    Vehic1 := Vehicule.Create; //instanciation dans le type
    Auto1 := Voiture.Create; //instanciation dans le type
    Auto2 := Voiture.Create; //instanciation dans le type
    Vehic1.Demarrer; //méthode du type Vehicule (1)
    Auto1.Demarrer; //méthode du type Voiture (2)
    Auto2.Demarrer; //méthode du type Voiture (2)
    Vehic1 := Auto1; //pointe vers un objet de type hérité
    Vehic1.Demarrer; // polymorphisme à l'œuvre ici: (2)
    .........
    Vehic1 := Voiture.Create; //instanciation un type hérité
    Vehic1.Demarrer; // polymorphisme à l'œuvre ici: (2)
```

Illustrons l'exemple précédent avec une image de la partie de code généré sur la méthode Demarrer et ensuite l'exécution de ce code sur des objets effectifs. Nous avons numéroté **(1)** et **(2)** les deux codes d'implantation de la méthode Demarrer dans la classe parent et dans la classe enfant :



Le code engendré contient des références vides



Lors de l'exécution après mécanisme de répartition selon le genre de liaison et le type de l'objet, les références pointent vers le code de la méthode du type effectif de l'objet.

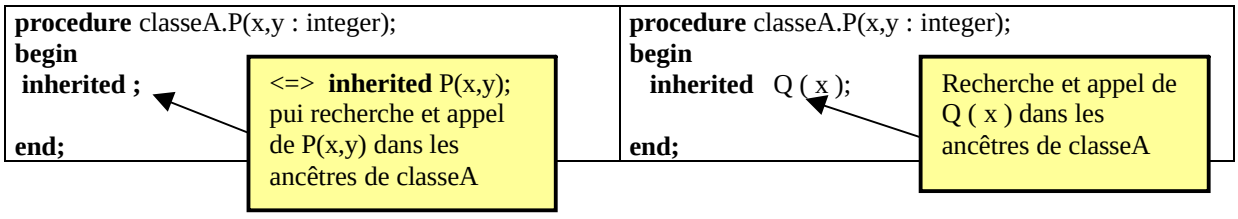
2.6 Réutilisation de méthodes avec inherited

Le mot réservé **inherited** joue dans Delphi un rôle particulier dans l'implémentation de comportements polymorphiques (il joue un rôle semblable à **super** dans java et **base** dans C#).

Il ne peut qu'apparaître dans une définition de méthode avec ou sans identificateur à la suite.

- Si **inherited** présent dans une methode P de la classeA est suivi par le nom d'une méthode Q, il représente un appel normal de la méthode Q, sauf que la recherche de la méthode à invoquer commence dans l'ancêtre immédiat de la classe de la méthode et remonte la hiérarchie.

- Si **inherited** présent dans une methode P de la classeA, n'est suivi d'aucun nom de méthode, il représente alors un appel à une méthode de même nom P, la recherche de la méthode à invoquer commence dans l'ancêtre immédiat de la classe de la méthode P actuelle et remonte la hiérarchie.



3. Polymorphisme de classes abstraites

Introduction

Les classes abstraites permettent de créer des classes génériques **expliquant certains comportements sans les implémenter** et fournissant une implémentation commune de certains autres comportements pour l'héritage de classes. Les classes abstraites sont un outil précieux pour le **polymorphisme**.

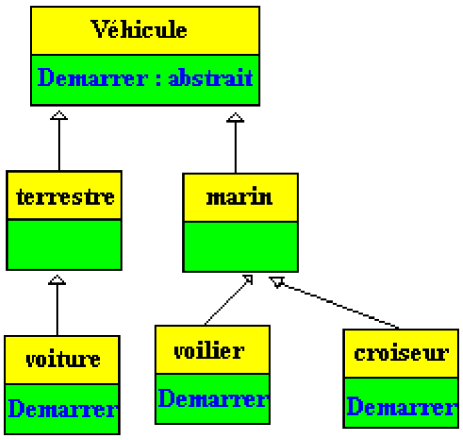
Vocabulaire et concepts :

- Une classe abstraite est une classe qui **ne** peut **pas** être instanciée.
- Une classe abstraite peut contenir des méthodes déjà implémentées.
- Une classe abstraite peut contenir des méthodes **non** implémentées.
- Une classe abstraite est héritable.
- On peut contsruire une hiérarchie de classes abstraites.
- Pour pouvoir construire un objet à partir d'une classe abstraite, il faut dériver une classe non abstraite en une classe implémentant **toutes** les méthodes **non** implémentées.

- Une méthode déclarée dans une classe, **non implémentée** dans cette classe, mais juste définie par la déclaration de sa signature, est dénommée **méthode abstraite**.

- Une méthode abstraite est une méthode à liaison dynamique n'ayant pas d'implémentation dans la classe où elle est déclarée. L' implémentation d'une méthode abstraite est déléguée à une classe dérivée.

Si vous voulez utiliser la notion de classe abstraite pour fournir un comportement polymorphe à un groupe de classes, elles doivent toutes hériter de la même classe abstaite, comme dans l'exemple ci-dessous :

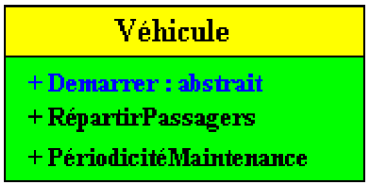


La classe **Véhicule** est abstraite, car la méthode **Démarrer** est abstraite et sert de "modèle" aux futures classes dérivant de **Véhicule**, c'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

Notons au passage que dans la hiérarchie précédente, les classes vehicule **Terrestre** et **Marin** héritent de la classe **Véhicule**, mais n'implémentent pas la méthode abstraite **Démarrer**, ce sont donc par construction des classes abstraites elles aussi.

Les classes abstraites peuvent également **contenir des membres déjà implémentés**. Dans cette éventualité, une classe abstraite propose un certain nombre de **fonctionnalités identiques** pour tous ses futurs descendants (*ceci n'est pas possible avec une interface*).

Exemple : la classe abstraite **Véhicule** n'implémente pas la méthode abstraite **Démarrer**, mais fournit et implante une méthode "**RépartirPassagers**" de répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), elle fournit aussi et implante une méthode "**PériodicitéMaintenance**" renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)



Ce qui signifie que toutes les classes **voiture**, **voilier** et **croiseur** savent comment répartir leurs éventuels passagers et quand effectuer une maintenance, chacune d'elle implémente son propre comportement de démarrage.

Syntaxe de l'exemple en Delphi et en Java :

Delphi	Java

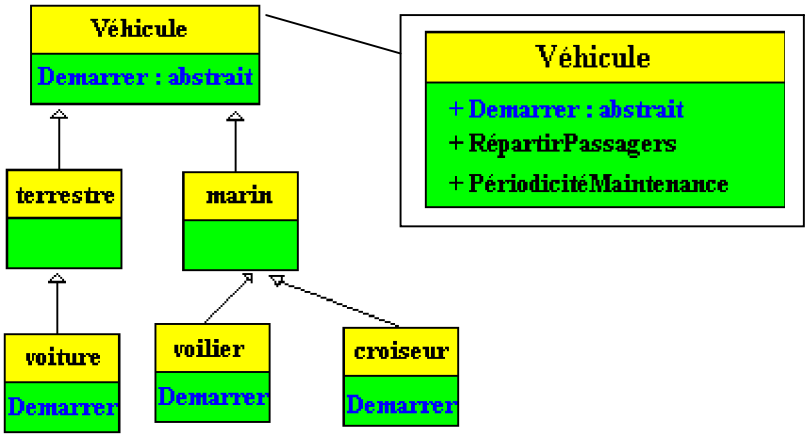
Vehicule = **class**
public
procedure Demarrer; **virtual**;**abstract**;
procedure RépartirPassagers; **virtual**;
procedure PériodicitéMaintenance; **virtual**;
end;
	abstract class ClasseA {
	public abstract void Demarrer( );
	public void RépartirPassagers( );
	public void PériodicitéMaintenance( );
	}

Utilisation pratique des classes abstraites

Utilisez une classe abstraite lorsque vous voulez :

- **regrouper** un ensemble de méthodes présentant des **fonctionnalités identiques**,
- **déléguer** l'implémentation de certaines méthodes à une classe dérivée,
- **disposer** immédiatement de fonctionnalités concrètes pour d'autres méthodes,
- **assurer un comportement polymorphe** aux méthodes dont on délègue l'implantation du code à des classes dérivées.

Exemple de code Delphi pour la hiérarchie ci-dessous :



Soit en Delphi l'écriture d'un exemple tiré de cette hiérarchie :

Delphi

```
Unit UclassVehicules;
    interface
    Vehicule = class
     public
        procedure Demarrer; virtual;abstract;
        procedure RépartirPassagers; virtual;
        procedure PériodicitéMaintenance; virtual;
    end;
    Terrestre = class ( Vehicule )
     public
        procedure PériodicitéMaintenance; override;
    end;
    Voiture = class ( Terrestre )
     public
        procedure Demarrer; override;
        procedure RépartirPassagers; override;
    end;
    Marin = class ( Vehicule )
     public
        procedure PériodicitéMaintenance; override;
    end;
    Voilier = class ( Marin )
     public
        procedure Demarrer; override;
        procedure RépartirPassagers; override;
    end;
    Croiseur = class ( Marin )
     public
        procedure Demarrer; override;
        procedure RépartirPassagers; override;
    end;
    implementation
    //--- les méthodes implantées de la classe abstraite Vehicule :
     procedure Vehicule.RépartirPassagers;
     begin
       .........
     end;
     procedure Vehicule.PériodicitéMaintenance;
     begin
```

```
      .....
    end;
  //--- les méthodes implantées de la classe abstraite Terrestre :
    procedure Terrestre.PériodicitéMaintenance;
    begin
      .....
    end;
  //--- les méthodes implantées de la classe abstraite Marin :
    procedure Marin.PériodicitéMaintenance;
    begin
      .....
    end;
  //--- les méthodes implantées de la classe Voiture :
    procedure Voiture.Demarrer;
    begin
      .....
    end;
    procedure Voiture.RépartirPassagers;
    begin
      .....
    end;
  //--- les méthodes implantées de la classe Voilier :
    procedure Voilier.Demarrer;
    begin
      .....
    end;
    procedure Voilier.RépartirPassagers;
    begin
      .....
    end;
  //--- les méthodes implantées de la classe Croiseur :
    procedure Croiseur.Demarrer;
    begin
      .....
    end;
    procedure Croiseur.RépartirPassagers;
    begin
      .....
    end;
end.
```

Dans cet exemple :

Les classes **Vehicule**, **Marin** et **Terrestre** sont abstraites car aucune n'implémente la méthode abstraite **Demarrer**

Les classes **Marin** et **Terrestre** contiennent chacune une surcharge dynamique implémentée de la méthode virtuelle PériodicitéMaintenance qui est déjà implémentée dans la classe Véhicule.

Les classes **Voiture**, **Voilier** et **Croiseur** ne sont pas abstraites car elles implémentent les (la) méthodes abstraites de leurs parents.

Exercice traité sur le polymorphisme

Objectifs : Comparer le polymorphisme et l'utilisation de tests de type if...then avec l'opérateur is.

On souhaite construire une application utilisant des classes de gestion de structures de données de noms (chaînes). Ces classes devraient être capables de lancer une intialisation de la structure, de trier par ordre croissant les données, de les afficher dans un éditeur de texte. Nous voulons disposer d'une quatrième classe possédant une méthode générale d'édition d'une structure de données après son ordonnancement.

- Au départ nous travaillons sur une classe contenant un tableau, une classe contenant une liste chaînée et une classe contenant un fichier. Chaque classe contient trois méthodes : Tri, Initialiser, et Ecrire qui ne seront qu'esquissées, car c'est la conception de la hiérarchie qui nous intéresse dans cet exemple et non le code interne. Le lecteur peut s'il le souhaite développer un code personnel pour toutes ces méthodes.

LES TYPES DE STRUCTURES DE BASE PROPOSES

Nous proposons de travailler avec les types de données suivants :

type

Element=**record** clef : integer; info : ShortString; **end**;	tableau=**Array**[1..100]**of** Element;
fichier = **file of** Element;	pointeur = ^chainon; chainon=**record** data:Element; suivant:pointeur **end**;

Nous supposons avoir fourni ces informations à deux équipes de développement leur laissant le choix de l'organisation des classes.

- La **première équipe** décide d'implanter les 3 classes à partir de la classe racine (TObject en Delphi).
- La **seconde équipe** de développement a choisi pour le même problème d'implémenter les 3 classes à partir d'une hiérarchie de classes fondée sur une classe abstraite.

CLASSES DESCENDANT DE TObject

Equipe1-1°) L'équipe implémente une gestion de structure de données à partir de classe héritant toutes de la classe racine TObject avec des **méthodes à liaison statique**.

Ttable=**class** T:Tableau; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;	Tliste=**class** L:pointeur; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;	Tfichier=**class** F:fichier; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;

Ce choix permet aux membres de l'équipe n°1 de déclarer et d'instancier des objets dans chaque classe :

```
var
   S1:Ttable;      S2:Tliste;      S3:Tfichier;
```

S1:=Ttable.Create; S1.Initialiser; S1.Tri; S1.Ecrire(Form1.Edit1);	S2:=Tliste.Create; S2.Initialiser; S2.Tri; S2.Ecrire(Form1.Edit1);	S3:=Tfichier.Create; S3.Initialiser; S3.Tri; S3.Ecrire(Form1.Edit1);

L'équipe n°1 pourra utiliser le **polymorphisme d'objet** en déclarant une référence d'objet de **classe racine** puis en l'instanciant selon les besoins dans l'une des trois classes. Il sera alors nécessaire de transtyper la référence en testant auparavant par sécurité, l'appartenance de l'objet référencé à la bonne classe :

```
var
    S1:TObject;
```

if S1 **is** Ttable **then**
begin
Ttable(S1).Initialiser;
Ttable(S1).Tri;
Ttable(S1).Ecrire(Form1.Edit1);
End
	if S1 **is** Tliste **then**
	begin
	Tliste(S1).Initialiser;
	Tliste(S1).Tri;
	Tliste(S1).Ecrire(Form1.Edit1);
	end
		if S1 **is** Tfichier **then**
		begin
		Tfichier(S1).Initialiser;
		Tfichier(S1).Tri;
		Tfichier(S1).Ecrire(Form1.Edit1);
		end

<< S1:=Ttable.Create; >>	<< S1:=Tliste.Create; >>	<< S1:=TFichier.Create; >>

Dans l'application, l'équipe n°1 construit une classe **TUseData** qui possède une méthode générale d'édition d'une structure de données après ordonnancement , cette méthode utilise le polymorphisme d'objet pour s'adapter à la structure de données à éditer :

```
TUseData=class
   Procedure Editer ( x : Tobject );
end ;
```

C'est le paramètre formel de classe générale TObject qui est polymorphe, les valeurs effectives qu'il peut prendre sont : n'importe quel objet de classe héritant de Tobject.

Code de la méthode Editer de TuseData :

```
Procedure TUseData .Editer ( x : Tobject );
Begin
        if  x  is Ttable then
         begin
           Ttable(x).Tri;
           Ttable(x).Ecrire(Form1.Edit1);
         end
Else
        if  x  is Tliste then
         begin
           Tliste(x).Tri;
           Tliste(x).Ecrire(Form1.Edit1);
         end
Else
        if  x  is Tfichier then
         begin
           Tfichier(x)Tri;
           Tfichier(x).Ecrire(Form1.Edit1);
         end
End;
```

Equipe1-2°) L'équipe livre au client l'application contenant en de multiples endroits un appel à la méthode Editer.

Equipe1-3°) Deux mois après la livraison et la mise en place, le client souhaite rajouter une nouvelle structure de données que nous nommons TDatas (*par exemple de structure d'arbre binaire*).

L'équipe propose de poursuivre la même organisation en créant et en implantant une nouvelle classe dérivant de TObject :	TAutre=**class** Data : TDatas; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;

Equipe1-4°) L'équipe 1 doit alors reprendre et modifier le code de la méthode **Procedure** Editer (x : Tobject); de la classe TUseData, en y ajoutant le test du nouveau type TDatas comme suit :

```
Procedure TUseData .Editer ( x : Tobject);
Begin
        if  x  is Ttable then …else
         if  x  is Tliste then … else
          if  x  is Tfichier then … else
           if  x  is TDatas then
            begin
              TDatas (x).Tri;
              TDatas (x).Ecrire(Form1.Edit1);
            end;
End;
```

Cette démarche de rajout et de recompilation, les membres de l'équipe n°1 devront l'accomplir dans chaque partie de l'application qui utilise la méthode Editer.

Equipe1-5°) L'équipe n°1 envoie ensuite au client :

- La nouvelle classe et son code,
- ainsi que la nouvelle version de toutes les parties de l'application qui font appel à la méthode Editer dont la précédente version est maintenant caduque.

CLASSES DESCENDANT DE LA MEME CLASSE ABSTRAITE

Equipe2-1°) L'équipe de développement a choisi pour le même problème d'implémenter une hiérarchie de classes fondée sur une classe **abstraite** qui a été nommée TStructData.

```
TStructData=class
      procedure Tri;virtual;abstract;
      procedure Initialiser;virtual;abstract;
      procedure Ecrire(Ed:Tedit);virtual;abstract;
end ;
```

Toutes les autres classes dériveront de TStructData, et possèderont des méthodes à liaisons dynamiques afin d'utiliser ici le polymorphisme de méthodes :

```
Ttable=class (TStructData)
      T:Tableau;
      procedure Tri;override;
      procedure Initialiser;override;
      procedure Ecrire(Ed:Tedit);override;
end ;
```

```
Tliste=class (TStructData)
      L:pointeur;
      procedure Tri;override;
      procedure Initialiser;override;
      procedure Ecrire(Ed:Tedit);override;
end ;
```

```
Tfichier=class (TStructData)
      F:fichier;
      procedure Tri;override;
      procedure Initialiser;override;
      procedure Ecrire(Ed:Tedit);override;
end ;
```

Ce choix permet aux membres de l'équipe n°2, comme pour ceux de l'équipe n°1, de déclarer et d'instancier des objets dans chaque classe :

```
var
   S1:Ttable;     S2:Tliste;     S3:Tfichier;
```

```
S1:=Ttable.Create;
S1.Initialiser;
S1.Tri;
S1.Ecrire(Form1.Edit1);
```

```
S2:=Tliste.Create;
S2.Initialiser;
S2.Tri;
S2.Ecrire(Form1.Edit1);
```

```
S3:=Tfichier.Create;
S3.Initialiser;
S3.Tri;
S3.Ecrire(Form1.Edit1);
```

L'équipe n°2 pourra utiliser le polymorphisme de méthode en déclarant une référence d'objet de **classe racine abstraite** puis en l'instanciant selon les besoins dans l'une des trois classes. Mais ici, il **ne** sera **pas** nécessaire de transtyper la référence **ni** de tester auparavant par sécurité, l'appartenance de l'objet référencé à la bonne classe. En effet les méthodes Initialiser, Tri et Ecrire étant virtuelles, c'est le type de l'objet lors de l'exécution qui déterminera le bon choix :

```
Var      S1 : TStrucData;
```

<< S1:=Ttable.Create; >> << S1:=Tliste.Create; >> << S1:=TFichier.Create; >>
S1.Initialiser; S1.Tri; S1.Ecrire (Form1.Edit1);

Dans l'application, l'équipe n°2 comme l'équipe n°1, construit une classe **TUseData** qui possède une méthode générale d'édition d'une structure de données après ordonnancement, cette méthode utilise le **polymorphisme d'objet** et le **polymorphisme de méthode** pour s'adapter à la structure de données à éditer :

```
TUseData=class
   Procedure Editer ( x : TstructData );
end ;
```

Méthode Editer de TuseData :

Méthode Editer de TuseData : **équipe n°2**	Méthode Editer de TuseData : **équipe n°1**

Procedure TuseData.Editer (x : TstructData);
Begin
  x.Tri;
  x.Ecrire(Form1.Edit1);
End;
	Procedure TUseData .Editer (x : Tobject);
	Begin
	  if x **is** Ttable **then**
	  begin
	  Ttable(x).Tri;
	  Ttable(x).Ecrire(Form1.Edit1);
	  end else
	  if x **is** Tliste **then**
	  begin
	  Tliste(x).Tri;
	  Tliste(x).Ecrire(Form1.Edit1);
	  end else
	  if x **is** Tfichier **then**
	  begin
	  Tfichier(x)Tri;
	  Tfichier(x).Ecrire(Form1.Edit1);
	  end
	End;

Equipe2-2°) L'équipe livre au client l'application contenant en de multiples endroits un appel à notre méthode Editer.

Equipe2-3°) Deux mois après l'équipe n°2 s'est vu demander comme pour l'autre équipe,

de rajouter une nouvelle structure de données (par exemple de structure d'arbre binaire). L'équipe n°2 crée et implante une nouvelle classe dérivant de la classe abstraite TStructData comme pour les 3 autres classes déjà existantes :

```
TAutre=class (TStructData)
   Data : TDatas;
   procedure Tri;override;
   procedure Initialiser;override;
   procedure Ecrire(Ed:Tedit);override;
end ;
```

Grâce au polymorphisme d'objet et de méthode à liaison dynamique la méthode Editer fonctionne avec toutes les descendants de TstructData.

Equipe2-4°) L'équipe n°2 n'a pas à modifier le code de la méthode **Procedure** Editer (x : TStructData).

Equipe2-5°) Il suffit à l'équipe n°2 d'envoyer au client la nouvelle classe et son code (toutes les parties de l'application qui font appel à la méthode Editer fonctionneront correctement automatiquement) !

Squelettes des méthodes

Les deux équipes ont coopéré entre elles, le code des méthodes est le même quelque soit le choix effectué (hériter de TObject ou hériter d'une classe abstraite).

```
////////////////////// Les tableaux ////////////////////////
 procedure Ttable.Tri;
 begin
  //algorithme de tri d'un tableau
  T[1].info:=T[1].info+' : tableau trié'
 end;

 procedure Ttable.Initialiser;
 begin
  T[1].clef:=100;
  T[1].info:='Durand';//etc...
 end;

 procedure Ttable.Ecrire(Ed:Tedit);
 begin
  Ed.Text:='Clef= '+inttostr(T[1].clef)+'//'+T[1].info
 end;

////////////////////// Les listes chaînées ////////////////////////
 procedure Tliste.Tri;
 begin
  //algorithme de tri d'une liste ...
  L.data.info:=L.data.info+' : liste triée'
 end;

 procedure Tliste.Initialiser;
 begin
```

```
 new(L);
 L.data.clef:=100;
 L.data.info:='Durand';
 L.suivant:=nil  //etc...
end;

procedure Tliste.Ecrire(Ed:Tedit);
begin
 Ed.Text:='Clef= '+inttostr(L.data.clef)+'//'+L.data.info
end;

////////////////////// Les fichiers ////////////////////////
procedure Tfichier.Tri;
var UnElement:Element;
begin
 //algorithme de tri d'un fichier ...
 AssignFile(F,'FichLocal');
 reset(F);
 read(F,UnElement);
 CloseFile(F);
 UnElement.info:=UnElement.info+' : fichier trié';
 reset(F);
 write(F,UnElement);
 CloseFile(F)
end;

procedure Tfichier.Initialiser;
var UnElement:Element;
begin
 AssignFile(F,'FichLocal');
 rewrite(F);
 UnElement.clef:=100;
 UnElement.info:='Durand';
 write(F,UnElement); //etc...
 CloseFile(F)
end;

procedure Tfichier.Ecrire(Ed:Tedit);
var UnElement:Element;
begin
 AssignFile(F,'FichLocal');
 reset(F);
 read(F,UnElement);
 Ed.Text:='Clef= '+inttostr(UnElement.clef)+'//'+UnElement.info;
 CloseFile(F);
end;

end.
```

Conclusion

Le polymorphisme a montré dans cet exemple ses extraordinaires facultés d'adaptation et de réutilisation. Nous avons constaté le gain en effort de développement obtenu par l'équipe qui a choisi de réfléchir à la construction d'une hiérarchie fondée sur une classe abstraite. Nous conseillons donc de penser à la notion de **classe abstraite** et aux **méthodes virtuelles** lors de la programmation d'un problème avec des classes.

5.4 Programmation événementielle et visuelle

Plan du chapitre:

Introduction

Programmation visuelle basée sur les pictogrammes
Programmation orientée événements
Normalisation du graphe événementiel

le graphe événementiel *arcs et sommets*
les diagrammes d'états UML *réduits*

Tableau des actions événementielles
Interfaces liées à un graphe événementiel
Avantages et modèle de développement RAD visuel
  le modèle de la spirale *(B.Boehm)*
  le modèle incrémental

1. Programmation visuelle basée sur les pictogrammes

Le développement visuel rapide d'application est fondé sur le concept de programmation visuelle associée à la montée en puissance de l'utilisation des Interactions Homme-Machine (IHM) dont le dynamisme récent ne peut pas être méconnu surtout par le débutant. En informatique les systèmes MacOs, Windows, les navigateurs Web, sont les principaux acteurs de l'ingénierie de l'IHM. Actuellement dans le développement d'un logiciel, un temps très important est consacré à l'ergonomie et la communication, cette part ne pourra que grandir dans un avenir proche; car les utilisateurs veulent s'adresser à des logiciels efficaces (ce qui va de soi) mais aussi conviviaux et faciles d'accès.

Les développeurs ont donc besoin d'avoir à leur disposition des produits de développement adaptés aux nécessités du moment. A ce jour la programmation visuelle est une des réponses à cette attente des développeurs.

La programmation visuelle au tout début a été conçue pour des personnes n'étant pas des programmeurs en basant ses outils sur des manipulations de pictogrammes.
Le raisonnement communément admis est qu'un dessin associé à une action élémentaire est plus porteur de sens qu'une phrase de texte.

A titre d'exemple ci-dessous l'on enlève le "fichier.bmp" afin de l'effacer selon deux modes de communication avec la machine: utilisation d'icônes ou entrée d'une commande textuelle.

Effacement avec un langage d'action visuelle (souris)

Action :	**Réponse :**
	

Effacement avec un langage textuel (clavier)

Action :	**Réponse :**
del c:\Exemple\Fichier.bmp	| ?

Nous remarquons donc déjà que l'interface de communication MacOs, Windows dénommée "bureau électronique" est en fait un outil de programmation de commandes systèmes.

Un langage de programmation visuelle permet "d'écrire" la partie communication d'un programme uniquement avec des dessins, diagrammes, icônes etc... Nous nous intéressons aux systèmes RAD (Rapid Application Development) visuels, qui sont fondés sur des langages objets à bases d'icônes ou pictogrammes. Visual Basic de MicroSoft est le premier RAD visuel a avoir été commercialisé dès 1991, il est fondé sur un langage Basic étendu incluant des objets étendus en VB.Net depuis 2001, puis dès 1995 Delphi le premier RAD visuel de Borland fondé sur Pascal objet, puis actuellement toujours de Borland : C++Builder RAD visuel fondé sur le langage C++ et Jbuilder, NetBeans RAD visuel de Sun fondés sur le langage Java, Visual C++, Visual J++ de Microsoft, Visual C# etc...

Le développeur trouve actuellement, une offre importante en outil de développement de RAD visuel y compris en open source. Nous proposons de définir un langage de RAD visuel ainsi :

Un **langage visuel** dans un RAD visuel est un **générateur de code source** du langage de base qui, derrière chaque action visuelle (dépôt de contrôle, click de souris, modifications des propriétés, etc...) engendre des lignes de code automatiquement et d'un manière transparente au développeur.

Des outils de développement tels que Visual Basic ou Delphi sont adaptés à la programmation visuelle pour débutant. Toutefois l'efficacité des dernières versions depuis 3 ans a étendu leur champ au développement en général et dans l'activité industrielle et commerciale avec des versions "entreprise" pour VB et "Architect "pour Delphi.

En outre, le système windows est le plus largement répandu sur les machines grand public (90% des PC vendus en sont équipés), il est donc très utile que le débutant en programmation sache utiliser un produit de développement (rapide si possible) sur ce système.

Proposition :

Nous considérons dans cet ouvrage, la programmation visuelle à la fois comme une **fin** et comme un **moyen**.

La programmation visuelle est sous-tendue par la réactivité des programmes en réponse aux actions de l'utilisateur. Il est donc nécessaire de construire des programmes qui répondent à des sollicitations externes ou internes et non plus de programmer séquentiellement (ceci est essentiellement dû aux architectures de Von Neumann des machines) : ces sollicitations sont appelées des événements.

Le concept de **programmation dirigée ou orientée par les événements** est donc la composante essentielle de la programmation visuelle.

Terminons cette présentation par 5 remarques sur le concept de RAD :

Utiliser un RAD simple mais puissant

Nous ne considérerons pas comme utile pour des débutants de démarrer la programmtion visuelle avec des RAD basés sur le **langage C++**. Du fait de sa large permissivité ce langage permet au programmeur d'adopter certaines **attitudes dangereuses** sans contrôle possible. Seul le programmeur confirmé au courant des pièges et des subtilités de la programmation et du langage, pourra exploiter sans risque la richesse de ce type de RAD.

Avoir de bonnes méthodes dès le début

Le RAD Delphi de Borland conçu en 1995 est une extension du langage Object Pascal, qui a des **caractéristiques très proches de celles de C++** sans en avoir les **inconvénients**. L'aspect fortement typé du langage pascal autorise la prise en compte par le développeur débutant de bonnes attitudes de programmation.

C'est Apple qui en a été le promoteur

Le premier environnement de développement visuel professionnel basé sur **Object Pascal** a été conçu par Apple pour le système d'exploitation **MacOs**, sous la dénomination de **MacApp** en 1986. Cet environnement objet visuel permettait de développer des applications MacIntosh avec souris, fenêtre, menus déroulants etc...

Microsoft l'a popularisé

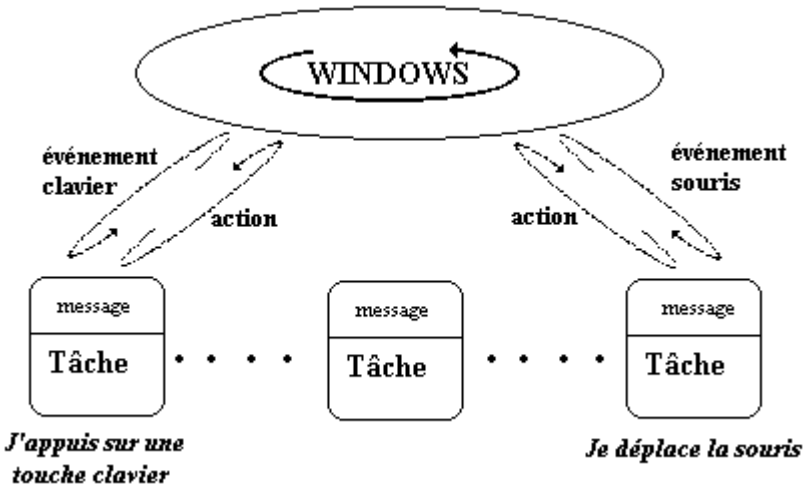
Le RAD Visual Basic de MicroSof conçu à partir de 1992, basé sur le langage Basic avait pour objectif le développement de petits logiciels sous Windows par des **programmeurs non expérimentés et occasionnels**. Actuellement il se décline en VB.Net un langage totalement orienté objet faisant partie intégrante de la plate-forme .Net Framework de Microsoft.

Le concept de RAD a de beaux jours devant lui

Le métier de développeur devrait à terme, consister grâce à des outils tels que les RAD visuels, à prendre un "caddie" et à aller dans un supermarché de composants logiciels génériques adaptés à son problème. Il ne lui resterait plus qu'à assembler le flot des événements reliant entre eux ces logiciels en kit.

2. Programmation orientée événements

Sous les versions actuelles de Windows, système multi-tâches préemptif sur micro-ordinateur, les concepts quant à la programmation par événement restent sensiblement les mêmes que sous les anciennes versions.



Nous dirons que le système d'exploitation passe l'essentiel de son " temps " à attendre une action de l'utilisateur (événement). Cette action déclenche un message que le système traite et envoie éventuellement à une application donnée.

Une définition de la programmation orientée événements

Logique de conception selon laquelle un programme est construit avec des objets et leurs propriétés et d'après laquelle les interventions de l'utilisateur sur les objets du programme déclenchent l'exécution des routines associées.

Par la suite, nous allons voir dans ce chapitre que la programmation d'une application " windows-like " est essentiellement une programmation par événements associée à une programmation classique.

Nous pourrons construire un logiciel qui réagira sur les interventions de l'utilisateur si nous arrivons à intercepter dans notre application les messages que le système envoie. Or l'environnement RAD (Delphi , comme d'ailleurs avant lui Visual Basic de Microsoft), autorise la consultation de tels messages d'un façon simple et souple.

Deux approches pour construire un programme

- *L'**approche événementielle*** intervient principalement dans l'interface entre le logiciel et l'utilisateur, mais aussi dans la liaison dynamique du logiciel avec le système, et enfin dans la sécurité.
- *L'**approche visuelle*** nous aide et simplifie notre tâche dans la construction du dialogue homme-machine.
- La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

Il est possible de relier certains objets entre eux par des relations événementielles. Nous les représenterons par un graphe (structure classique utilisée pour représenter des relations). Lorsque l'on utilise un système multi-fenêtré du genre windows, l'on dispose du clavier et de la souris pour agir sur le système. En utilisant un RAD visuel, il est possible de **construire un logiciel qui se comporte comme le système sur lequel il s'exécute**. L'intérêt est que l'utilisateur aura moins d'efforts à accomplir pour se servir du programme puisqu'il aura des fonctionnalités semblables au système. Le fait que l'utilisateur reste dans un **environnement familier** au niveau de la manipulation et du confort du dialogue, assure le logiciel d'un capital confiance de départ non négligeable.

3. Normalisation du graphe événementiel

Il n'existe que peu d'éléments accessibles aux débutants sur la programmation orientée objet par événements. Nous construisons une démarche méthodique pour le débutant, en partant de remarques simples que nous décrivons sous forme de schémas dérivés des diagrammes d'états d'UML. Ces schémas seront utiles pour nous aider à décrire et à implanter des relations événementielles en Delphi ou dans un autre RAD événementiel.

Voici deux principes qui pour l'instant seront suffisants à nos activités de programmation.

Dans une interface windows-like nous savons que:

- Certains événements déclenchent immédiatement des actions comme par exemple des appels de routines système.
- D'autres événements ne déclenchent pas d'actions apparentes mais activent ou désactivent certains autres événements système.

Nous allons utiliser ces deux principes pour conduire notre programmation par événements.

Nous commencerons par **le concept d'activation et de désactivation**, les autres événements fonctionneront selon les mêmes bases. Dans un enseignement sur la programmation événementielle, nous avons constaté que ce **concept était suffisant** pour que les étudiants comprennent les fondamentaux de l'approche événementielle.

Remarque :

Attention! Il ne s'agit que d'une manière particulière de conduire notre programmation, ce ne peut donc être ni la seule, ni la meilleure (le sens accordé au mot meilleur est relatif au domaine pédagogique). Cette démarche s'est révélée être fructueuse lors d'enseignements d'initiation à ce genre de programmation.

Hypothèses de construction

Nous supposons donc que lorsque l'utilisateur intervient sur le programme en cours d'exécution, ce dernier réagira en première analyse de deux manières possibles :

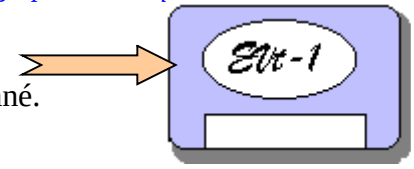
- soit il lancera l'appel d'un routine (exécution d'une action, calcul, lecture de fichier, message à un autre objet comme ouverture d'une fiche etc...),
- soit il modifiera l'état d'activation d'autres objets du programme et/ou de lui-même, soit il ne se passera rien, nous dirons alors qu'il s'agit d'une modification nulle.

Ces hypothèses sont largement suffisantes pour la plupart des logiciels que nous pouvons raisonnablement espérer construire en initiation. Les concepts plus techniques de messages dépassent assez vite l'étudiant qui risque de replonger dans de " la grande bidouille ".

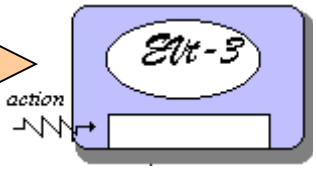
3.1 le graphe événementiel arcs et sommets

Nous proposons de construire un graphe dans lequel :

chaque **sommet** est un objet sensible à un événement donné.

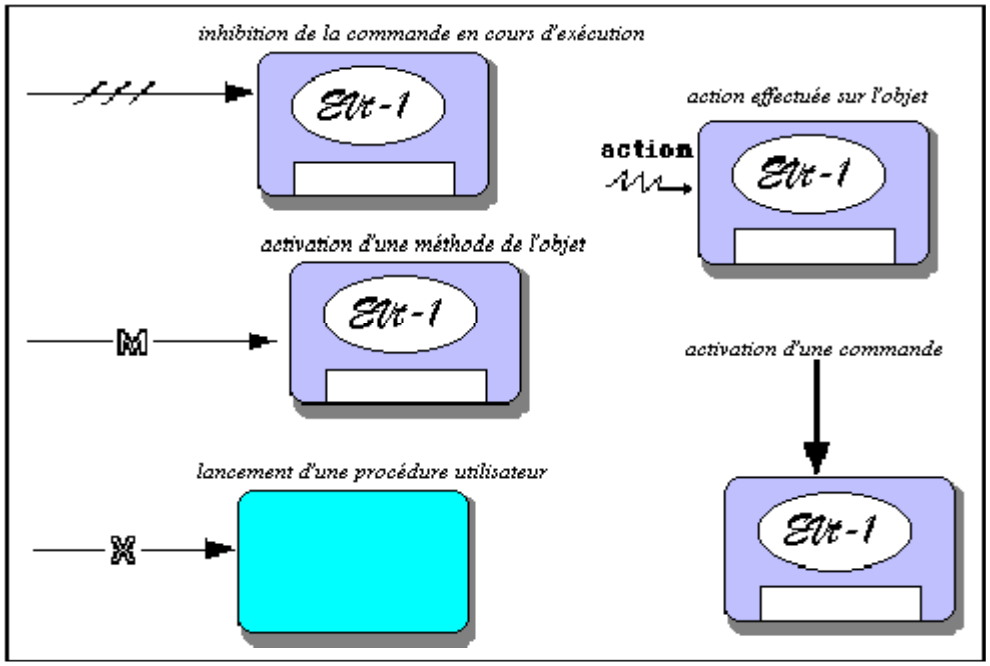


L'événement donné est déclenché par une action extérieure à l'objet.

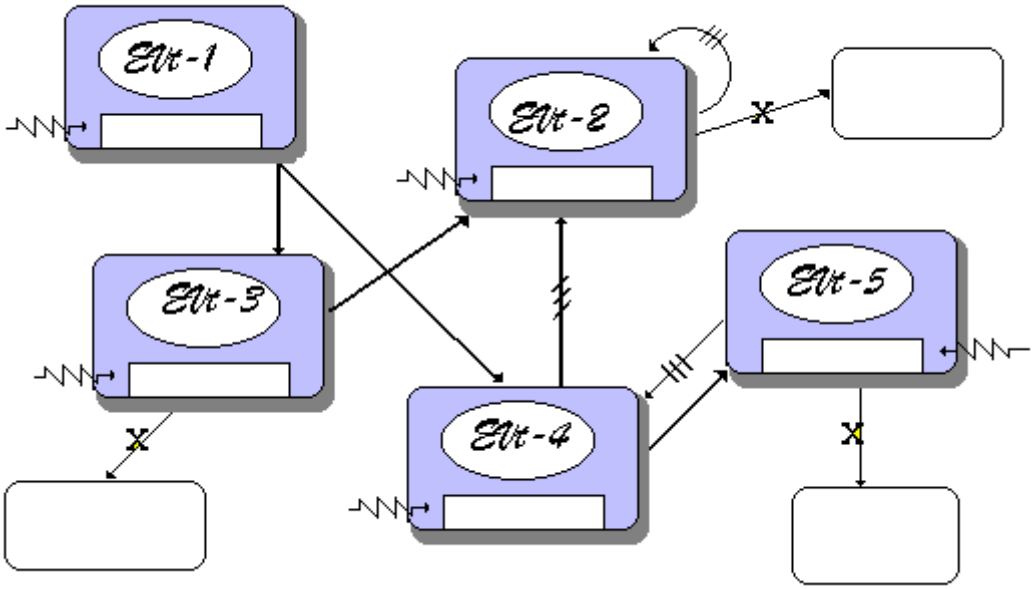


Les arcs du graphe représentent des actions lancées par un sommet.

Les actions sont de 4 types

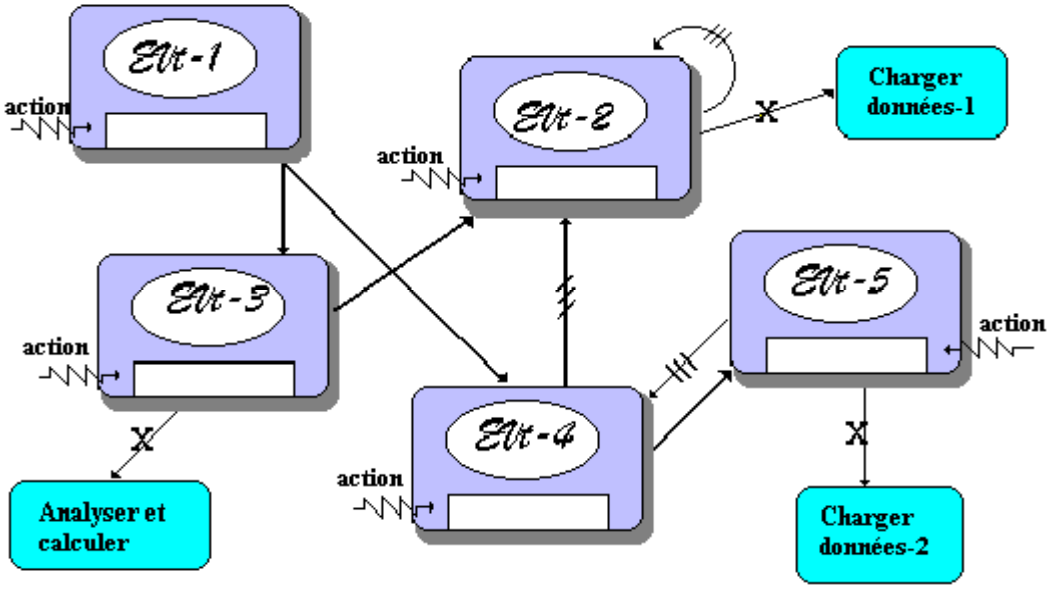


Soit le graphe événementiel suivant composé de 5 objets sensibles chacun à un événement particulier dénoté Evt-1,..., Evt-5; ce graphe comporte des réactions de chaque objet à l'événement auquel il est sensible :



Imaginons que ce graphe corresponde à une analyse de chargements de deux types différents de données à des fins de calcul sur leurs valeurs.

La figure suivante propose un tel graphe événementiel à partir du graphe vide précédent.



Cette notation de graphe événementiel est destinée à s'initier à la pratique de la description d'au maximum 4 types de réactions d'un objet sur la sollicitation d'un seul événement.

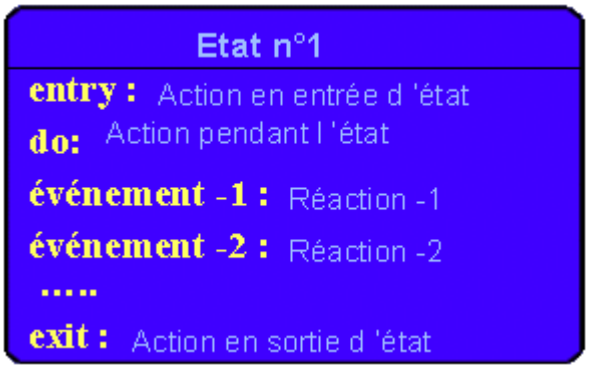
Remarques :

- L'arc nommé, représentant l'activation d'une méthode correspond très exactement à la notation UML de l'envoi d'un message.

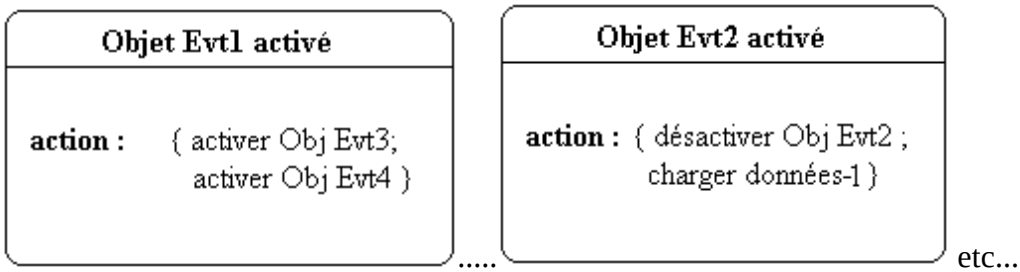
- Lorsque nous voudrons représenter d'une manière plus complète d'autres réactions d'**un seul** objet **à plusieurs événements différents**, nous pourrons utiliser la notation UML réduite de diagramme d'état pour un objet (réduite parce qu'un objet **visuel** ne prendra pour nous, que 2 états: activé ou désactivé).

3.2 les diagrammes d'états UML réduits

Nous livrons ci-dessous la notation générale de diagramme d'état en UML, les cas particuliers et les détails complets sont décrits dans le document de spécification d'UML.



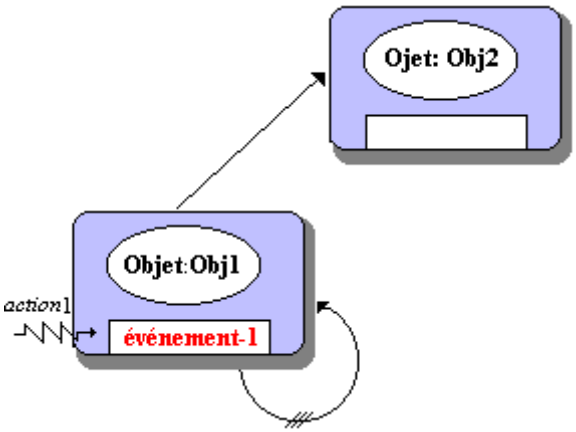
Voici les diagrammes d'états réduits extraits du graphe événementiel précédent, pour les objets **Evt-1** et **Evt-2** :



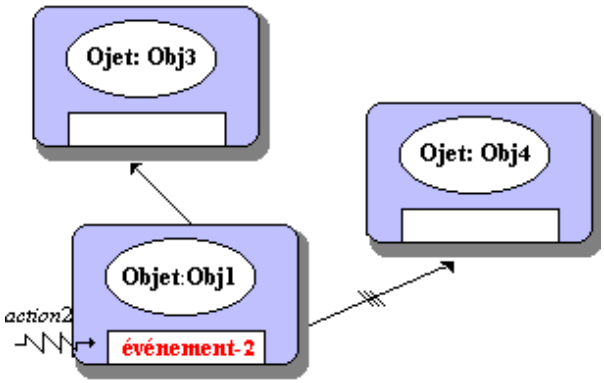
..... etc...

Nous remarquerons que cette écriture, pour l'instant, ne produit pas plus de sens que le graphe précédent qui comporte en sus la vision globale des interrelations entre les objets.
Ces diagrammes d'états réduits deviennent plus intéressants lorsque nous voulons exprimer le fait par exemple, qu'un seul objet **Obj1** réagit à 3 événements (événement-1, événement-2, événement-3). Dans ce cas décrivons les portions de graphe événementiel associés à chacun des événements :

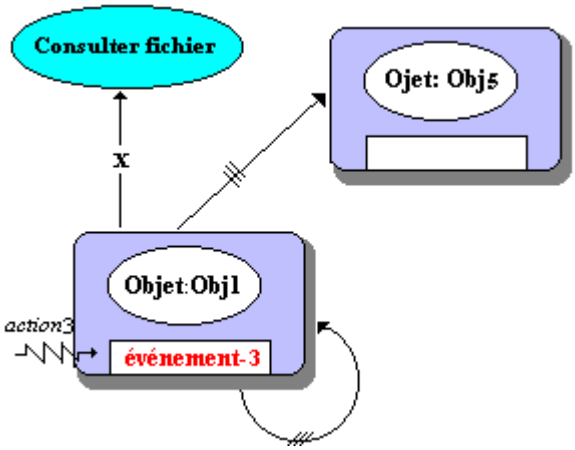
Réaction de obj1 à l'événement-1 :



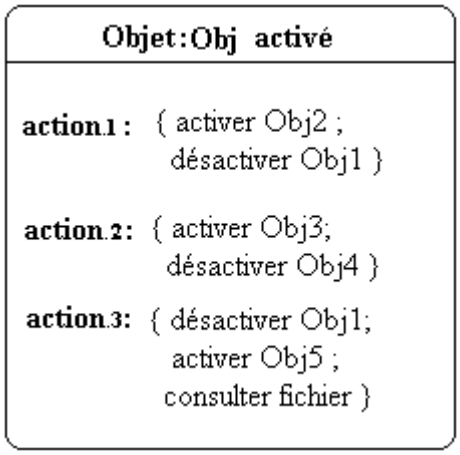
Réaction de obj1 à l'événement-2 :



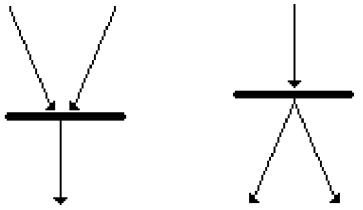
Réaction de obj1 à l'événement-3 :



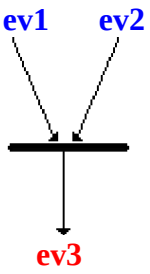
Synthétisons dans un diagramme d'état réduit les réactions à ces 3 événements :



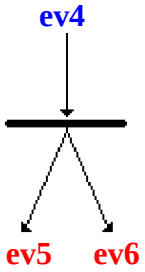
Lorsque nous jugerons nécessaire à la compréhension de relations événementielles dans un logiciel visuel, nous pourrons donc utiliser ce genre de diagramme pour renforcer la sémantique de conception des objets visuels. La notation UML sur les diagrammes d'états comprend les notions d'état de départ, de sortie, imbriqué, historisé, concurrents...
Lorsque cela sera nécessaire nous utiliserons la notation UML de synchronisation d'événements :



Dans le premier cas la notation représente la conjonction des deux événements **ev1** et **ev2** qui déclenche l'événement **ev3**.



Dans le second cas la notation représente l'événement **ev4** déclenchant conjointement les deux événements **ev5** et **ev6**.



4. Tableau des actions événementielles

L'exemple de graphe événementiel précédent correspond à une application qui serait sensible à 5 événements notés EVT-1 à EVT-5, et qui exécuterait 3 procédures utilisateur. Nous

notons dans un tableau (nommé " ***tableau des actions événementielles*** ")les résultats obtenus par analyse du graphe précédent, événement par événement..

EVT-1	EVT-3 activable EVT-4 activable
EVT-2	Appel de procédure utilisateur "chargement-1" désactivation de l' événement EVT-2
EVT-3	Appel de procédure utilisateur "Analyser" EVT-2 activable
EVT-4	EVT-2 désactivé EVT-5 activable
EVT-5	EVT-4 désactivé immédiatement Appel de procédure utilisateur "chargement-2"

Nous adjoignons à ce tableau une table des états des événements dès le lancement du logiciel (elle correspond à l'état initial des objets). Par exemple ici :

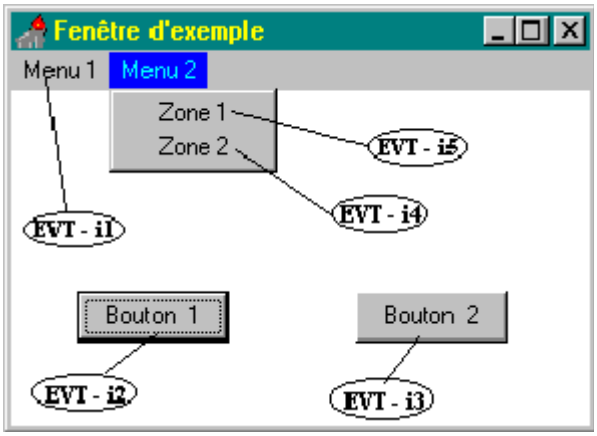
Evt1	activé
Evt2	désactivé
Evt3	activé
Evt4	activé
Evt5	désactivé

etc...

5. Interfaces liées à un graphe événementiel

construction d'interfaces liées au graphe précédent

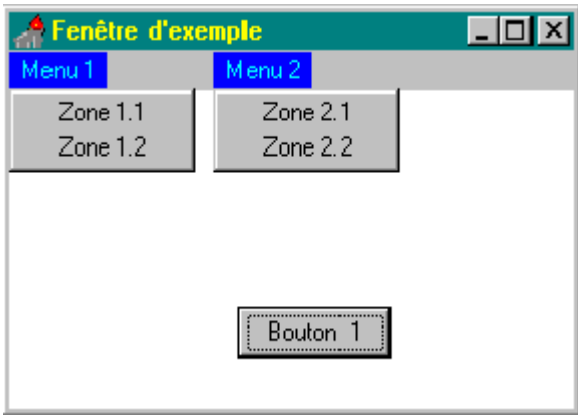
Interface n°1



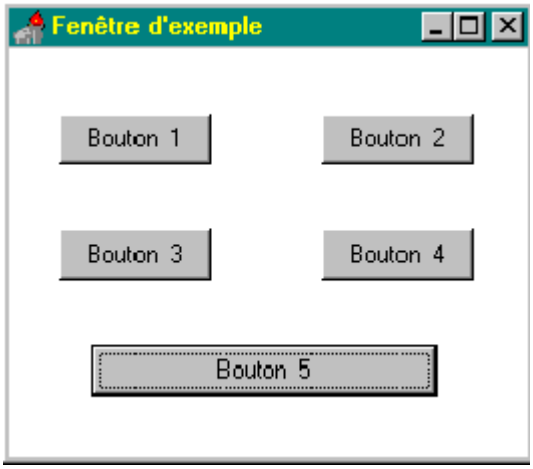
Dans l'exemple de droite, (i1,i2,i3,i4,i5)est une permutation sur (1,2,3,4,5) .

Ce qui nous donne déjà 5!=120 interfaces possibles avec ces objets et uniquement avec cette topologie.

La figure précédente montre une IHM (construite avec des objets Delphi) à partir du graphe événementiel étudié plus haut. Ci-dessous deux autres structures d'interfaces possibles avec les mêmes objets combinés différemment et associés au même graphe événementiel :



Interface n°2



Interface n°3

Pour les choix n°2 et n°3, il y a aussi 120 interfaces possibles…

6. Avantages du modèle de développement RAD visuel

L'approche uniquement structurée (privilégiant les fonctions du logiciel) impose d'écrire du code long et compliqué en risquant de ne pas aboutir, car il faut tout tester afin d'assurer un bon fonctionnement de tous les éléments du logiciel.

L'approche événementielle préfère bâtir un logiciel fondé sur une construction graduelle en fonction des besoins de communication entre l'humain et la machine. Dans cette optique, le programmeur élabore les fonctions associées à une action de communication en privilégiant le dialogue. Ainsi les actions internes du logiciel sont subordonnées au flux du dialogue.

Avantages liés à la programmation par RAD visuel

- Il est possible de construire très rapidement un prototype.
- Les fonctionnalités de communication sont les guides principaux du développement (approche plus vivante et attrayante).
- L'étudiant est impliqué immédiatement dans le processus de conception - construction.

L'étudiant acquiert très vite comme naturelle l'attitude de réutilisation en se servant de " logiciels en kit " (soit au début des composants visuels ou non, puis par la suite ses propres composants).

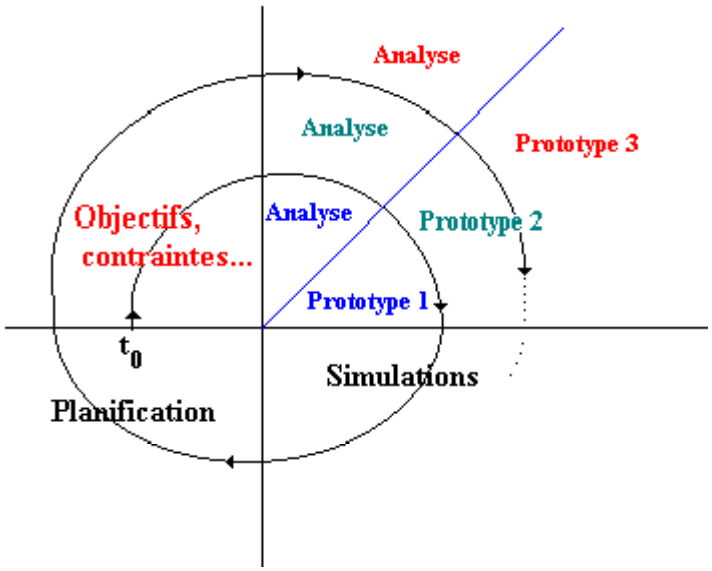
Il n'y a pas de conflit ni d'incohérence avec la démarche structurée : les algorithmes étant conçus comme des boîtes noires permettant d'implanter certaines actions, seront réutilisés immédiatement.
La méthodologie objet de COO et de POO reste un facteur d'intégration général des activités du programmeur.

Les actions de communications classiques sont assurées immédiatement par des objets standards (composants visuels ou non) réagissant à des événements extérieurs.
Le RAD fournira des classes d'objets standards non visuels (extensibles si l'étudiant augmente sa compétence) gérant les structures de données classiques (Liste, arbre, etc..).
L'extensibilité permet à l'enseignant de rajouter ses kits personnels d'objets et de les mettre à la disposition des étudiants comme des outils standards.

Modèles de développement avec un RAD visuel objet

Le développement avec ce genre de produit autorise une logique générale articulée sur la combinaison de deux modèles de développement :

le modèle de la spirale (B.Boehm) en version simplifiée

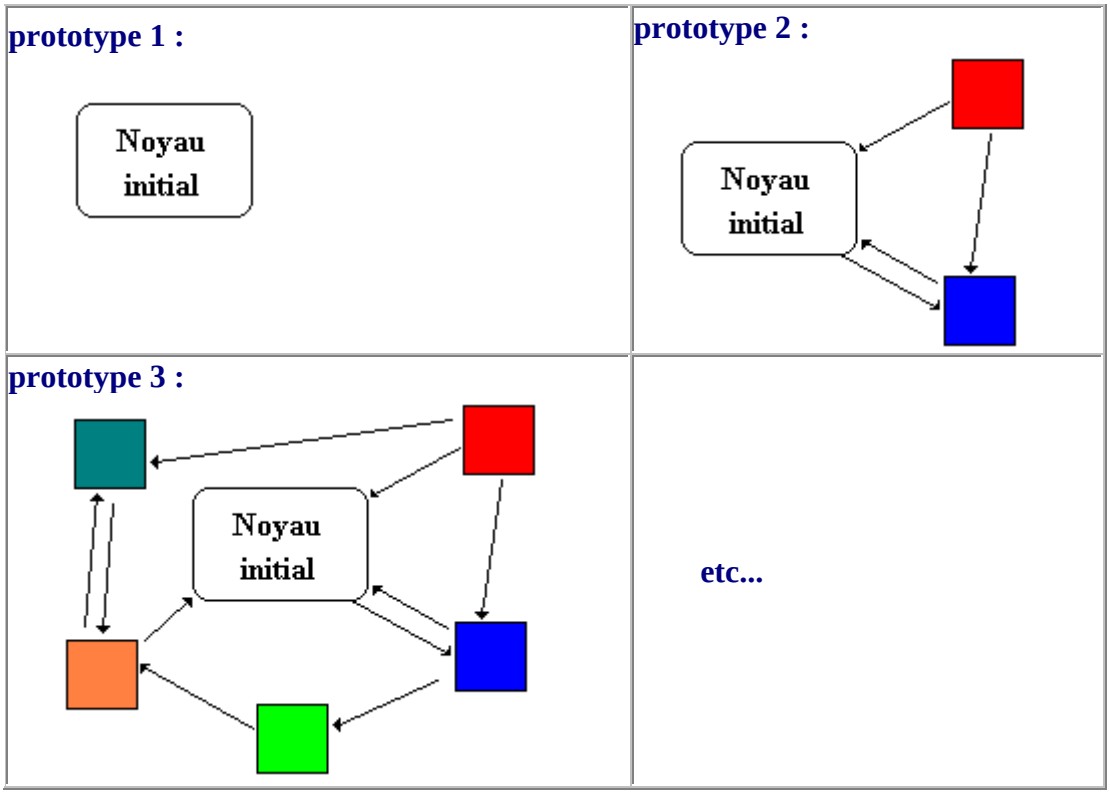


Dans le modèle de la spirale la programmation exploratoire est utilisée sous forme de prototypes simplifiés cycle après cycle. L'analyse s'améliore au cours de chaque cycle et fixe le type de développement pour ce tour de spirale.

le modèle incrémental

Il permet de réaliser chaque prototype avec un bloc central au départ, s'enrichissant à chaque phase de nouveaux composants et ainsi que de leurs interactions.

Associé à un cycle de prototypage dans la spirale, une seule famille de composants est développée pour un cycle fixé.



Ce modèle de développement à l'aide d'objets visuels ou non, fournira en fin de parcours un prototype opérationnel qui pourra s'intégrer dans un projet plus général.

Nous verrons sur des exemples comment ce type d'outil peut procurer aussi des avantages au niveau de la programmation défensive.

5.5 Les événements avec Delphi

Plan du chapitre:

Programmes événementiels

1. Programmes événementiels avec Delphi

Delphi comme les autres RAD événementiels permet de construire du code qui est exécuté en réponse à des événements. Un événement Delphi est une propriété d'un type spécial que nous allons examiner plus loin.

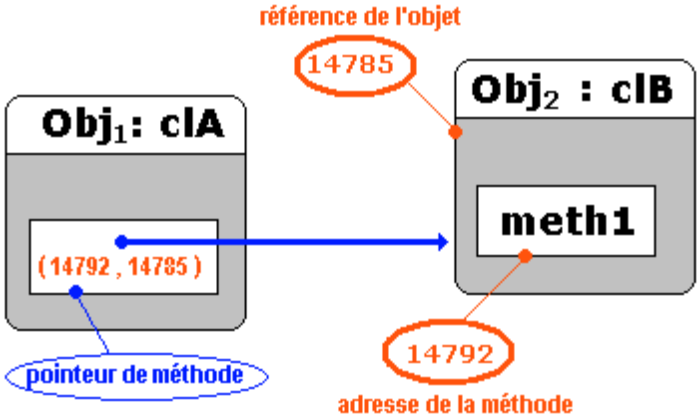
Le code de réaction à un événement particulier est une méthode qui s'appelle un *gestionnaire de cet événement*.

Un événement est en fait un pointeur vers la méthode qui est chargée de le gérer. Définissons la notion de pointeur de méthode qui est utilisée ici, notion largement utilisée en général dans les langages objets.

Pointeur de méthode

Un **pointeur de méthode** est **une paire de pointeurs** (adresses mémoire), le premier contient l'**adresse d'une méthode** et le **second une référence à l'objet** auquel appartient la méthode.

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ du type pointeur vers la méthode **meth1** de l'objet **Obj2** de classe **clB** :



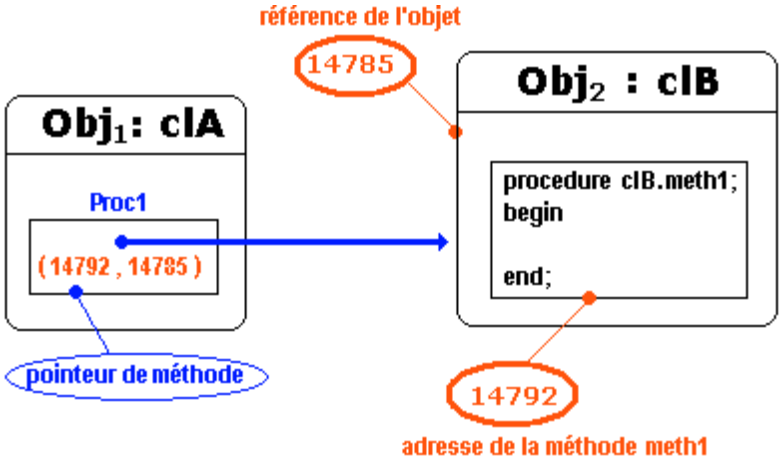
Pointeur de méthode en Delphi

Pour pointer la méthode d'une instance d'objet en Delphi, nous devons déclarer un nouveau type auquel nous ajoutons à la fin le qualificateur **of object** .

Exemples de types pointeurs de méthode et de variable de type pointeur de méthode :

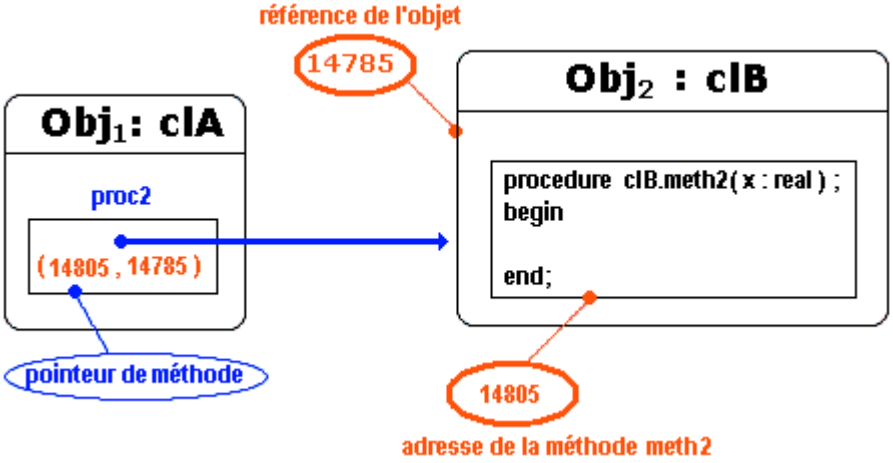
```
type

  ptrMethode1 = procedure of object;
  ptrMethode2 = procedure (x : real) of object;
  ptrMethode3 = procedure (x,y: integer; var z:char) of object;

var
    Proc1 : ptrMethode1;  // pointeur vers une méthode sans paramètre
    Proc2 : ptrMethode2; // pointeur vers une méthode à un  paramètre real
    Proc3 : ptrMethode3; // pointeur vers une méthode à trois  paramètres 2 entiers, 1 char
```

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ proc1 du type ptrMethode1 qui pointe vers la **meth1** de l'objet **Obj2** de classe **clB.** On suppose que l'adresse mémoire de l'objet est 14785 et l'adresse de la méthode **meth1** de l'objet est 14792 :



Il est impératif que la méthode vers laquelle pointe la variable de pointeur de méthode soit du type prévu par le type pointeur de méthode, ici la méthode meth1 doit obligatoirement être une procédure sans paramètre (compatibilité d'en-tête).

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ proc2 du type ptrMethode2 qui pointe vers la **meth2** de l'objet **Obj2** de classe **clB.** On suppose que l'adresse mémoire de l'objet est 14785 et l'adresse de la méthode **meth2** de l'objet est 14805 :



La remarque précédente sur l'obligation de compatibilité de l'en-tête de la méthode et le type pointeur de méthode implique que la méthode **meth2** doit nécessairement être une procédure à un seul paramètre real.

Affecter un pointeur de méthode

Nous venons de voir comment déclarer un type pointeur de méthode et un champ de ce même type, nous avons signalé que ce champ doit pointer vers une méthode ayant une en-tête compatible avec le type pointeur de méthode, il nous reste à connaître le mécanisme qu'utilise Delphi pour **lier un champ pointeur et une méthode à pointer**.

Types pointeurs de méthodes

```
ptrMethode1 = procedure of object;
ptrMethode2 = procedure (x : real) of object;
```

champs de pointeurs de méthodes

```
Proc1 : ptrMethode1;  // pointeur vers une méthode sans paramètre
Proc2 : ptrMethode2; // pointeur vers une méthode à un  paramètre real
```

Diverses méthodes :

procedure P1;
begin
 	
end;
	procedure P2 (x:real);
	begin
	
	end;
		procedure P3 (x:char);
		begin
		
		end;
			procedure P4;
			begin
			
			end;

Recensons d'abord les compatibilités d'en-tête qui autoriseront le pointage de la méthode :
Proc1 peut pointer vers P1 , P4 qui sont les deux seules méthodes compatibles.
Proc2 ne peut pointer que vers P2 qui est la seule méthode à un paramètre de type real.

La liaison (le pointage) s'effectue tout naturellement à travers une affectation :
L'affectation Proc1 := P1; indique que Proc1 pointe maintenant vers la méthode P1 et peut être utilisé comme un identificateur de procédure ayant la même signature que P1.

Exemple d'utilisation :

```
Proc2 := P2; // liaison du pointeur et de la procédure P2

Proc2(45.8); // appel de la procédure vers laquelle Proc2 pointe avec passage du paramètre 45.8
```

Un événement est un pointeur de méthode

Nous avons indiqué que les gestionnaires d'événements sont des méthodes, les champs du genre événements présents dans les classes Delphi sont en fait des pointeurs de méthode, qui peuvent pointer vers des gestionnaires d'événements.

Un type d'événement est donc un type pointeur de méthode, Delphi possède plusieurs types d'événements par exemple :

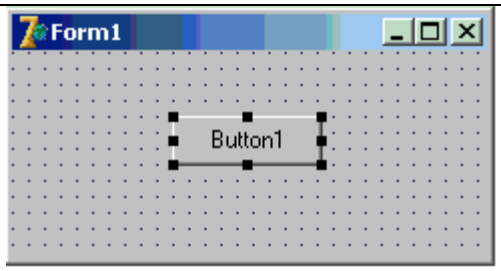
```
TNotifyEvent = procedure (Sender: TObject) of object;
TMouseMoveEvent = procedure (Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;
TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
etc …
```

Un événement est donc une propriété de type pointeur de méthode (type événement) :

```
property OnClick: TNotifyEvent;          // événement click de souris
property OnMouseMove: TMouseMoveEvent; // événement passage de la souris
property OnKeyPress: TKeyPressEvent; // événement touche de clavier pressée
```

Quel est le code engendré pour gérer un événement ?

Intéressons nous maintenant au code engendré par un programme simple constitué d'une fiche Form1 de classe TForm1 avec un objet Button1 de classe Tbutton déposé sur la fiche :




```
unit Unit1;

interface

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

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

var   Form1: TForm1;

implementation

{$R *.dfm}

end.
```

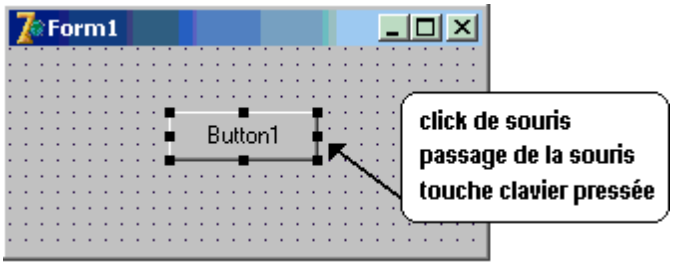
Le code source apparent fournit au programmeur. Il permet l'intervention du programmeur sur la fiche et sur ses composants.

```
object Form1: TForm1
  Left = 198
  Top = 109
  Width = 245
  Height = 130
  Caption = 'Form1'
  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 = 80
   Top = 32
   Width = 75
   Height = 25
   Caption = 'Button1'
   TabOrder = 0
 end

end
```

Le code intermédiaire caché au programmeur. Il permet l'initialisation automatique de la fiche et de ses composants.

Le code intermédiaire caché de l'objet **Button1** déposé sur la fiche Form1.

Demandons à Delphi de nous fournir (à partir de l'inspecteur d'objet) les gestionnaires des 3 événements OnClick, OnMouseMove et OnKeyPress de réaction de l'objet Button1 au click de souris, au passage de la souris et à l'appui sur une touche du clavier:




```
unit Unit1;

interface

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

type
 TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click (Sender: TObject);
  procedure Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer);
  procedure Button1KeyPress (Sender: TObject;    var Key: Char);

  private
   { Déclarations privées }
  public
   { Déclarations publiques }
  end;

var   Form1: TForm1;

implementation

{$R *.dfm}
procedure TForm1.Button1Click (Sender: TObject);
begin
                        { Gestionnaire OnClick }
end;

procedure TForm1.Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer);
begin
                        { Gestionnaire OnMouseMove }
end;

procedure TForm1.Button1KeyPress (Sender: TObject;    var Key: Char);
begin
                        { Gestionnaire OnKeyPress }
end;

end.
```

Nouveau code intermédiaire caché de l'objet **Button1** contenant les affectations des événements à leurs gestionnaires.

```
object Button1: TButton
   Left = 80
   Top = 32
   Width = 75
   Height = 25
   Caption = 'Button1'
   TabOrder = 0
   OnClick = Button1Click
   OnKeyPress = Button1KeyPress
   OnMouseMove = Button1MouseMove
  end
```

Delphi engendre un code source visible en pascal objet modifiable et un code intermédiaire (que l'on peut voir et éventuellement modifier) :

- Le code source visible est celui qui nous sert à programmer nos algorithmes, nos classes, et les réactions aux événements.

- Le code intermédiaire est généré au fur et à mesure que nous intervenons visuellement sur l'interface et contient l'initialisation de l'interface (affectations de gestionnaires d'événements, couleurs des fonds, positions des composants, taille, polices de caractères, conteneurs et contenus etc…) il sert au compilateur.

Nous venons de voir que Delphi a généré en code intermédiaire pour nous, les affectations de chaque événement à un gestionnaire :

```
OnClick = Button1Click
OnKeyPress = Button1KeyPress
OnMouseMove = Button1MouseMove
```

Et il nous a fournit les squelettes vides de chacun des trois gestionnaires :

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

procedure TForm1.Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer); …

procedure TForm1.Button1KeyPress (Sender: TObject;    var Key: Char); …
```

La dernière étape du processus de programmation de la réaction du Button1 est de programmer du code à l'intérieur des squelettes des gestionnaires.

Lors de l'exécution si nous cliquons avec la souris sur le Button1, un mécanisme d'interception et de répartition figuré ci-dessous appelle le gestionnaire de l'événement OnClick dont le corps a été programmé :

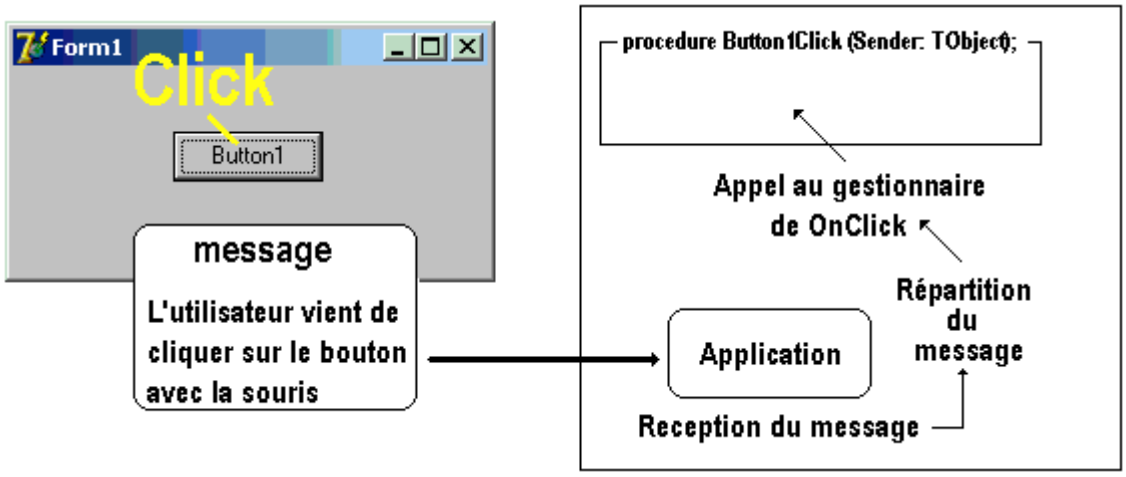


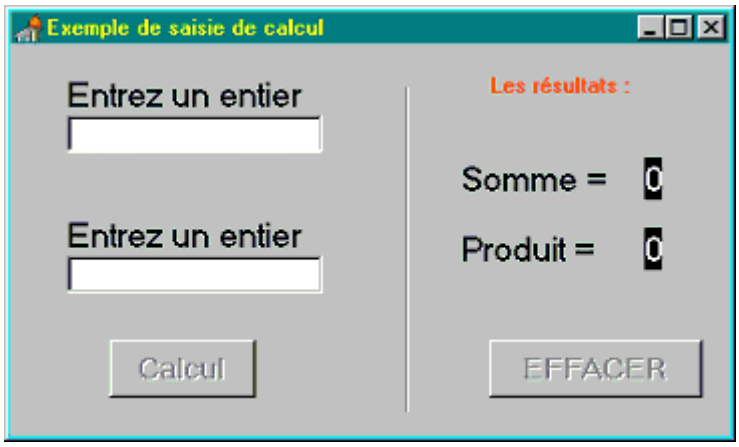
fig : *Appel du gestionnaire* ***procedure*** *TForm1.Button1Click (Sender: TObject) sur click de souris*

Exercice-récapitulatif

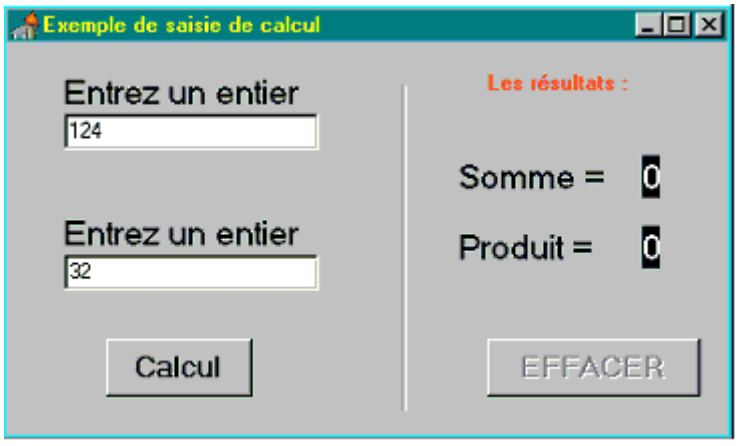
Nous voulons construire une interface permettant la saisie par l'utilisateur de deux entiers et l'autorisant à effectuer leur somme et leur produit uniquement lorsque les entiers sont entrés tous les deux. Aucune sécurité n'est apportée **pour l'instant** sur les données.

Voici l'état visuel de l'interface au lancement du programme :

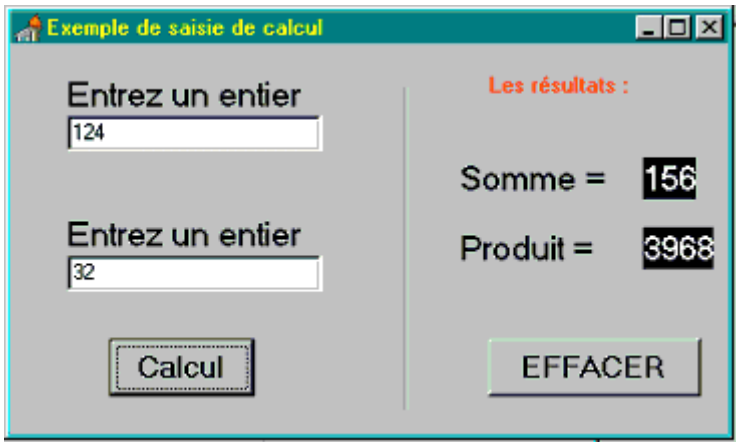
2 zones de saisie
2 boutons d'actions
2 zones d'affichages des résultats



Dès que les deux entiers sont entrés, le bouton **Calcul** est activé:



Lorsque l'on clique sur le bouton **Calcul**, le bouton **EFFACER** est activé et les résultats s'affichent dans leurs zones respectives :



Un clic sur le bouton **EFFACER** ramène l'interface à l'état initial.

Graphe événementiel complet de l'interface

Avec comme conventions sur les messages :

X_1 = exécuter les calculs sur les valeurs entrées.

X_2 = exécuter l'effacement des résultats.

M_1 = message à l'edit1 de tout effacer.

M_2 = message à l'edit2 de tout effacer.

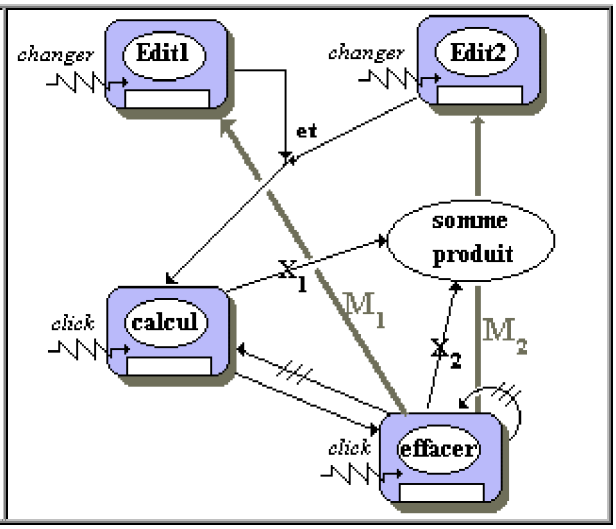


Table des actions événementielles associées au graphe

Changer Edit1	Calcul activable (si changer Edit2 a eu lieu)
Changer EDIT2	Calcul activable (si changer Edit1 a eu lieu)
Clic CALCUL	Exécuter X1 EFFACER activable Afficher les résultats
Clic EFFACER	EFFACER désactivé CALCUL désactivé Message M1 Message M2

Table des états initiaux des objets sensibles aux événements

Edit1	***activé***
Edit2	***activé***
Buttoncalcul	***désactivé***
Buttoneffacer	***désactivé***

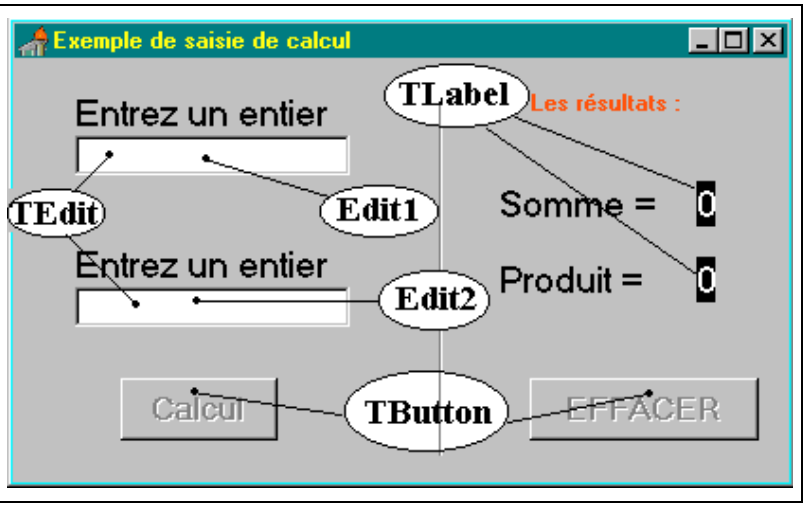
- Nous voulons disposer d'un bouton permettant de lancer le calcul lorsque c'est possible et d'un bouton permettant de tout effacer. Les deux boutons seront désactivés au départ.
- Nous choisissons 6 objets visuels : deux TEdit, **Edit1** et **Edit2** pour la saisie ; deux Tbutton, **ButtonCalcul** et **Buttoneffacer** pour les changements de plans d'action ; deux Tlabel **LabelSomme** et **LabelProduit** pour les résultats

Les objets visuels Delphi choisis

Edit1: TEdit;
Edit2: TEdit;
LabelSomme: TLabel;
LabelProduit: TLabel;

ButtonCalcul: TButton;

Buttoneffacer: TButton;



Construction progressive du gestionnaire d'événement Onchange

Nous devons réaliser une synchronisation des entrées dans Edit1 et Edit2 : le déverrouillage (activation) du bouton " ButtonCalcul " ne doit avoir lieu que lorsque Edit1 et Edit2 contiennent des valeurs. Ces deux objets de classe TEdit, sont sensibles à **l'événement OnChange** qui indique que le contenu du champ **text** a été modifié, l'action est indiquée dans le graphe événementiel sous le vocable " *changer* ".

La synchronisation se fera à l'aide de deux drapeaux binaires (des booléens) qui seront levés chacun séparément par les TEdit lors du changement de leur contenu. Le drapeau Som_ok est levé par l'Edit1, le drapeau Prod_ok est levé par l'Edit2.

Implantation : deux champs privés booléens

```
  Som_ok , Prod_ok : boolean;
```

Le drapeau Som_ok est levé par l'Edit1 lors du changement du contenu de son champ text, sur l'apparition de l'événement OnChange, il en est de même pour le drapeau Prod_ok et l'Edit2 :

Implantation n°1 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
   Prod_ok:=true; // drapeau de Edit2 levé
end;
```

Une méthode privée de test vérifiera que les deux drapeaux ont été levés et lancera l'activation du **ButtonCalcul**.

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
 if  Prod_ok and Som_ok then
    ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;
```

Nous devons maintenant tester lorsque nous levons un drapeau si l'autre n'est pas déjà levé. Cette opération s'effectue dans les gestionnaires de l'événement OnChange de chacun des Edit1 et Edit2 :

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
  TestEntrees;
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  Prod_ok:=true; // drapeau de Edit2 levé
  TestEntrees;
end;
```

Construction des gestionnaires d'événement Onclick

Nous devons gérer l'événement clic sur le bouton calcul qui doit calculer la somme et le produit, placer les résultats dans les deux Tlabels et activer le Buttoneffacer.

Implantation du gestionnaire de OnClick du ButtonCalcul :

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
   S:=strtoint(Edit1.text); // transtypage : string à integer
   P:=strtoint(Edit2.text); // transtypage : string à integer
   LabelSomme.caption:=inttostr(P+S);
   LabelProduit.caption:=inttostr(P*S);
   Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

Nous codons une méthode privée dont le rôle est de réinitialiser l'interface à son état de départ indiqué dans la table des états initiaux :

Edit1	***Activé***
Edit2	***Activé***
Buttoncalcul	***Désactivé***
Buttoneffacer	***désactivé***

```
procedure TForm1.RAZTout;
begin
   Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
   ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
   LabelSomme.caption:='0'; // RAZ valeur somme affichée
   LabelProduit.caption:='0'; // RAZ valeur produit affichée
   Edit1.clear; // message M1
   Edit2.clear; // message M2
   Prod_ok:=false; // RAZ drapeau Edit2
   Som_ok:=false; // RAZ drapeau Edit1
end;
```

Nous devons gérer l'événement click sur le Buttoneffacer qui doit remettre l'interface à son état initial (par appel à la procédure RAZTout) :

Implantation du gestionnaire de OnClick du Buttoneffacer:

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;
```

Au final, lorsque l'application se lance et que l'interface est créée nous devons positionner tous les objets à l'état initial (nous choisissons de lancer cette initialisation sur l'événement **OnCreate** de création de la fiche):

Implantation du gestionnaire de OnCreate de la fiche Form1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
end;
```

Si nous essayons notre interface nous constatons que nous avons un problème de sécurité à deux niveaux dans notre saisie :

- Le premier niveau est celui des corrections apportées par l'utilisateur (effacement de chiffres déjà entrés) et la synchronisation avec l'éventuelle vacuité d'au moins un des champs text après une telle modification : en effet l'utilisateur peut effacer tout le contenu d'un " Edit " et lancer alors le calcul, provoquant ainsi des erreurs.

- Le deuxième niveau classique est celui du transtypage incorrect, dans le cas où l'utilisateur commet des fautes de frappe et rentre des données autres que des chiffres (des lettres ou d'autres caractères du clavier).

Améliorations de sécurité du premier niveau par plan d'action

Nous protégeons les calculs dans le logiciel, par un test sur la vacuité du champ text de chaque objet de saisie TEdit. Dans le cas favorable où le champ n'est pas vide, on autorise la saisie ; dans l'autre cas on désactive systématiquement le ButtonCalcul et l'on abaisse le drapeau qui avait été levé lors de l'entrée du premier chiffre, ce qui empêchera toute erreur ultérieure. L'utilisateur comprendra de lui-même que tant qu'il n'y a pas de valeur dans les entrées, le logiciel ne fera rien et on ne passera donc pas au plan d'action suivant (calcul et affichage). Cette amélioration s'effectue dans les gestionnaires d'événement OnChange des deux TEdit.

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit1.text<' ' then // champ text non vide ok !
 begin
   Som_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; // bouton désactivé
   Som_ok:=false; // drapeau de Edit1 baissé
 end
end;
```

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit2.text<' ' then // champ text non vide ok !
 begin
   Prod_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
   Prod_ok:=false; // drapeau de Edit2 baissé
 end
end;
```

On remarque que les deux codes précédents sont très proches, ils diffèrent par leTEdit et le drapeau auxquels ils s'appliquent. Il est possible de réduire le code redondant en construisant par exemple une méthode privée avec deux paramètres.

Regroupement du code

Soit la méthode privée Autorise possédant deux paramètres, le premier est la référence d'un des deux TEdit, le second le drapeau booléen associé :

```
procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
 if Ed.text<' ' then // champ text non vide ok !
 begin
   flag :=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
    flag:=false; // drapeau de Ed baissé
 end
end;
```

Nouvelle implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 Autorise ( Edit1 , Som_ok );
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
 Autorise ( Edit2 , Prod_ok );
end;
```

Code final où tout est regroupé dans la classe TForm1

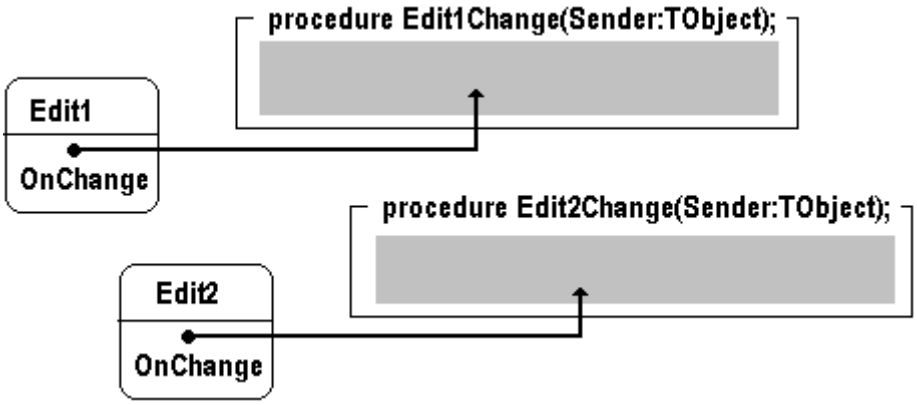
```
unit UFcalcul;

interface

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

type
 TForm1= class ( TForm )
    Edit1: TEdit;
    Edit2: TEdit;
    ButtonCalcul: TButton;
    LabelSomme: TLabel;
    LabelProduit: TLabel;
    Buttoneffacer: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit2Change(Sender: TObject);
    procedure ButtonCalculClick(Sender: TObject);
    procedure ButtoneffacerClick(Sender: TObject);
  private   { Déclarations privées }
      Som_ok , Prod_ok :
      procedure TestEntrees;
      procedure RAZTout;
      procedure Autorise ( Ed : TEdit ; var flag : boolean );
  public   { Déclarations publiques }
  end;
```

```
implementation
{ ---------- Méthodes privées ----------}
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
  if  Prod_ok and Som_ok then
     ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;

procedure TForm1.RAZTout;
begin
    Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
    ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
    LabelSomme.caption:='0'; // RAZ valeur somme affichée
    LabelProduit.caption:='0'; // RAZ valeur produit affichée
    Edit1.clear; // message M1
    Edit2.clear; // message M2
    Prod_ok:=false; // RAZ drapeau Edit2
    Som_ok:=false; // RAZ drapeau Edit1
end;

procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
  if Ed.text<' ' then // champ text non vide ok !
  begin
    flag :=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; //bouton désactivé
     flag:=false; // drapeau de Ed baissé
  end
end;

{ ------------------- Gestionnaires d'événements -------------------}
procedure TForm1.FormCreate(Sender: TObject);
begin
   RAZTout;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
   Autorise ( Edit1 , Som_ok );
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
   Autorise ( Edit2 , Prod_ok );
end;

procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
     S:=strtoint(Edit1.text); // transtypage : string à integer
     P:=strtoint(Edit2.text); // transtypage : string à integer
     LabelSomme.caption:=inttostr(P+S);
     LabelProduit.caption:=inttostr(P*S);
     Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;

end.
```

Un gestionnaire d'événement centralisé grâce au :
- polymorphisme d'objet
- pointeur de méthode

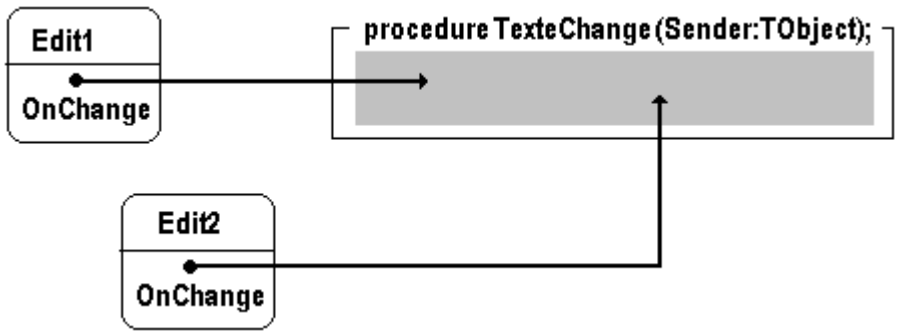
Chaque Edit possède son propre gestionnaire de l'événement OnChange :




```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 Autorise ( Edit1 , Som_ok );
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
 Autorise ( Edit2 , Prod_ok );
end;
```

On peut avoir envie de mettre en place un gestionnaire unique de l'événement OnChange, puis de le lier à chaque zone de saisie Edit1 et Edit2 (souvenons-nous qu'un champ d'événement est un pointeur de méthode) :



Nous définissons d'abord une méthode public par exemple, qui jouera le rôle de gestionnaire centralisé d'événement, nous la nommons TexteChange.

Cette méthode pour être considérée comme un gestionnaire de l'événement OnChange, doit obligatoirement être compatible avec le type de l'événement Onchange. Une consultation de la documentation Delphi nous indique que :

property OnChange: TNotifyEvent;

L'événement OnChange est du même type que l'événement OnClick (TnotifyEvent = **procedure**(Sender:Tobject) **of object**), donc notre gestionnaire centralisé TexteChange doit avoir l'en-tête suivante :

procedure TexteChange (Sender : Tobject);

Le paramètre Sender est la référence de l'objet qui appelle la méthode qui est passé automatiquement lors de l'exécution, en l'occurrence ici lorsque Edit1 appellera ce gestionnaire c'est la référence de Edit1 qui sera passée comme paramètre, de même lorsqu'il s'agira d'un appel de Edit2.

Implantation du gestionnaire centralisé

```
procedure TForm1.TexteChange(Sender: TObject);
begin
  if  Sender is TEdit then
  begin
     if (Sender as TEdit )=Edit1 then
        Autorise ( (Sender as TEdit ) , Som_ok );
     else
        Autorise ( (Sender as TEdit ) , Prod_ok );
   end
end;
```

On teste si l'émetteur Sender est bien un TEdit,

polymorphisme d'objet :

Si l'émetteur est Edit1 on le **transtype** en TEdit pour pouvoir le passer en premier paramètre à la méthode Autorise **sinon** c'est Edit2 et l'on fait la même opération.

On lie maintenant ce gestionnaire à chacun des champs OnChange de chaque TEdit dès la création de la fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
  Edit1.OnChange := TexteChange ;
  Edit2.OnChange := TexteChange ;
end;
```

pointeur de méthode

chaque champ Onchange pointe vers le même gestionnaire (méthode)

```
TForm1= class ( TForm )
   Edit1: TEdit;       …
   procedure FormCreate(Sender: TObject);
   procedure ButtonCalculClick(Sender: TObject);
   procedure ButtoneffacerClick(Sender: TObject);
 private    { Déclarations privées }
    Som_ok , Prod_ok :
    procedure TestEntrees;
    procedure RAZTout;
    procedure Autorise ( Ed : TEdit ; var flag : boolean );
 public    { Déclarations publiques }
   procedure TexteChange(Sender: TObject);
 end;
```

Le gestionnaire centralisé est déclaré ici, puis il est implémenté à la section **implementation** de la **unit**.

NOTICE METHODOLOGIQUE
construire un nouvel événement

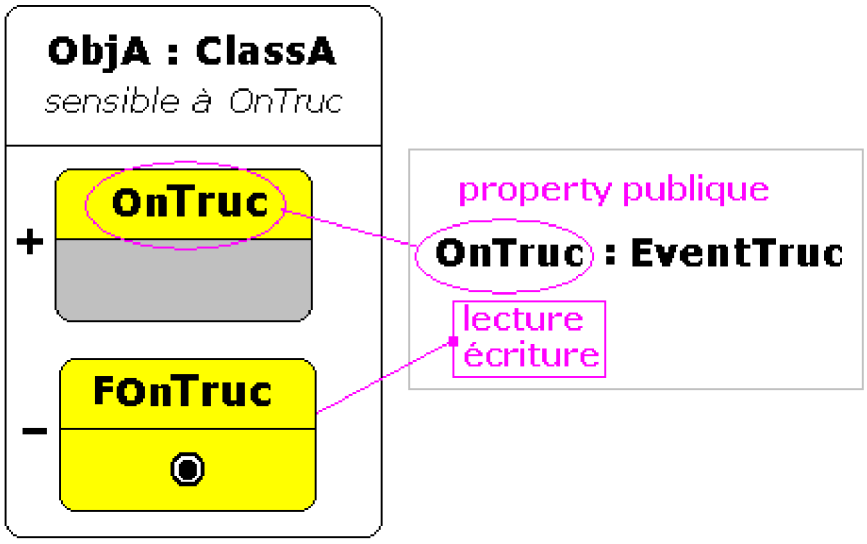
Objectif : Nous proposons ici en suivant pas à pas l'enrichissement du code de montrer comment implanter un nouvel événement nommé OnTruc dans une classe dénotée ClassA. Puis nous appliquerons cette démarche à une pile Lifo qui sera rendu sensible à l'empilement et au dépilement, par adjonction de deux événements à la classe.

Pour construire un nouvel événement dans ClassA :

- Il nous faut d'abord définir un type pour l'événement : EventTruc
- Il faut ensuite mettre dans ClassA une propriété d'événement : **property** OnTruc : EventTruc
- Il faut créer un champ privé nommé FOnTruc de type EventTruc en lecture et écriture qui servira de champ de stockage de la propriété OnTruc.

type de l'événement : Pointeur de méthode

```
EventTruc = procedure (...) of object
```

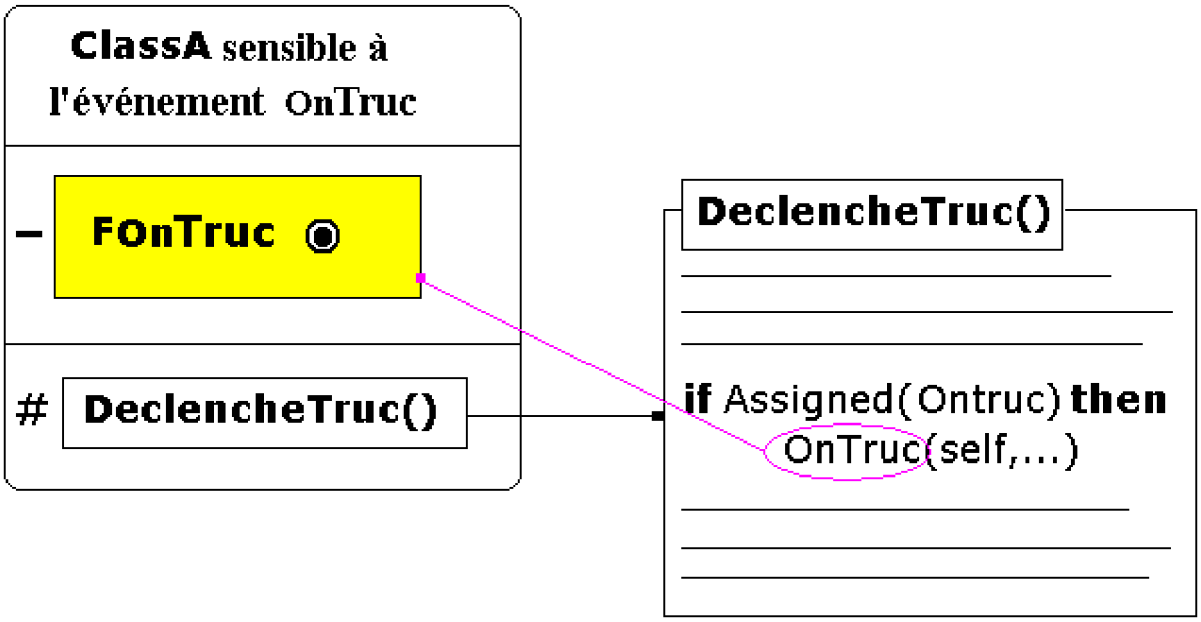


Version-1 du code source

```
Unit UDesignEvent ;

interface
 type
  EventTruc = procedure (Sender:TObject; info:string) of object ;

  ClassA = class
      private
```

```
        FOnTruc : EventTruc ;
    public
        OnTruc : EventTruc read FOnTruc  write FOnTruc ;
  end;

implementation


end.
```

- Il nous faut maintenant construire une méthode qui va déclencher l'événement, nous utilisons une procédure surchargeable dynamiquement afin de permettre des redéfinitions utlérieures par les descendants.



Lorsque l'événement se nomme OnXXX, les équipes de développement Borland donnent la fin du nom de l'événement OnXXX à la procédure redéfinissable. Ici pour l'événement **OnTruc** à la place de **DeclencheTruc**, nous la nommerons **Truc.**

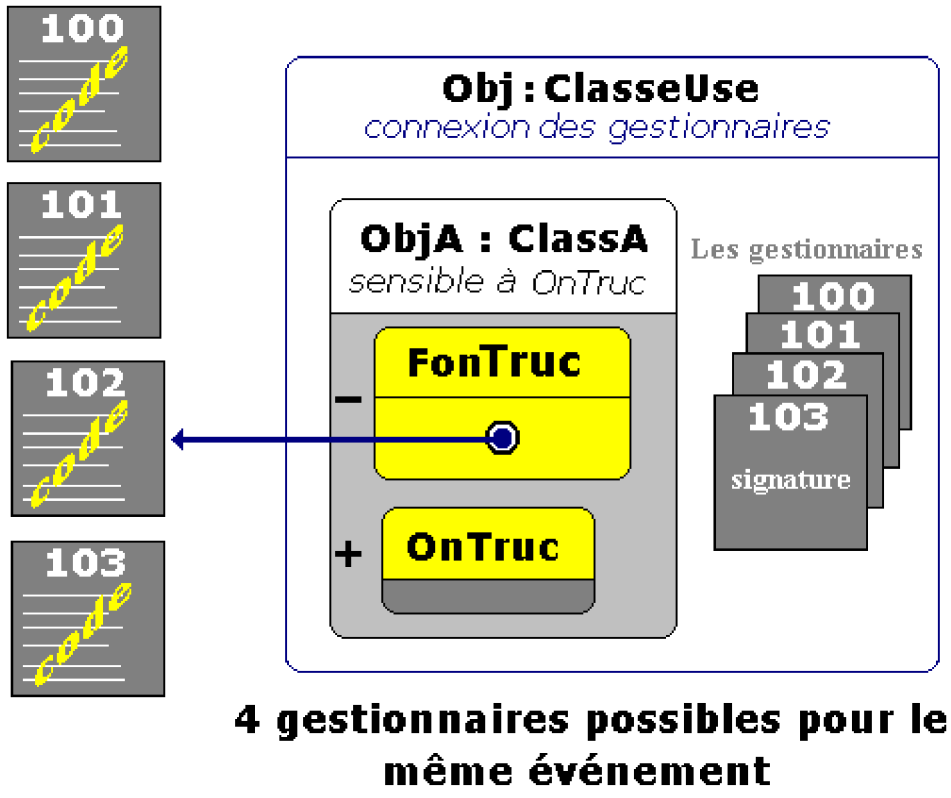
Version-2 du code source

```
Unit UDesignEvent ;

interface
 type
  EventTruc = procedure (Sender:TObject; info:string) of object ;
  ClassA = class
      private
         FOnTruc : EventTruc ;
      protected
         procedure Truc(s:string);virtual; // surchargeable dynamiquement
      public
         OnTruc : EventTruc read FOnTruc  write FOnTruc ;
   end;

implementation

 procedure ClassA.Truc(s:string);
```

```
begin
  if Assigned ( FOnTruc ) then
     FOnTruc (self , s)
end;

end.
```

- Pour terminer, il nous reste à définir un ou plusieurs gestionnaires possibles de l'événement OnTruc (ici nous en avons mis quatre), et à en connecter un à la propriété OnTruc de l'objet de classe ClassA.

Nous définissons une classe ClasseUse qui utilise sur un objet de classe ClassA l'événement OnTruc.



4 gestionnaires possibles pour le même événement

Version-3 du code source

```
Unit UDesignEvent ;

interface
 type   EventTruc = procedure (Sender:TObject; info:string) of object ;

  ClassA = class
      private
         FOnTruc : EventTruc ;
      protected
         procedure Truc(s:string);virtual; // surchargeable dynamiquement
      public
         ObjA : ClassA ;
         OnTruc : EventTruc read FOnTruc  write FOnTruc ;
         procedure LancerTruc; // Declenche l'événement OnTruc
```

```
  end;

  ClasseUse = class
      public
         procedure method_100(Sender:TObject; info:string);
         procedure method_101(Sender:TObject; info:string);
         procedure method_102(Sender:TObject; info:string);
         procedure method_103(Sender:TObject; info:string);
         procedure principale;
  end;

implementation

{------------------- ClassA  ------------------------}
 procedure ClassA.Truc(s:string);
 begin
   if Assigned ( FOnTruc ) then
      FOnTruc (self , s)
 end;

 procedure ClassA.LancerTruc ;
 begin
    ....
   Truc ("événement déclenché");
    ....
 end;
{------------------- ClasseUse  ------------------------}
 procedure ClasseUse.principale;
 begin
    //....
    ObjA := ClassA.Create ;
    ObjA.OnTruc := method_102 ; // connexion
    //....
    ObjA.LancerTruc ; // lancement
 end;

 procedure ClasseUse.method_100(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_101(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_102(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_103(Sender:TObject; info:string);
 begin
    //....
 end;

end.
```

Code pratique : une pile Lifo événementielle

Objectif : Nous livrons une classe de pile lifo héritant d'une Tlist (Un objet Tlist de Delphi, stocke un tableau de pointeurs, utilisé ici pour gérer une liste d'objets) et qui est réactive à l'empilement et au dépilement d'un objet. Nous suivons la démarche précédente en nous inspirant de son code final pour construire deux événements dans la pile lifo et lui permettre de réagir à ces deux événements.

```
unit  ULifoEvent ;

 interface
 uses  classes,Dialogs ;

 type
   DelegateLifo = procedure ( Sender: TObject ;  s :string ) of  object ;

  ClassLifo = class (TList)
  private
     FOnEmpiler : DelegateLifo ;
     FOnDepiler : DelegateLifo ;
  public
    function  Est_Vide  : boolean ;
    procedure  Empiler (elt : string ) ;
    procedure  Depiler ( var  elt : string ) ;
    property  OnEmpiler : DelegateLifo  read  FOnEmpiler  write  FOnEmpiler ;
    property  OnDepiler : DelegateLifo  read  FOnDepiler  write  FOnDepiler ;
end;

ClassUseLifo = class
 public
   procedure  EmpilerListener( Sender: TObject ;  s :string ) ;
   procedure  DepilerListener( Sender: TObject ;  s :string ) ;
   constructor  Create ;
   procedure  main ;
end;

implementation

 procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
   if  assigned(FOnDepiler)  then
     FOnDepiler ( self ,elt )
  end
 end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , PChar(elt)) ;
  if  assigned(FOnEmpiler)  then
     FOnEmpiler ( self ,elt )
 end;
```

Le type de l'événement : type pointeur de méthode (2 paramètres)

Champs privés stockant la valeur de l'événement (pointeur de méthode).

Evénement OnEmpiler :
- pointeur de méthode.

Evénement OnDepiler :
- pointeur de méthode.

self = la pile Lifo

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnDepiler **pointe vers elle, il est donc non nul.**
Sinon FOnDepiler **est nul (non assigné)**

L'instruction FOnDepiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnDepiler **pointe.**

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnEmpiler **pointe vers elle, il est donc non nul.**
Sinon FOnEmpiler **est nul (non assigné)**

L'instruction FOnEmpiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnEmpiler **pointe.**

```
function  ClassLifo.Est_Vide : boolean ;
begin
  result :=  self.Count = 0 ;
end;

 { ClassUseLifo }

constructor  ClassUseLifo.Create ;
begin
  inherited;
end;

procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a depile : ' ,s) ;
end;

procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a empile : ' ,s) ;
end;

procedure  ClassUseLifo.main ;
var
  pileLifo : ClassLifo ;
  ch :string ;
 begin
   pileLifo := ClassLifo.Create ;
   pileLifo.OnEmpiler :=  EmpilerListener ;
   pileLifo.OnDepiler :=  DepilerListener ;
   pileLifo.Empiler( '[ eau ]' ) ;
   pileLifo.Empiler( '[ terre ]' ) ;
   pileLifo.Empiler( '[ mer ]' ) ;
   pileLifo.Empiler( '[ voiture ]' ) ;
   writeln ( 'Depilement de la pile :' ) ;
   while  not  pileLifo.Est_Vide  do
   begin
     pileLifo.Depiler(ch) ;
     writeln (ch) ;
   end;
   writeln ( 'Fin du depilement.' ) ;
   readln ;
 end;

end.
```

Classe de test créant une pile Lifo

Gestionnaire de l'événement OnDepiler : signature compatible avec DelegateLifo**.**

Gestionnaire de l'événement OnEmpiler : signature compatible avec DelegateLifo**.**

Méthode main **de test**

Instanciation d'une pile Lifo à tester.

Affectation de chaque gestionnaire à l'événement qu'il est chargé de gérer.

Empilement de 4 éléments

Dépilement de toute la pile

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
   execLifo := ClassUseLifo.Create;
   execLifo.main
end.
```

exécution

```
C:\Program Files\Borland\Delphi7\Bin\Project
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
```

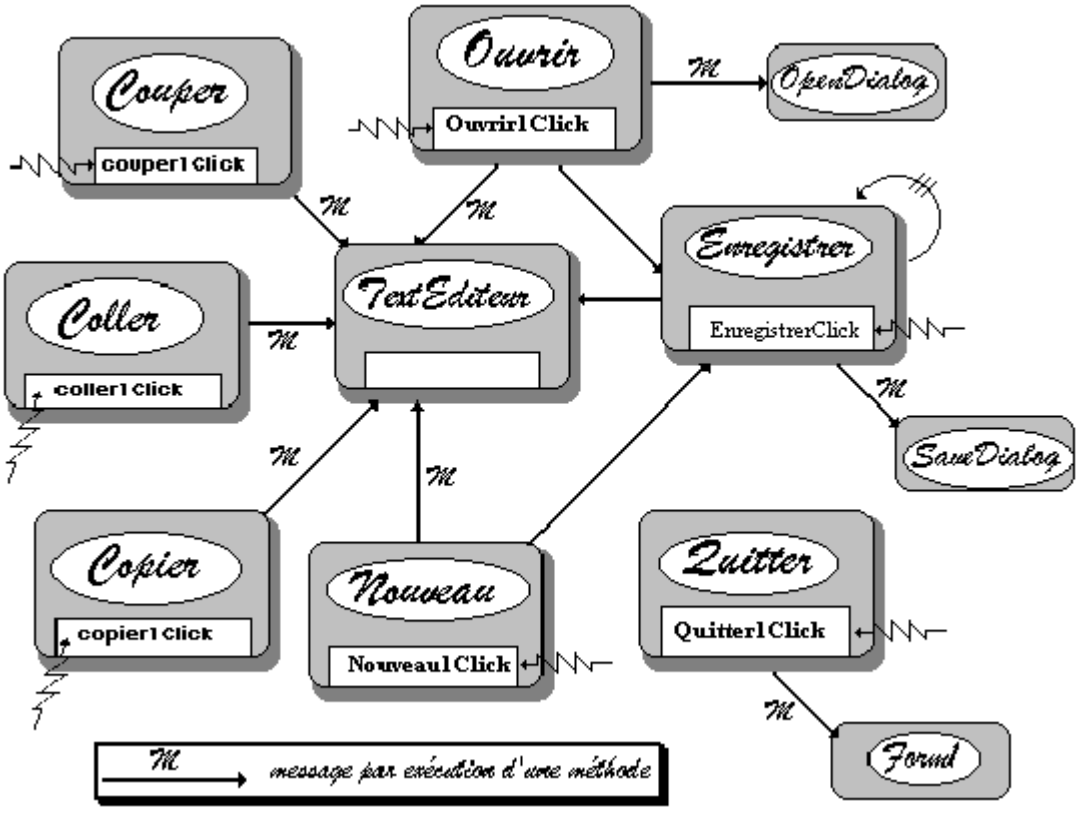
Exemple événementiel : un éditeur de texte

1. L'énoncé et les choix

Nous souhaitons développer rapidement un petit éditeur de texte qui nous permettra :

- de **lire** du texte à partir d'un fichier sur disque ou sur disquette (*ouvrir*),
- de **taper** du texte (*nouveau*),
- de **visualiser** le texte entré,
- d'effectuer sur ce texte les opérations classiques de *copier/ coller/ couper*,
- de le **sauvegarder** sur disque ou sur disquette (*enregistrer*).

Les objets potentiels réagiront à ces événements selon le graphe ci-dessous



L'objet ***TextEditeur*** est l'objet central sur lequel agissent tous les autres objets, il contiendra le texte à éditer.
En faisant ressortir les opérations à effectuer, l'utilisation de la notion d'abstraction est naturelle.

Nous envisageons les actions suivantes obtenues par réaction à un événement Click de souris (choix des objets du graphe précédent):

nouveau *(utilise un objet à définir)*
couper *(utilise un objet à définir)*

copier *(utilise un objet à définir)*
coller *(utilise un objet à définir)*
ouvrir *(utilise un objet de dialogue)*
enregistrer *(utilise un objet de dialogue)*
quitter *(utilise un objet prédéfini Form)*

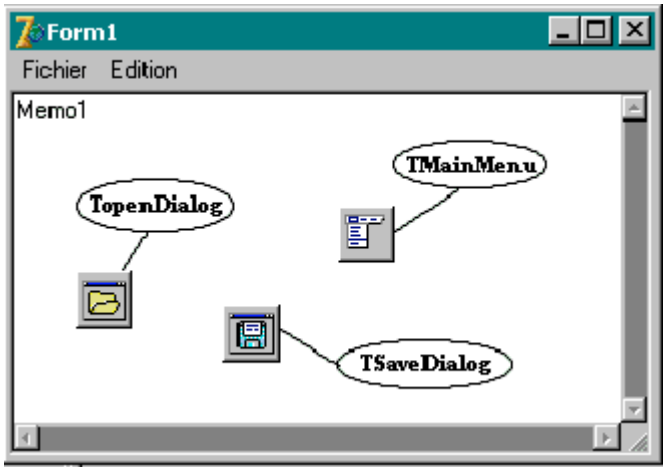
2. Les objets de composants

Delphi étant orienté objet nous allons exploiter complètement l'analyse présentée dans le graphe événementiel ci-haut en mettant en place quatre objets du genre composants visuels ou non visuels de Delphi.

L'interface choisie

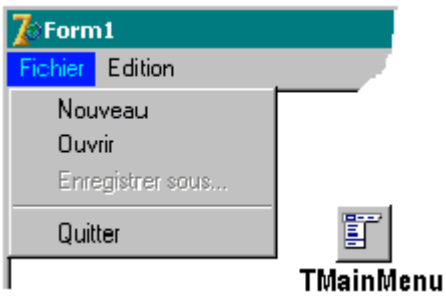
*Une fiche (**Form1**) comportant :*
- Un Tmemo avec deux ascenseurs

Trois composants non visuels :
- **TMainmenu** (une barre de 2 menus)
- **TopenDialog** (pour le chargement de fichier)
- **TSaveDialog** (pour la sauvegarde d'un texte)



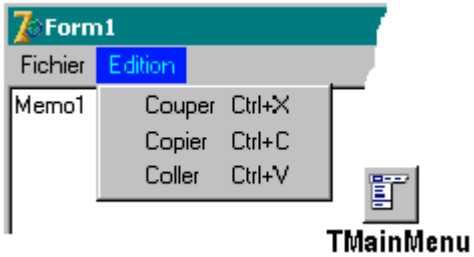
Le composant TMainmenu(1er menu)

comporte un premier menu *Fichier à 5 items*

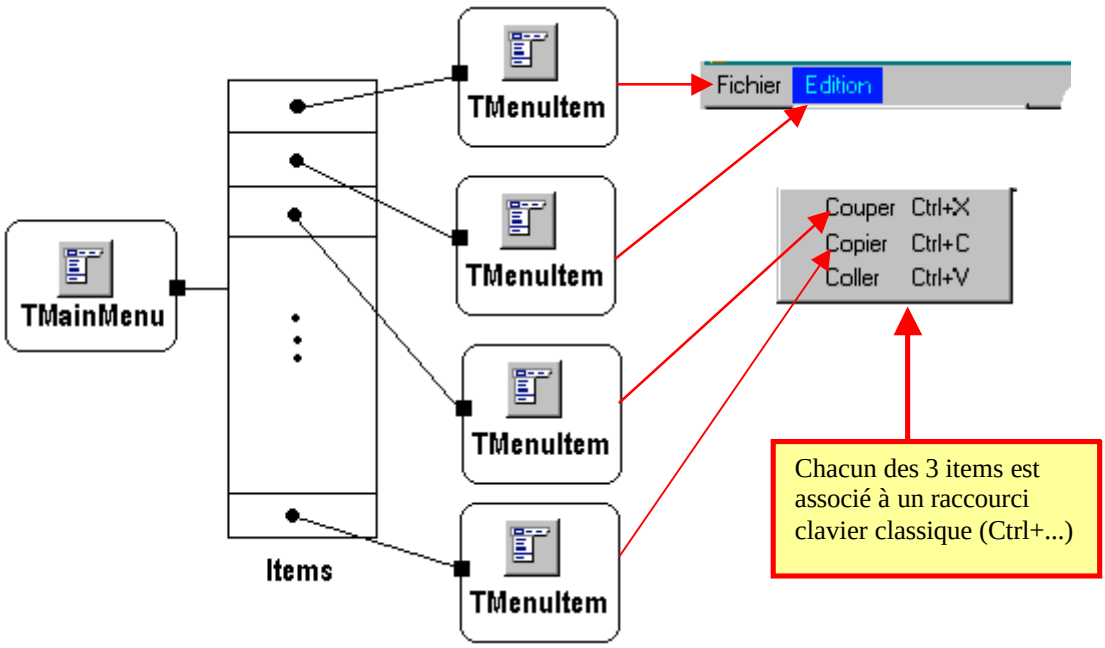


Le composant TMainmenu (2ème menu)

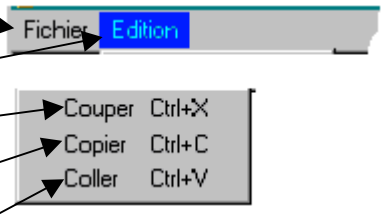
comporte un deuxième menu *Edition à 3 items.*



Les menus dans la barre et les sous-menus sont tous des objets de classe TmenuItem qui sont gérés à travers le champ Items d'un objet de classe TMainMenu :




```
object MainMenu1: TMainMenu
   ….
   object fichier1: TMenuItem
     Caption = 'fichier'
   end
   object edition1: TMenuItem
      Caption = 'edition'
      object couper1: TMenuItem
        Caption = 'couper'
      end
      object copier1: TMenuItem
        Caption = 'copier'
      end
      object coller1: TMenuItem
        Caption = 'coller'
      end
   end
end …..
```



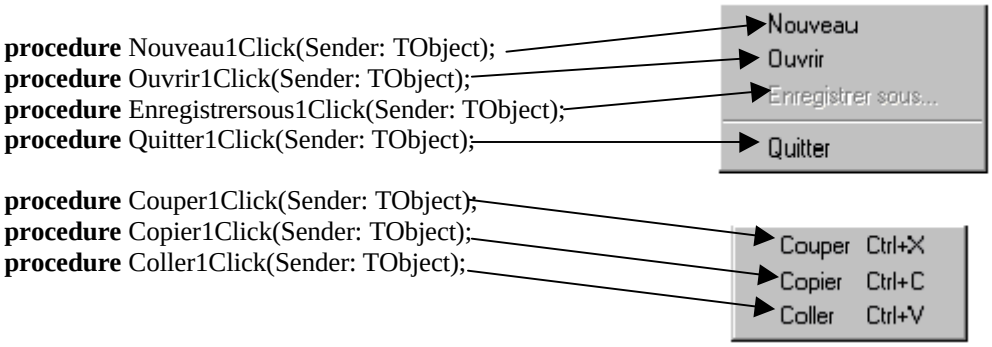
Correspondance entre le code caché par Delphi et l'inclusion des objets de TmenuItem dans un TmainMenu.

Mise en place de la gestion des événements.

A chacun des items utilisables de chacun des 2 menus, il nous faut associer un gestionnaire d'événement qui indique au programme comment il doit réagir lorsque l'utilisateur sélectionne un champ de l'un des menus par un click de souris. Nous avons vu que Delphi contient le mécanisme d'association des événements à nos gestionnaires. Beaucoup de contrôles (classes visuelles)et beaucoup d'autres classes non visuelles comme les TMenuItem, de Delphi sont sensibles à l'événement de click de souris : **property** OnClick : TnotifyEvent.

Les gestionnaires de l'événement OnClick pour les TMenuItem

Voici les en-têtes des 7 gestionnaires de OnClick automatiquement construits:



Voici pour chaque gestionnaire le code écrit par le programmeur.

Le code du gestionnaire *Quitter1Click*

```
procedure TForm1.Quitter1Click(Sender: TObject);
begin
  Close; //écrit par le développeur (fermeture de la fenêtre)
end;
```

Le code du gestionnaire *Ouvrir1Click*

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
begin
  if  OpenDialog1.Execute then //écrit par le développeur
  begin
    Enregistrersous1.enabled:=true;
    TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
  end
end;
```

Le code du gestionnaire *Enregistrersous1Click*

```
procedure TForm1.Enregistrersous1Click(Sender: TObject);
begin
  if  SaveDialog1.Execute then // écrit par le développeur
  begin
    Enregistrersous1.enabled:=false;
    TextEditeur.Lines.SaveToFile( SaveDialog1.FileName );
  end
end;
```

Le code du gestionnaire *Couper1Click*

```
procedure TForm1.Couper1Click(Sender: TObject);
begin
  TextEditeur.CutToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Copier1Click*

```
procedure TForm1.Copier1Click(Sender: TObject);
begin
  TextEditeur.CopyToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Coller1Click*

```
procedure TForm1.Coller1Click(Sender: TObject);
begin
  TextEditeur.PasteFromClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Nouveau1Click*

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  TextEditeur.Clear; &not; écrit par le développeur
  Enregistrersous1.enabled:=true &not; écrit par le développeur
end;
```

Il est à noter que nous avons écrit au total 16 lignes de programme Delphi pour construire notre micro-éditeur. Ceci est rendu possible par la réutilisabilité de méthodes déjà intégrées dans les objets de Delphi et en fait présentes dans le système Windows.

Vous pouvez voir la puissance de ce **RAD** lorsque vous observez par exemple l'instruction qui a permis de faire exécuter le "copier" ou le "chargement d'un fichier" :

```
TextEditeur.CopyToClipboard;
```

La méthode **CopyToClipboard** s'applique à l'objet **TextEditeur** qui est de la classe TMemo; cette instruction correspond à un ensemble complexe d'opérations de bas niveau :
le TMemo autorise une suite complexe d'opérations permettant de sélectionner un texte écrit sur plusieurs lignes du TMemo. Il les affiche en surbrillance et la méthode **CopyToClipboard** récupère le texte sélectionné puis le recopie enfin dans le presse-papier du système.

```
TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
```

Nous n'avons par ailleurs eu aucun code particulier à écrire pour le chargement de fichier texte, la méthode **LoadFromFile** présente dans l'objet **Lines** (de classe TStrings) effectue toutes les actions.

A titre de travail personnel il est recommandé d'enrichir ce micro-éditeur avec la possibilité de changer la police de caractère, la couleur du fond etc…

5.6 Programmation défensive
(les exceptions)

Plan du chapitre:

Introduction

Nous allons montrer comment on peut concevoir la programmation défensive en protégeant directement le code à l'aide de la notion d'exception (semblable à celle du C++ ou d'Ada). L'objectif principal est d'améliorer la qualité de " *robustesse* " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et néanmoins efficace de logiciels robustes.

Rappelons au lecteur que la sécurité d'une application est susceptible d'être mise à mal par toute une série de facteurs :

- les problèmes liés au matériel : par exemple la perte subite d'une connection à un port, un disque défectueux...
- les actions imprévues de l'utilisateur, entrainant par exemple une division par zéro...

1. Notions de défense et de protection

A l'occasion de la traduction Algorithme à langage évolué comme pascal, nous avons répertorié en plus des actions algorithmiques, **des actions de sécurité** et des actions ergonomiques qui doivent elles aussi être programmées.

La partie ergonomie est incluse dans la notice consacrée aux interfaces de communication logiciel-utilisateur.
La partie sécurité a déjà été abordée ailleurs, nous regroupons ici ce que nous devons connaître.

Pour obtenir une certaine " **robustesse** " dans nos programmes nous savons déjà que la sécurité doit porter au moins sur :

- les domaines de définitions des données,
- les contraintes d'implantation,
- le filtrage des saisies,
- les problèmes de transtypage

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue.

La ***programmation défensive*** est plus une attitude de pensée et de comportement qu'une nouvelle méthode. Cette attitude consiste à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

La comparaison des coûts de différents ratios de productivité établie par B.Boehm, montre que l'exigence de fiabilité est l'une des plus chères (surcoût de 87%). Cette remarque pourrait en apparence nous conduire à croire que ce facteur est donc une affaire de spécialistes et ne constitue pas une préoccupation dans un discours d'initiation. Nous allons voir qu'il n'en est rien et qu'en fait notre méthode de travail et les outils que nous utilisons concourent à une attitude de programmation défensive sans que nous contraignions fortement notre pensée en ce sens.

Nous pensons enfin qu'il est bon d'induire dans l'esprit du développeur en programmation visuelle débutant des idées fondées sur des pratiques méthodiques qui lui éviteront l'écueil de " l'art indiscipliné " du logiciel, mais qui pourraient lui donner le goût de continuer dans cette discipline.

1.1 Outils participant à la programmation défensive

Voici cinq secteurs d'activité parmi ceux que nous connaissons, participant à une méthode de programmation défensive.

La modularité

Le principe du découpage en modules restreint la propagation des effets perturbateurs sur d'autres modules à partir d'une erreur survenant dans un module donné.

L'encapsulation

Associée à la modularité, elle protège les constituants internes d'un module (champs et méthodes).

Les TAD (types abstraits)

En fournissant un outil de spécification des données avec des préconditions sur la validité des opérateurs, un TAD assure la description des vérifications de domaines.

L'utilisation de méthodes systématiques

Comme l'algorithmique, la programmation par la syntaxe, les machines abstraites, les générateurs d'automates,... ces outils encouragent l'étudiant à s'essayer à une programmation sûre.

L'utilisation d'un RAD visuel comme Delphi

Ce genre de système de développement apporte un certains nombres d'avantages en la matière :

- il autorise la mise en œuvre des cinq secteurs précédents d'activité,
- il fournit au programmeur des kits d'objets sécurisés et réutilisables,
- il permet de construire rapidement des interfaces qui participent à une meilleure défense du logiciel contre des actions non valides (par exemple en utilisant la méthode de séquencement par plans d'action).

A côté de cet éventail d'outils intégrant en général la notion de programmation défensive, il existe un outil spécifique dédié à ce genre de programmation : ***les exceptions***.

1.2 Rôle et mode d'action d'une exception

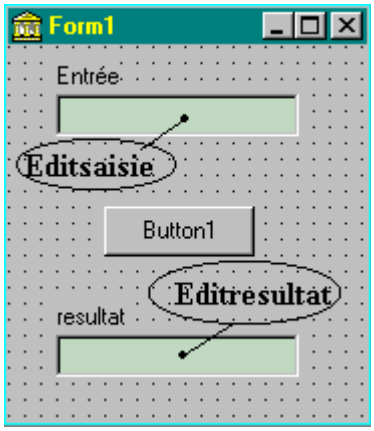
Rôle d'une exception

Réservé jusqu'à présent aux spécialistes en Ada, en C++ ou en Java, cet outil est mis dans le RAD Delphi à la portée du débutant. Comme son nom l'indique, une exception est chargée de signaler un comportement *exceptionnel* (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même. C'est le cas de Delphi qui intègre les exceptions comme une classe particulière : la classe **Exception**. Cette classe contient un nombre important de classes dérivées.

Comment agit une exception

Dès qu'une erreur se produit comme un manque de mémoire , un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " *déclenche une exception* ".

Exemple : soit une saisie d'un entier dans un Tedit nommé Editsaisie.



Objets visuels :

Editsaisie: TEdit;
Editresultat: TEdit;
Button1: TButton;

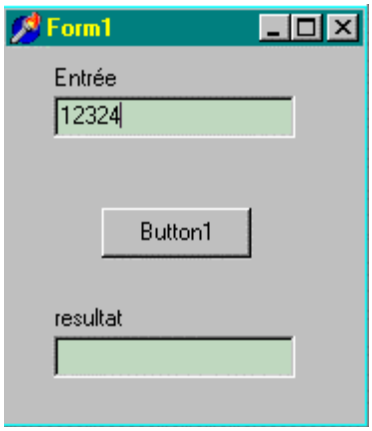
Actions :

L'utilisateur entre un entier dans Editsaisie, il clique sur le bouton Button1 qui effectue un transtypage dans une variable locale " n : integer " puis il se désactive et le Tedit Editresultat, change de couleur.

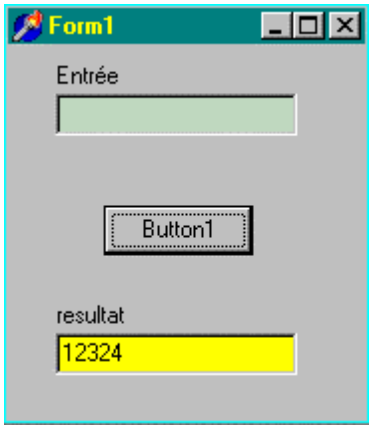
Le gestionnaire d'événement clic de Button1 est alors le suivant :

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= Editsaisie.text;
  Editsaisie.clear ;
end;
```

Une exécution sans incident donne ceci :



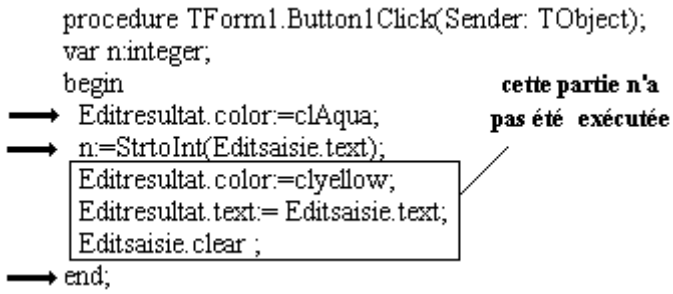
avant clic sur Button1



après clic sur Button1

Lors de l'exécution, supposons que nous entrons dans Editsaisie la chaîne " asa12324 " qui n'est pas un entier.
La fonction **StrtoInt** déclenche une exception et nous envoie un message d'erreur général sur l'incident qui vient de se produire : (après clic sur Button1 Message d'erreur)

En outre, l'incident a arrêté l'exécution du code dans le gestionnaire Button1Clic. Le code a été exécuté normalement jusqu'à l'instruction de transtypage qui a déclenché l'exception. Le reste des lignes de code a été ignoré :



Si nous voulons que le message soit plus explicite, ou si nous voulons par exemple que malgré tout une certaine partie du code s'exécute en fournissant des valeurs par défaut malgré l'incident, nous devons gérer nous-mêmes cette exception.
Afin de pouvoir gérer une exception déclenchée, il nous faut disposer d'un gestionnaire d'exception qui permette de traiter tous les types d'exception.

1.3 Gestion de la protection du code

Le langage Delphi contient un mécanisme appelé gestionnaire d'exception qui s'insère dans les lignes de code là où nous souhaitons assurer une protection.

Syntaxe du gestionnaire : mots clefs (try, except)

try
- ...
<lignes de code à protéger>
- ...

except
- ...
<lignes de code réagissant à l'exception>
- ...

end ;

Principe de fonctionnement d'un tel gestionnaire :

Dès qu'une exception est déclenchée dans le bloc de lignes compris entre **try**....**except**, il y a déroutement de l'exécution (arrêt d'exécution séquentielle du code) vers la première ligne du bloc **except**...**end** et l'exécution continue séquentiellement à partir de cet endroit.
Reprenons l'exemple précédent légèrement modifié dans le code du gestionnaire du clic de Button1.

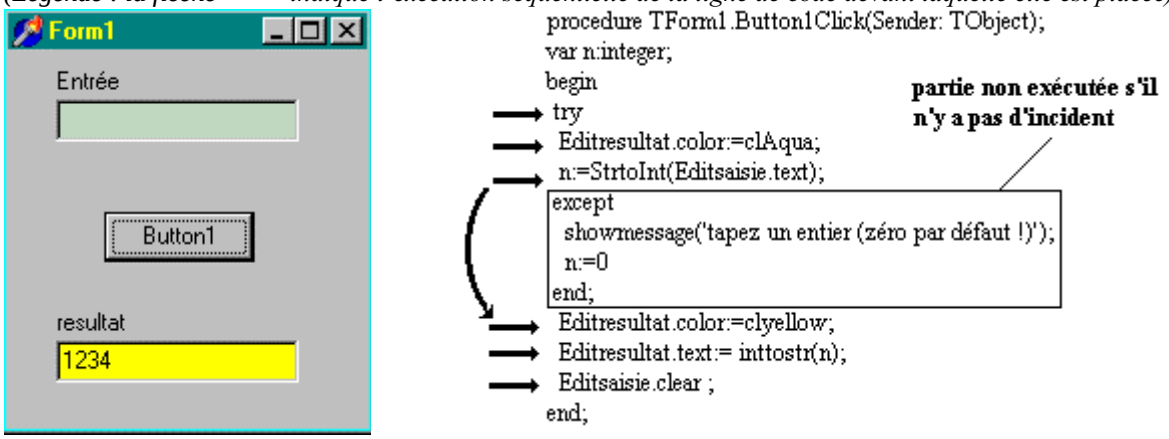
```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= inttostr(n);
  Editsaisie.clear ;
end;
```

Mettons en place une protection de l'instruction incriminée, tout en conservant l'exécution des lignes de code suivantes. Comme la variable " n " n'aura pas de valeur à cause de l'incident, nous lui attribuons la valeur zéro par défaut si un incident se produit.

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
 try
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
 except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
 end;
 Editresultat.color:=clyellow;
 Editresultat.text:= inttostr(n);
 Editsaisie.clear ;
end;
```

1.3.1 - Fonctionnement sans incident :

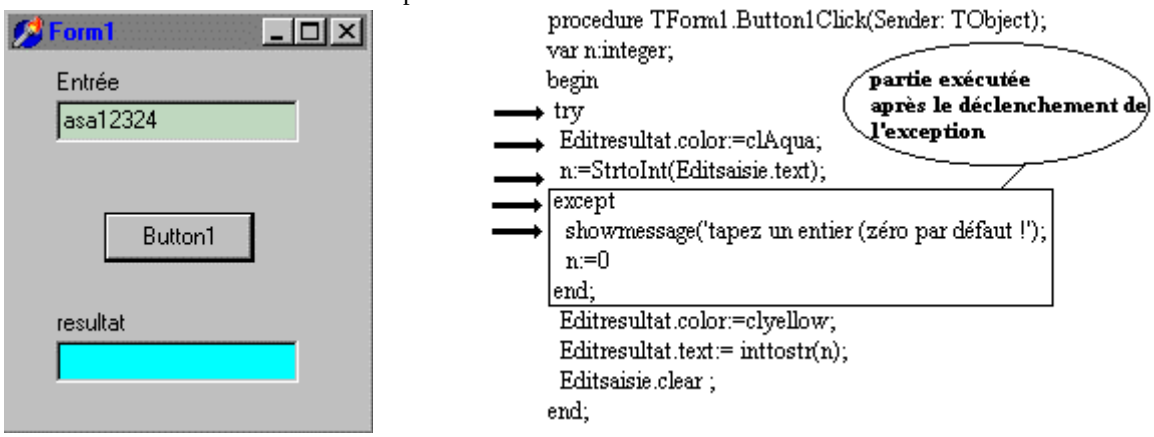
(Légende : la flèche ➡ indique l'exécution séquentielle de la ligne de code devant laquelle elle est placée).



Les lignes de code sont exécutées séquentiellement sauf le bloc **except**...**end** (qui n'est exécuté qu'en cas d'incident).

1.3.2 - Fonctionnement avec incident :

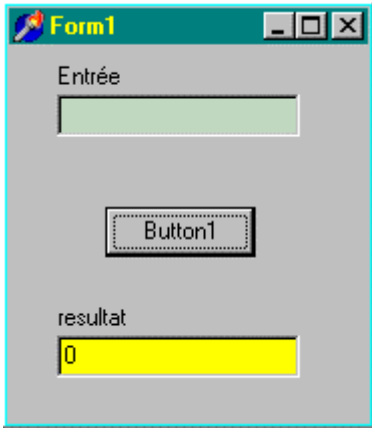
Nous entrons dans Editsaisie comme précédemment une valeur non entière.



Message envoyé par la fonction Showmessage dans le bloc **except**...**end**.

Ensuite le code continue à s'exécuter en séquence à partir de n :=0 ;...

*Suite du déroulement séquentiel de l'exécution dans le bloc **except…end** :*




```
procedure TForm1.Button1Click(Sender: TObject);
var n:integer;
begin
 try
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
end;
  Editresultat.color:=clyellow;
  Editresultat.text:= inttostr(n);
  Editsaisie.clear ;
end;
```

partie exécutée après le déclenchement de l'exception

*En fait dans cet exemple, n'importe quelle exception déclenchée dans le bloc **try…except** déroute vers le bloc **except…end**.*

1.4 Effets dus à la position du bloc except...end

Afin de bien comprendre cette notion de déroutement (dont le placement est à la charge du programmeur) observons ce qu'apporte une modification de la place du bloc **except...end** au déroulement du code pendant l'exécution.
Soit, dans le même exemple, le nouveau code dans Button1Clmick où nous avons repoussé le bloc **except**...**end** à la fin.

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
 try
   Editresultat.color:=clAqua;
   n:=StrtoInt(Editsaisie.text);
   Editresultat.color:=clyellow;
   Editresultat.text:= inttostr(n);
   Editsaisie.clear ;
 except
   showmessage('tapez un entier (zéro par défaut !');
   n:=0
 end;
end;
```

Puis exécutons le programme.

1.4.1 - Fonctionnement sans incident :

Identique au précédent paragraphe, le bloc **except...end** étant ignoré.

1.4.2 - Fonctionnement avec incident :

(Légende : la flèche ➡ indique l'exécution séquentielle de la ligne de code devant laquelle elle est placée).


```
procedure TForm1.Button1Click(Sender: TObject);
var n:integer;
begin
try
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= inttostr(n);
  Editsaisie.clear ;
except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
end;
end;
```

partie non exécutée suite à la levée de l'exception

le code continue son déroulement dans le bloc **except...end** en " sautant " trois instructions.


```
except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
end;
end;
```

Etat final des résultats après traitement de l'exception :

les deux Tedit Editresultat et Editsaisie n'ont pas changé puisque les instructions:

```
Editresultat.color:=clyellow;
Editresultat.text:= inttostr(n);
Editsaisie.clear ;
```

n'ont pas été exécutées, par suite du déroutement du code.

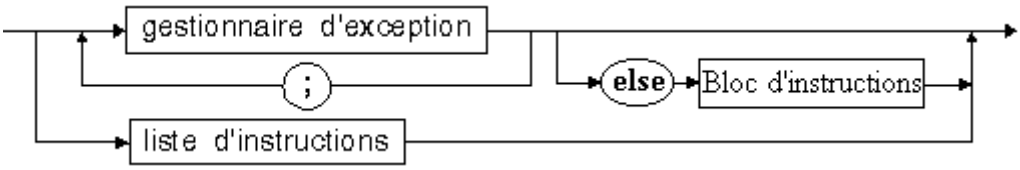
Nous notons que cette méthode de déroutement du code est très proche du fonctionnement d'une machine de Von Neumann. Nous venons de voir comment intercepter une exception quelconque sans savoir exactement sa catégorie. Nous pouvons en fait intercepter une exception d'une manière encore plus " fine " avec le gestionnaire **try...except...end**.
Il nous permet de sélectionner la classe exacte de l'exception et de ne faire fonctionner le déroutement du code que pour une exception définie à l'avance.

1.5 Interception d'une exception d'une classe donnée

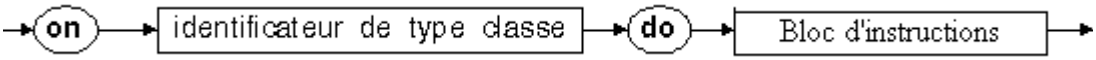
Diagrammes de syntaxe du gestionnaire :

<instruction try> :

<Bloc d'exception> :

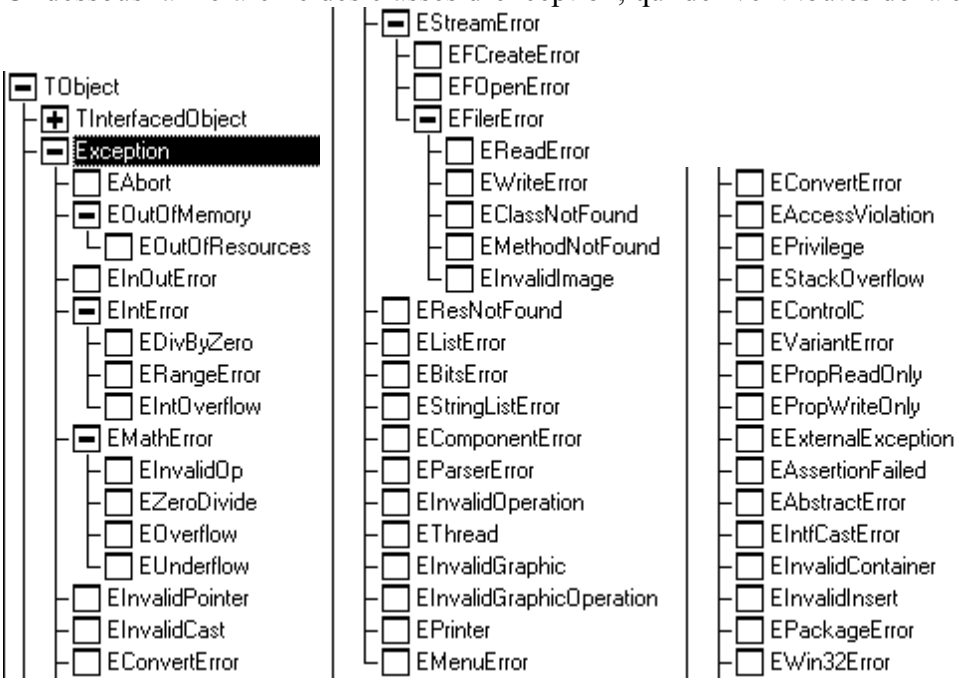


<gestionnaire d'exception> :



Différentes classes d'exception Delphi

Ci-dessous la hiérarchie des classes d'exception, qui dérivent toutes de la classe **Exception**:



Exemple d'écriture d'un gestionnaire :

```
try
        <lignes de code à protéger>
    except
     On EConvertError do begin
        <lignes de code réagissant à l'exception EConvertError >
```

```
        end ;
        On EintError do begin
            <lignes de code réagissant à l'exception EintError >
        end ;
        else begin
            <lignes de code réagissant à d'autres exceptions >
        end ;
       end ;
```

Principe de fonctionnement d'un tel gestionnaire

Dès qu'une exception est déclenchée dans le bloc de lignes compris entre **try**....**except**, il y a déroutement de l'exécution (arrêt d'exécution séquentielle du code) vers le bloc **except**...**end**.
Le sélecteur de gestionnaire d'exception **on...do** fonctionne approximativement comme un **case...of**, en n'exécutant que le champ sélectionné.
Supposons que l'exception levée soit de la classe EintError ; l'instruction try n'exécute alors que le code du " bon " gestionnaire en l'occurrence :

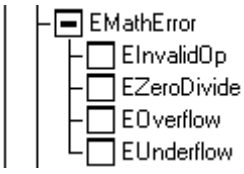
```
  On EintError do begin
    <lignes de code réagissant à l'exception EintError >
  end ;
```

Puis l'exécution se poursuit après le **end** du bloc **except...end**.

1.6 Ordre dans l'interception d'une exception

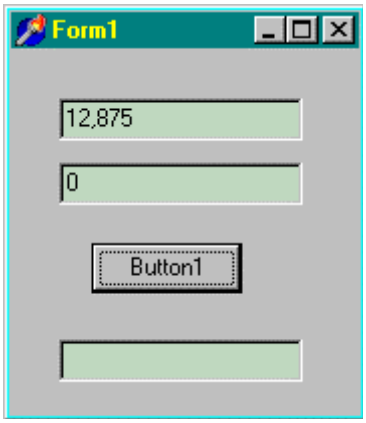
A la différence d'un "**case** … **of**" pascal, le choix du sélecteur de gestionnaire (**on**...**do**) s'effectue séquentiellement dans l'ordre d'écriture des lignes de code. On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

Exemple : division par zéro dans un calcul en virgule flottante
EMathError est la classe des exceptions pour les erreurs de calcul.



EZeroDivide indique la division par zéro dans une telle opération.

Programmons un calcul à partir d'un bouton :



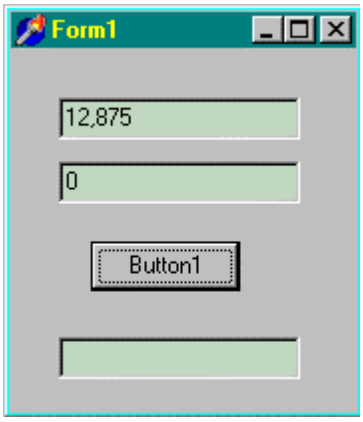

```
procedure TForm1.Button1Clic(Sender:
TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
  .............
  .............
 end;
end;
```

Au départ nous avons edit1.text = 12,875 et edit2.text = 0. Le calcul est erroné car x vaut 12,85 et y vaut zéro ce qui va induire une erreur de division par zéro dans l'instruction z := x / y .

Observons ce qui se passe lorsque nous interceptons les exceptions **EMathError** et **EZeroDivide**.

1.6.1 - Interception dans l'ordre de la hiérarchie :

La sélection d'exception est programmée dans l'ordre de la hiérarchie des classes :
EMathError
  |____ **EZeroDivide**

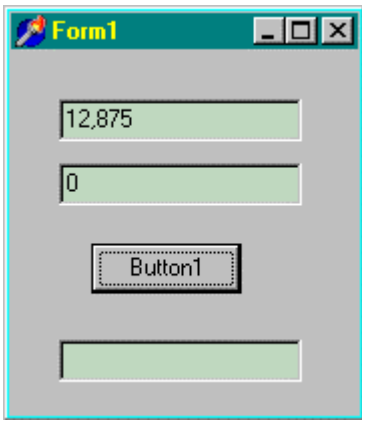



```
procedure TForm1.Button1Clic(Sender: TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
   on EMathError do
     Edit3.text:='Erreur générale';
   on EZeroDivide do
     Edit3.text:='division par zéro';
 end;
end;
```

EMathError est interceptée en premier.

1.6.2 - Interception dans l'ordre inverse :

La sélection d'exception est programmée dans l'ordre inverse de la hiérarchie des classes.



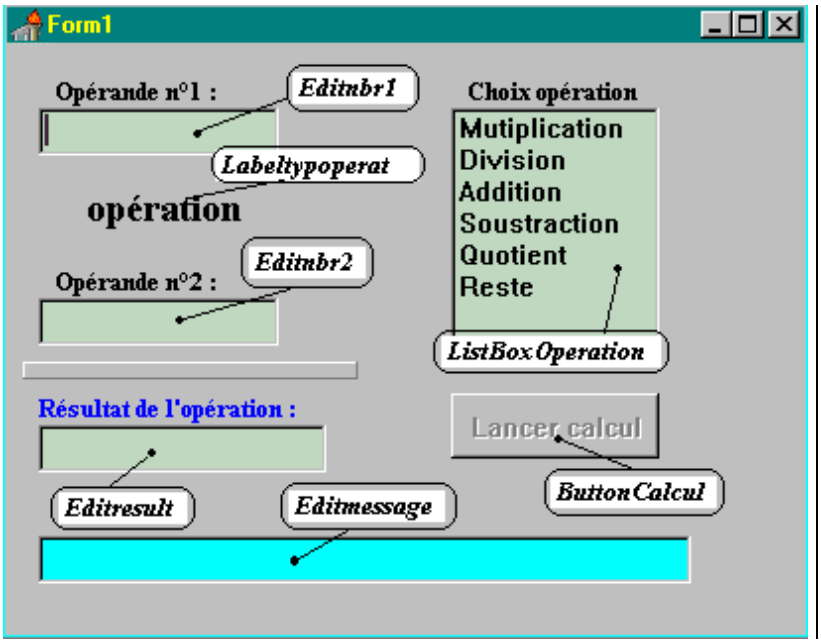

```
procedure TForm1.Button1Clic(Sender: TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
  on EZeroDivide do
     Edit3.text:='division par zéro';
   on EMathError do
     Edit3.text:='Erreur générale';
 end;
end;
```

C'est **EZeroDivide** qui est interceptée en premier.

2. Traitement d'un exemple de protections

Reprenons comme base le premier exemple étudié dans le chapitre sur les interfaces.
Supposons que nous voulons élargir notre interface de calcul aux opérations entières :
Mutiplication, Division, Addition, Soustraction, Quotient, Reste.
Nous limiterons nos entiers au type **Smallint** Delphi identique au type integer du pascal (Smallint = -32768..32767, signé sur 16 bits).

Interface retenue



Editnbr1: TEdit;

Editnbr2: TEdit;

Editmessage: TEdit;

Editresult: TEdit;

ListBoxOperation: TListBox;

ButtonCalcul: TBitBtn;

Labeltypoperat: TLabel;

2.1 Le code de départ de l'unité

```
Champ privé de la classe Tform1 :
  Toper:array[0..5]of string;
```

implementation

Le tableau Toper contient les symboles des opérateurs entiers, il est initialisé lors de l'activation de la fiche

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Toper[0]:='*'; Toper[1]:='/';
  Toper[2]:='+'; Toper[3]:='-';
  Toper[4]:=' div '; Toper[5]:=' mod ';
end;
```

L'utilisateur choisit par sélection dans le ListBoxOperation l'opération qu'il veut effectuer ; l'étiquette Labeltypoperat affiche le symbole associé :

```
procedure TForm1.ListBoxOperationClic(Sender: TObject);
begin
  Labeltypoperat.caption:=Toper[ListBoxOperation.ItemIndex];
end;
```

Les deux Edit de saisie Editnbr1 et Editnbr2 sont sensibles à l'événement OnChange qui permet de déverrouiller le bouton de calcul :

```
procedure TForm1.Editnbr1Change ( Sender :
TObject);
begin
  ButtonCalcul.enabled:=true;
end;
```

```
procedure TForm1.Editnbr2Change  ( Sender :
TObject);
begin
  ButtonCalcul.enabled:=true;
end;
```

Une fois que les entiers sont entrés et le genre d'opération sélectionné, le ButtonCalcul lance le calcul sur un clic de l'utilisateur :

```
procedure TForm1.ButtonCalculClic(Sender: TObject);
var op1,op2,result:smallint;
begin
  ButtonCalcul.enabled:=false;
  op1:=strtoint(Editnbr1.text);
  op2:=strtoint(Editnbr2.text);
  case ListBoxOperation.ItemIndex of
    0: result:=op1 * op2;
    1: result:=trunc(op1 / op2);
    2: result:=op1 + op2;
    3: result:=op1 - op2;
    4: result:=op1 div op2;
    5: result:=op1 mod op2;
  end;
  Editresult.text:=inttostr(result);
```

```
  Editmessage.text:=inttostr(op1)+Toper[ListBoxOperation.ItemIndex]
      +' '+inttostr(op2)+'= '+Editresult.text;
end;
```

A ce stade la saisie n'est pas encore sécurisée. Afin que le lecteur puisse retrouver les éléments déjà traités dans le chapitre sur les interfaces, nous reprenons comme premier niveau de sécurité le même genre de programmation : synchronisation des 3 objets de saisie Editnbr1, Editnbr2 et ListBoxOperation, puis programmation par plans d'action.

2.2 Code de la version.1 (premier niveau de sécurité)

Champs privés de la classe Tform1**:**
oper1,oper2,operat:boolean; *//les 3 drapeaux*
Toper:**array**[0..5]**of** string;

implementation

- Le tableau Toper contient les symboles des opérateurs entiers, il est initialisé lors de l'activation de la fiche (pas d'ajout de code ni de changement).
- Ajout de la méthode **RAZtout** positionnant l'interface à son état initial (correspondant à la table des états initiaux) :

```
procedure TForm1.RAZTout;
begin
  ButtonCalcul.enabled:=false;
  Editnbr1.clear;
  Editnbr2.clear;
  Editresult.clear;
  Editmessage.clear;
  oper1:=false;
  oper2:=false;
  operat:=false;
end;
```

Adjonction de la méthode TestEntrees chargée de lancer l'activation deButtonCalcul si les trois drapeaux sont tous levés :

```
procedure TForm1.TestEntrees;
begin
  if oper1 and oper2 and operat then
     ButtonCalcul.enabled:=true
end;
```

L'utilisateur choisit par sélection dans le ListBoxOperation l'opération qu'il veut effectuer, l'étiquette Labeltypoperat affiche le symbole associé. Adjonction dans le code du drapeau et du test :

```
procedure TForm1.ListBoxOperationClic(Sender: TObject);
begin
  operat:=true;
  TestEntrees;
  Labeltypoperat.caption:=Toper[ListBoxOperation.ItemIndex];
end;
```

Les deux Edit de saisie Editnbr1 et Editnbr2 sont sensibles à l'événement OnChange ; voici l'ajout du code de plans d'action dans les gestionnaires de cet événement :

```
procedure TForm1.Editnbr1Change(Sender: TObject);
begin
 if Editnbr1.text<>'' then begin
   oper1:=true;
   TestEntrees;
 end
 else begin
   ButtonCalcul.enabled:=false;
   oper1:=false; // drapeau de Editnbr1 baissé
 end
end;
```

```
procedure TForm1.Editnbr12Change(Sender: TObject);
begin
 if Editnbr2.text<>'' then begin
    oper2:=true;
    TestEntrees;
 end
 else begin
    ButtonCalcul.enabled:=false;
    oper2:=false; // drapeau de Editnbr2 baissé
 end
end;
```

Le code du plan d'action associé au ButtonCalcul reste strictement le même.

Nous proposons dans le paragraphe qui suit, un complément de sécurité apporté par des interceptions d'exception.

2.3 Code de la version.2 (deuxième niveau de sécurité)

Tout le code de la version.1 reste identique dans la version.2, les adjonctions sont uniquement dans le gestionnaire du ButtonCalcul. Nous lançons les levées d'exception lorsque les incidents ont lieu. Nous avons ajouté une méthode signe qui renvoie +1 ou -1 selon le signe de l'entier d'entrée et d'une constante d'entier maximum:

```
const Maxint=32767;
function signe(n:smallint):smallint;
begin
  if n>0 then signe:=1
  else if n<0 then signe:=-1
         else signe:=0
end;
```

Le code est protégé par les exceptions suivantes :

```
procedure TForm1.ButtonCalculClic(Sender: TObject);
var op1,op2,result:smallint;
begin
 ButtonCalcul.enabled:=false;
try
  op1:=strtoint(Editnbr1.text);
 except
  on ERangeError do
   if Editnbr1.text[1]='-' then
      op1:=-Maxint
   else op1:=Maxint;
  on EConvertError do op1:=Maxint;
end;
try
 op2:=strtoint(Editnbr2.text);
except
 on EConvertError do op2:=Maxint;
 on ERangeError do
  if Editnbr2.text[1]='-' then
    op2:=-Maxint
  else op2:=Maxint;
end;

try
 case ListBoxOperation.ItemIndex of
  0: result:=op1 * op2;
  1: result:=trunc(op1 / op2);
  2: result:=op1 + op2;
  3: result:=op1 - op2;
  4: result:=op1 div op2;
  5: result:=op1 mod op2;
 end;
except
 on EDivByZero do
  if ListBoxOperation.ItemIndex=4 then
   result:=signe(op1)*Maxint
  else result:=op1;
 on EZeroDivide do
  result:=signe(op1)*Maxint;
 on EIntOverFlow do
  result:=signe(op1)*signe(op2)*Maxint;
end;

Editresult.text:=inttostr(result);
Editmessage.text:=inttostr(op1)+Toper[ListBoxOperation.ItemIndex]
      +' '+inttostr(op2)+'= '+Editresult.text;
end;
```

Le lecteur modifiera les choix de valeurs par défaut dans les gestionnaires d'exception. Dans l'exemple plus haut, ces choix ne sont qu'indicatifs et servent à montrer que l'on peut, soit arrêter un calcul, soit le continuer avec une valeur de remplacement après signalement de l'erreur. Il s'assurera par lui-même que la conjonction entre les plans d'action et les exceptions est un système de programmation défensive efficace.

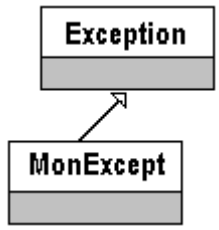
Créer et lancer ses propres exceptions

- Il est possible de construire de nouvelle classe d'exceptions personnalisées en héritant d'une des classes de Delphi mentionnées plus haut, ou à minima de la classe de base **Exception**.(cette classe contient une propriété public **property** Message: **string**, qui contient en type string le texte à afficher dans la boîte de dialogue des exceptions quand l'exception est déclenchée).

- Il est aussi possible de lancer une exception personnalisée (comme si c'était une exception propre à Delphi) à n'importe quel endroit dans le code d'une méthode d'une classe en instanciant un objet d'exception personnalisé et en le préfixant du mot clef **raise**.

- Le mécanisme général d'interception des exceptions à travers des gestionnaires d'exceptions **try**…**except** s'applique à tous les types d'exceptions y compris les exceptions personnalisées.

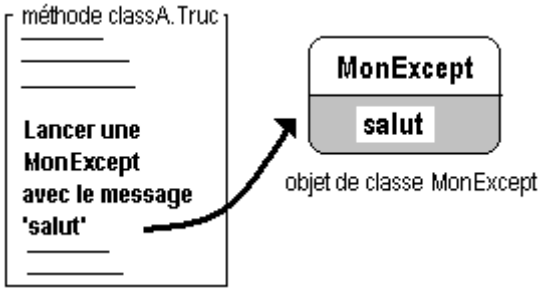
Création d'une nouvelle classe




```
Type
  MonExcept = class (Exception)

  End;
```

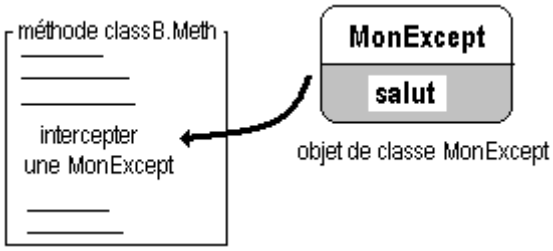
MonExcept hérite par construction de la propriété public **Message** de sa mère.

Lancer une MonExcept




```
Type
  MonExcept = class (Exception)

  End;

Procedure classA.Truc;
begin
   …..
 raise MonExcept.Create ( 'salut' );
  …..

end;
```

intercepter une MonExcept




```
MonExcept = class (Exception)…    end;

Procedure classB.Meth;
begin
  …..
   try….. // code à protéger
   except on MonExcept do begin
                …..  // code de réaction à l'exception
            end;
    end;
end;
```

Exercices chapitre 5

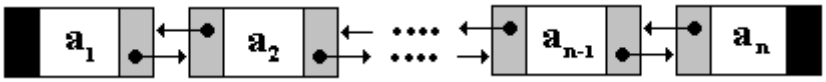
Ex-1 : Il est demandé de construire une classe de pile lifo ClassLifo héritant d'une Tlist (Un objet Tlist de Delphi, stocke un tableau de pointeurs, utilisé ici pour gérer une liste d'objets) et qui est réactive à l'empilement et au dépilement d'un objet. Nous proposons de suivre la démarche de la notice méthodologique du cours en nous inspirant de son code final pour construire deux événements dans la pile lifo et lui permettre de réagir à ces deux événements.

Ex-2 : Nous souhaitons développer rapidement un petit éditeur de texte qui nous permettra :

- de **lire** du texte à partir d'un fichier sur disque ou sur disquette (*ouvrir*),
- de **taper** du texte (*nouveau*),
- de **visualiser** le texte entré,
- d'effectuer sur ce texte les opérations classiques de *copier/ coller/ couper*,
- de le **sauvegarder** sur disque ou sur disquette (*enregistrer*).

Ex-3 : Nous reprenons la classe déjà construite ClassLifo de pile lifo héritant d'une Tlist réactive à l'empilement et au dépilement d'un objet, il est demandé de la rendre plus robuste en lui permettant lorsque la pile est vide de lancer une exception dépilement impossible.

Ex-4 : On définit la structure de données de liste chaînée double, qui est une liste chaînée pouvant se parcourir dans les deux sens, chaque maillon de la liste est lié à son suivant et à son précédent sauf les maillons situés aux extrémités de la liste; (a1, … , an) sont les données de la liste :



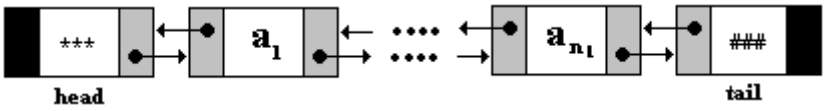
Implanter en delphi la classe TListeDble représentant une telle liste chaînée double et TcellDble un maillon de la liste (un maillon est donc un objet de calesse TcellDble), l'information du maillon est une chaîne de caractères.

```
TCellDble = class
  public
    info:string;
    constructor Créer (avant , apres:TCellDble;
elt:string);
    procedure InsererAvant (cell:TCellDble);
    procedure InsererApres (cell:TCellDble);
  private
    next:TCellDble;
    prec:TCellDble;
 end;
```

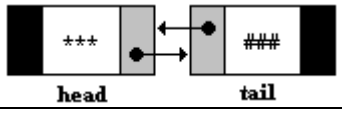
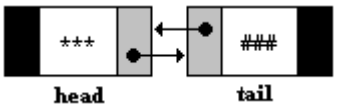
Les méthodes InsererAvant et InsererApres réalisent l'insertion (avant l'objet ou après l'objet)

```
TlisteDble =class
 public
  constructor Create;
  destructor Liberer;
  procedure AjouterLeft (elt:string);
  procedure AjouterRight (elt:string);
  procedure InsererLeft (rang:integer;elt:string);
  procedure InsererRight (rang:integer;elt:string);
  procedure SupprimerLeft (rang:integer);
  procedure SupprimerRight (rang:integer);
  function ElementFromLeft (rang:integer):string;
  function ElementFromRight (rang:integer):string;
  function IndexFromLeftOf (elt:string):integer;
  function IndexFromRightOf (elt:string):integer;
  function Tete:TCellDble;
  function Fin:TCellDble;
  function Longueur : integer;
  procedure Clear;
  function ParcourirLeft:string;
 private
   head , tail : TCellDble;
end;
```

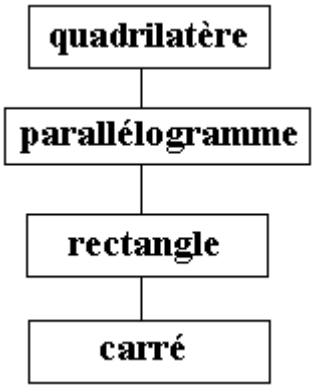
On propose pour simplifier l'écriture des algorithmes de parcours de la liste de mettre deux sentinelles ***head*** et ***tail*** bornant les deux "bouts" de la liste (cellules ne contenant de données significatives) :



Spécifications des méthodes à implanter :

constructor Create;	Crée une liste vide : 
destructor Liberer;	Libère la mémoire utilisée par la liste.
procedure AjouterLeft (elt : **string**);	Ajoute la donnée elt: **string** après head.
procedure AjouterRight (elt : **string**);	Ajoute la donnée elt: **string** avant tail.
procedure InsererLeft (rang:integer ; elt:**string**);	Insère la donnée elt: **string** au rang : integer , rang compté à partir de head (parcours à gauche).
procedure InsererRight (rang:integer ; elt:**string**);	Insère la donnée elt: **string** au rang : integer , rang compté à partir de tail (parcours à droite).
procedure SupprimerLeft (rang:integer);	Supprime la donnée de rang : integer , comptée à partir de head (parcours à gauche).
procedure SupprimerRight (rang:integer);	Supprime la donnée de rang : integer , comptée à partir de tail (parcours à droite).
function ElementFromLeft (rang:integer):**string**;	Renvoie la donnée de rang : integer , comptée à partir de head (parcours à gauche).
function ElementFromRight (rang:integer):**string**;	Renvoie la donnée de rang : integer , comptée à partir de tail (parcours à droite).
function IndexFromLeftOf (elt : **string**):integer;	Renvoie le rang de la position de la donnée elt : **string** compté à partir de head (parcours à gauche).
function IndexFromRightOf (elt : **string**):integer;	Renvoie le rang de la position de la donnée elt: **string** compté à partir de tail (parcours à droite).
function Tete:TCellDble;	Renvoie une référence sur head.
function Fin:TCellDble;	Renvoie une référence sur tail.
function Longueur : integer;	Fournit le nombre d'éléments utiles de la liste.
procedure Clear;	Remet la liste à vide : 
function ParcourirLeft : **string**;	Parcours des chaînes de la liste de head à tail en les concaténant entre elles et renvoie la chaîne unique obtenue.

Ex-5 : On définit la hiérarchie suivante dans les figures géometriques :



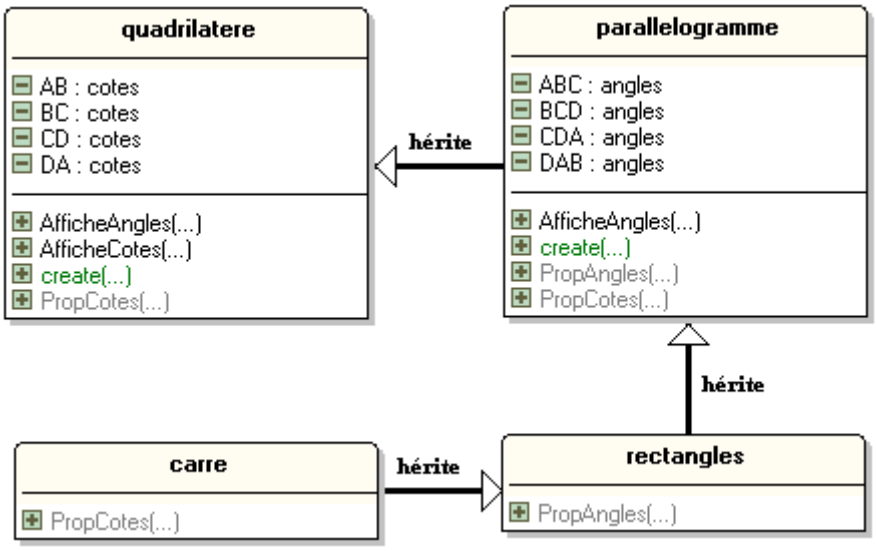
Un quadrilatère est une figure (A,B,C,D) possédant quatre côtés.

Un parallélogramme est un quadrilatère dont les côtés opposés sont parallèles et les angles opposés égaux.

Un rectangle est un parallélogramme dont tous les angles ont la même mesure : 90°

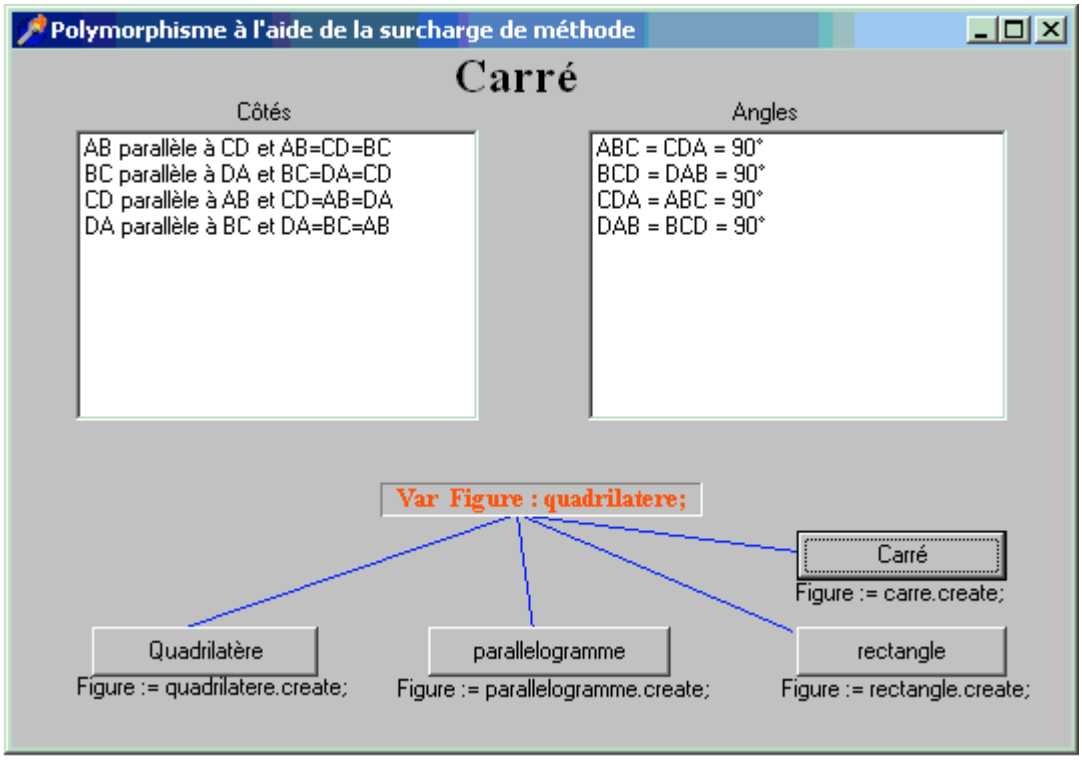
Un carré est un rectangle dont tous les côtés sont égaux.

Il est demandé d'implanter en Delphi la hiérarchie de classe selon les diagrammes UML ci-après :



Spécifications des méthodes à implanter	
type cotes = string; angles = string;	Les noms des côtés ou des sous forme de **string** angles (côtés :"AB", "CD", … ou angles : "ABC", "BCD",…).
function PropAngles (x:angles) **:** string;	Renvoie dans une **string** les propriétés d'un angle x : angles.
function PropCotes (x:cotes) **:** string;	Renvoie dans une **string** les propriétés d'un côté x : cotes.
procedure AfficheCotes (List **:** TlistBox);	Affiche dans un TlistBox les propriétés des 4 côtés.
procedure AfficheAngles (List **:** TlistBox);	Affiche dans un TlistBox les propriétés des 4 angles.

Il est demandé de construire une interface visuelle de test d'un objet de classe quadrilatère instancié à la demande selon l'une des trois classes filles (exemple ci-dessous d'une instanciation d'un quadrilatère en carré):



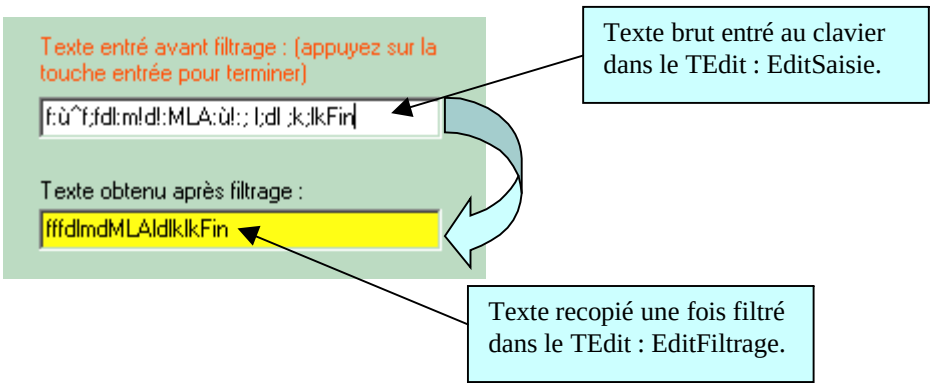
Ex-6 : Vous aurez à utiliser la classe TTimer qui encapsule les fonctions de timer de l'API Windows, on rappelle les propriétés utiles que cette classe contient :

```
property Enabled: Boolean; // Détermine si le timer répond aux événements timer.
property Interval: Cardinal; // Détermine l'intervalle de temps, exprimé en millisecondes, s'écoulant avant que le composant timer génère un autre événement OnTimer.
property OnTimer: TNotifyEvent; // Se produit quand le temps spécifié par la propriété Interval s'est écoulé.

unit UClassclickMemo;
// classe TMemoFlash

interface
uses stdctrls, extctrls, Graphics, classes, controls;

 type
 TMemoFlash = class (…)
     public
      …
     private
      …
  end;

implementation

end.
```

Questions :

Construire une classe visuelle complète TMemoFlash héritée des TMemo qui clignote 10 fois (il changera de couleur jaune-bleu, par intermittence de 100ms entre chaque flash) lorsque l'utilisateur clique avec la souris dans le composant. Un TMemoFlash doit avoir automatiquement comme parent son propriétaire et lors de sa création il affichera l'adresse mémoire de la fenêtre de son parent et celle de sa propre fenêtre.

Ex-7 : construire une IHM de filtrage d'un texte entré au clavier dans un TEdit et recopié dans un TMemo une fois filtré :



Le filtrage consiste à ne conserver dans EditFiltrage que les lettres majuscules et minuscules, et à exclure tout autre caractère lors de la recopie du texte entré dans EditSaisie.

Le filtrage doit s'effectuer à la volée (c'est à dire au fur et à mesure que l'on tape du texte au clavier dans EditSaisie) et lorsque l'utilisateur appui sur la touche entrée du clavier le texte qui a été filtré doit être rangé dans un TMemo nommé MemoApresFiltrage. Prévoir deux versions possibles selon que l'utilisateur est autorisé à utiliser la touche d'effacement arrière (backspace) ou non dans la saisie du texte et utiliser l'événement OnKeypress pour effectuer le filtrage à la volée.

Ex-1 Code solution pratique : une pile Lifo événementielle

```
unit ULifoEvent ;

interface
uses classes,Dialogs ;

type
  DelegateLifo = procedure ( Sender: TObject ;  s :string ) of  object ;

 ClassLifo = class (TList)
 private
    FOnEmpiler : DelegateLifo ;
    FOnDepiler : DelegateLifo ;
 public
   function  Est_Vide  : boolean ;
   procedure  Empiler (elt : string ) ;
   procedure  Depiler ( var  elt : string ) ;
   property  OnEmpiler : DelegateLifo  read  FOnEmpiler  write  FOnEmpiler ;
   property  OnDepiler : DelegateLifo  read  FOnDepiler  write  FOnDepiler ;
end;

ClassUseLifo = class
 public
  procedure  EmpilerListener( Sender: TObject ;  s :string ) ;
  procedure  DepilerListener( Sender: TObject ;  s :string ) ;
  constructor  Create ;
  procedure  main ;
end;

implementation

 procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
   if  assigned(FOnDepiler)  then
     FOnDepiler ( self ,elt )
   end
  end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , PChar(elt)) ;
 if  assigned(FOnEmpiler)  then
     FOnEmpiler ( self ,elt )
 end;
```

Le type de l'événement : type pointeur de méthode (2 paramètres)

Champs privés stockant la valeur de l'événement (pointeur de méthode).

Evénement OnEmpiler :
- pointeur de méthode.

Evénement OnDepiler :
- pointeur de méthode.

self = la pile Lifo

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnDepiler **pointe vers elle, il est donc non nul.**
Sinon FOnDepiler **est nul (non assigné)**

L'instruction FOnDepiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnDepiler **pointe.**

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnEmpiler **pointe vers elle, il est donc non nul.**
Sinon FOnEmpiler **est nul (non assigné)**

L'instruction FOnEmpiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnEmpiler **pointe.**

```
function  ClassLifo.Est_Vide : boolean ;
begin
  result :=  self.Count = 0 ;
end;

 { ClassUseLifo }

constructor  ClassUseLifo.Create ;
begin
  inherited;
end;

procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a depile : ' ,s) ;
end;

procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a empile : ' ,s) ;
end;

procedure  ClassUseLifo.main ;
var
  pileLifo : ClassLifo ;
  ch :string ;
 begin
   pileLifo := ClassLifo.Create ;
   pileLifo.OnEmpiler :=  EmpilerListener ;
   pileLifo.OnDepiler :=  DepilerListener ;
   pileLifo.Empiler( '[ eau ]' ) ;
   pileLifo.Empiler( '[ terre ]' ) ;
   pileLifo.Empiler( '[ mer ]' ) ;
   pileLifo.Empiler( '[ voiture ]' ) ;
   writeln ( 'Depilement de la pile :' ) ;
   while  not  pileLifo.Est_Vide  do
   begin
     pileLifo.Depiler(ch) ;
     writeln (ch) ;
   end;
   writeln ( 'Fin du depilement.' ) ;
   readln ;
 end;

end.
```

Classe de test créant une pile Lifo

Gestionnaire de l'événement OnDepiler : signature compatible avec DelegateLifo**.**

Gestionnaire de l'événement OnEmpiler : signature compatible avec DelegateLifo**.**

Méthode main **de test**

Instanciation d'une pile Lifo à tester.

Affectation de chaque gestionnaire à l'événement qu'il est chargé de gérer.

Empilement de 4 éléments

Dépilement de toute la pile

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
   execLifo := ClassUseLifo.Create;
   execLifo.main
end.
```

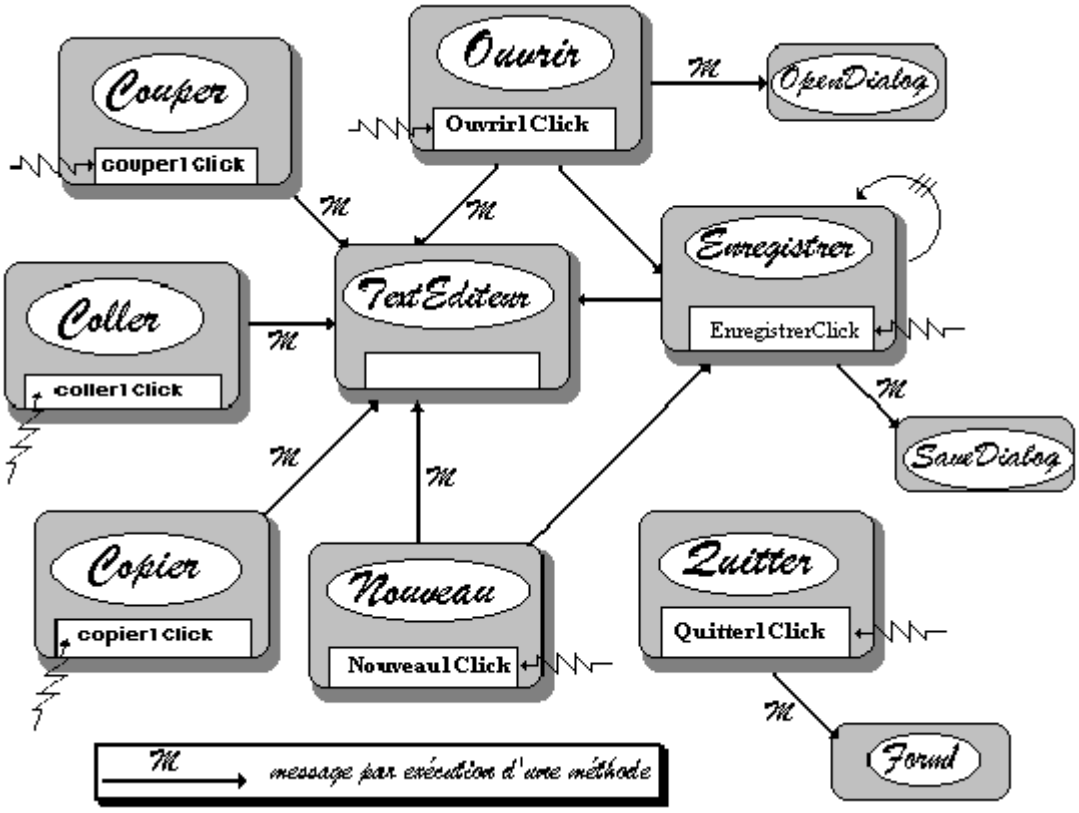
exécution

```
...rogram Files\Borland\Delphi7\Bin\Project
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
```

Ex-2 Solution détaillée : un éditeur de texte

1. Les choix à partir de l'énoncé

Les objets potentiels réagiront à ces événements selon le graphe ci-dessous



L'objet ***TextEditeur*** est l'objet central sur lequel agissent tous les autres objets, il contiendra le texte à éditer. En faisant ressortir les opérations à effectuer, l'utilisation de la notion d'abstraction est naturelle.

Nous envisageons les actions suivantes obtenues par réaction à un événement Click de souris (choix des objets du graphe précédent):

nouveau *(utilise un objet à définir)*
couper *(utilise un objet à définir)*
copier *(utilise un objet à définir)*
coller *(utilise un objet à définir)*
ouvrir *(utilise un objet de dialogue)*
enregistrer *(utilise un objet de dialogue)*
quitter *(utilise un objet prédéfini Form)*

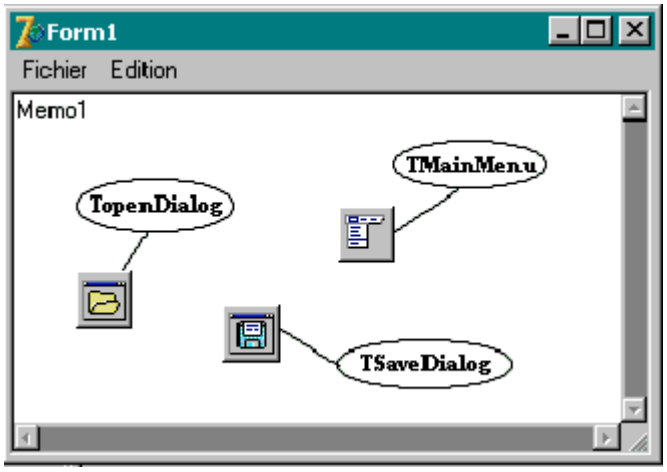
2. Les objets de composants

Delphi étant orienté objet nous allons exploiter complètement l'analyse présentée dans le graphe événementiel ci-haut en mettant en place quatre objets du genre composants visuels ou non visuels de Delphi.

L'interface choisie

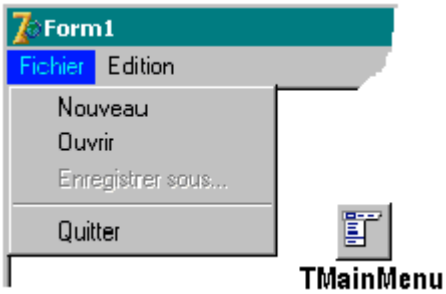
*Une fiche (**Form1**) comportant :*
- Un Tmemo avec deux ascenseurs

Trois composants non visuels :
- **TMainmenu** (une barre de 2 menus)
- **TopenDialog** (pour le chargement de fichier)
- **TSaveDialog** (pour la sauvegarde d'un texte)



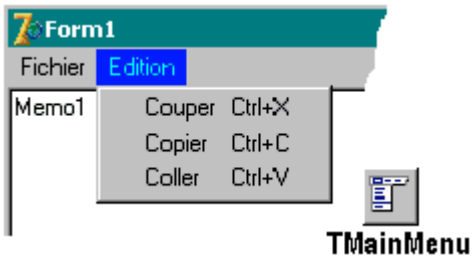
Le composant TMainmenu(1er menu)

comporte un premier menu *Fichier à 5 items*

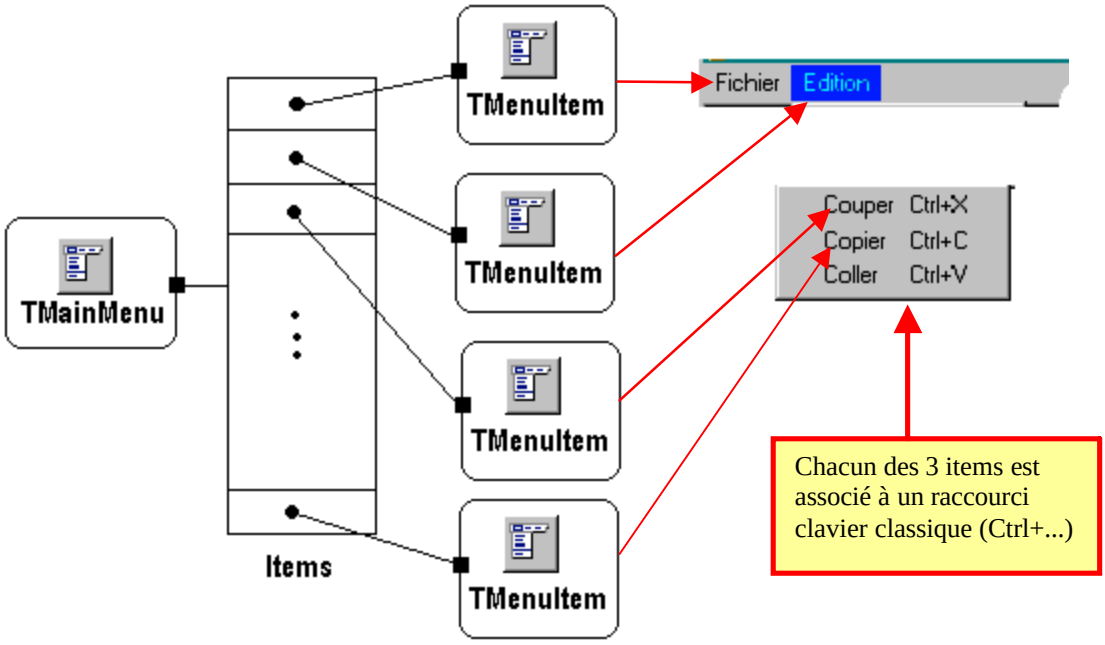


Le composant TMainmenu (2ème menu)

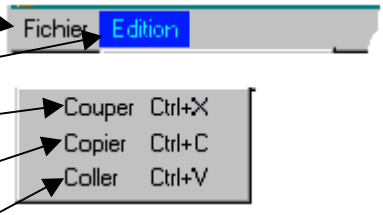
comporte un deuxième menu *Edition à 3 items.*



Les menus dans la barre et les sous-menus sont tous des objets de classe TmenuItem qui sont gérés à travers le champ Items d'un objet de classe TMainMenu :




```
object MainMenu1: TMainMenu
   ….
   object fichier1: TMenuItem
     Caption = 'fichier'
   end
   object edition1: TMenuItem
      Caption = 'edition'
      object couper1: TMenuItem
        Caption = 'couper'
      end
      object copier1: TMenuItem
        Caption = 'copier'
      end
      object coller1: TMenuItem
        Caption = 'coller'
      end
   end
end …..
```



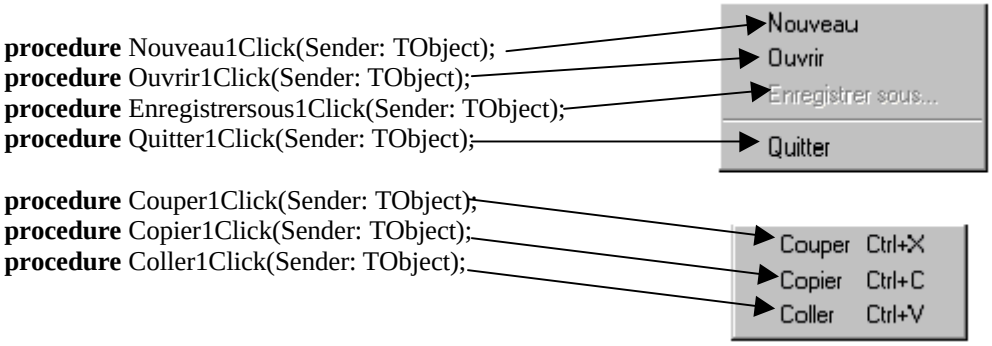
Correspondance entre le code caché par Delphi et l'inclusion des objets de TmenuItem dans un TmainMenu.

Mise en place de la gestion des événements.

A chacun des items utilisables de chacun des 2 menus, il nous faut associer un gestionnaire d'événement qui indique au programme comment il doit réagir lorsque l'utilisateur sélectionne un champ de l'un des menus par un click de souris. Nous avons vu que Delphi contient le mécanisme d'association des événements à nos gestionnaires. Beaucoup de contrôles (classes visuelles)et beaucoup d'autres classes non visuelles comme les TMenuItem, de Delphi sont sensibles à l'événement. de click de souris : **property** OnClick : TnotifyEvent.

Les gestionnaires de l'événement OnClick pour les TMenuItem

Voici les en-têtes des 7 gestionnaires de OnClick automatiquement construits :



Voici pour chaque gestionnaire le code écrit par le programmeur.

Le code du gestionnaire *Quitter1Click*

```
procedure TForm1.Quitter1Click(Sender: TObject);
begin
  Close; //écrit par le développeur (fermeture de la fenêtre)
end;
```

Le code du gestionnaire *Ouvrir1Click*

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
begin
  if  OpenDialog1.Execute then //écrit par le développeur
  begin
    Enregistrersous1.enabled:=true;
    TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
  end
end;
```

Le code du gestionnaire *Enregistrersous1Click*

```
procedure TForm1.Enregistrersous1Click(Sender: TObject);
begin
  if  SaveDialog1.Execute then // écrit par le développeur
  begin
    Enregistrersous1.enabled:=false;
    TextEditeur.Lines.SaveToFile( SaveDialog1.FileName );
  end
end;
```

Le code du gestionnaire *Couper1Click*

```
procedure TForm1.Couper1Click(Sender: TObject);
begin
  TextEditeur.CutToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Copier1Click*

```
procedure TForm1.Copier1Click(Sender: TObject);
begin
  TextEditeur.CopyToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Coller1Click*

```
procedure TForm1.Coller1Click(Sender: TObject);
begin
  TextEditeur.PasteFromClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Nouveau1Click*

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  TextEditeur.Clear; &not; écrit par le développeur
  Enregistrersous1.enabled:=true &not; écrit par le développeur
end;
```

Il est à noter que nous avons écrit au total 16 lignes de programme Delphi pour construire notre micro-éditeur. Ceci est rendu possible par la réutilisabilité de méthodes déjà intégrées dans les objets de Delphi et en fait présentes dans le système Windows.

Vous pouvez voir la puissance de ce **RAD** lorsque vous observez par exemple l'instruction qui a permis de faire exécuter le "copier" ou le "chargement d'un fichier" :

```
TextEditeur.CopyToClipboard;
```

La méthode **CopyToClipboard** s'applique à l'objet **TextEditeur** qui est de la classe TMemo; cette instruction correspond à un ensemble complexe d'opérations de bas niveau :
le TMemo autorise une suite complexe d'opérations permettant de sélectionner un texte écrit sur plusieurs lignes du TMemo. Il les affiche en surbrillance et la méthode **CopyToClipboard** récupère le texte sélectionné puis le recopie enfin dans le presse-papier du système.

```
TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
```

Nous n'avons par ailleurs eu aucun code particulier à écrire pour le chargement de fichier texte, la méthode **LoadFromFile** présente dans l'objet **Lines** (de classe TStrings) effectue toutes les actions.

A titre de travail personnel il est recommandé d'enrichir ce micro-éditeur avec la possibilité de changer la police de caractère, la couleur du fond etc…

Ex-3 Code solution pratique : une pile Lifo avec exception

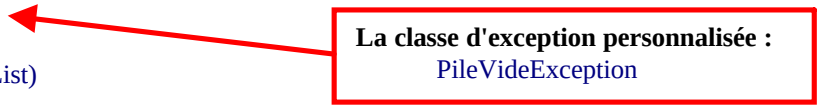
Création d'une nouvelle classe

```
PileVideException = class (Exception)
end;
```

Lancer une PileVideException

```
procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    ……
  end
  else raise PileVideException.Create ('Impossible de dépiler: la pile est vide' )
 end;
```

Code complet de la pile




```
unit  ULifoEvent ;

 interface
 uses  classes,Dialogs, SysUtils ;

 type
   DelegateLifo = procedure ( Sender: TObject ;  s :string ) of  object ;

   PileVideException = class (Exception)
   end;

  ClassLifo = class (TList)
  private
     FOnEmpiler : DelegateLifo ;
     FOnDepiler : DelegateLifo ;
  public
    function  Est_Vide  : boolean ;
    procedure  Empiler (elt : string ) ;
    procedure  Depiler ( var  elt : string ) ;
    property  OnEmpiler : DelegateLifo  read  FOnEmpiler  write  FOnEmpiler ;
    property  OnDepiler : DelegateLifo  read  FOnDepiler  write  FOnDepiler ;
end;

ClassUseLifo = class
 public
   procedure  EmpilerListener( Sender: TObject ;  s :string ) ;
   procedure  DepilerListener( Sender: TObject ;  s :string ) ;
   constructor  Create ;
   procedure  main ;
end;

implementation
```

```
procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
    if  assigned(FOnDepiler)  then
      FOnDepiler ( self ,elt )
   end
   else raise PileVideException.Create ('Impossible de dépiler: la pile est vide' )
  end;

procedure  ClassLifo.Empiler(elt : string ) ;
begin
  self.Insert(0 , PChar(elt)) ;
  if  assigned(FOnEmpiler)  then
     FOnEmpiler ( self ,elt )
end;

function  ClassLifo.Est_Vide : boolean ;
begin
   result :=  self.Count = 0 ;
end;

 { ClassUseLifo }

constructor  ClassUseLifo.Create ;
begin
   inherited;
end;

procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a depile : ' ,s) ;
end;

procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a empile : ' ,s) ;
end;

procedure  ClassUseLifo.main ;
var
  pileLifo : ClassLifo ;
  ch :string ;
 begin
   pileLifo := ClassLifo.Create ;
   pileLifo.OnEmpiler :=  EmpilerListener ;
   pileLifo.OnDepiler :=  DepilerListener ;
   pileLifo.Empiler( '[ eau ]' ) ;
   pileLifo.Empiler( '[ terre ]' ) ;
   pileLifo.Empiler( '[ mer ]' ) ;
   pileLifo.Empiler( '[ voiture ]' ) ;
   writeln ( 'Depilement de la pile :' ) ;
   while  not  pileLifo.Est_Vide  do
    begin
      pileLifo.Depiler(ch) ;
```

Instanciation d'un objet de type :
PileVideException
Et lancement de cet objet d'exception.

```
      writeln (ch) ;
    end;
  writeln ( 'Fin du depilement.' ) ;
  try
     pileLifo.Depiler(ch) ;
     writeln (ch) ;
   except on Ex : PileVideException do begin
            writeln ( Ex.Message ) ;
          end;
  end;
  readln ;
end;

end.
```

On tente de dépiler alors que la pile a été entièrement vidée !

Interception d'un objet Ex de type : PileVideException
Et gestion de son Message.

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
  execLifo := ClassUseLifo.Create;
  execLifo.main
end.
```

exécution

```
C:\Program Files\Borland\Delphi7\Bin\Project2.exe
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
Impossible de dúpiler: la pile est vide
```

Ex-4 Code solution pratique : liste chaînée double

```
unit UDblListChn;

 interface
```

```
type

 TCellDble = class
public
 info:string;
 constructor Creer(avant,apres:TCellDble;elt:string);
 procedure InsererAvant(cell:TCellDble);
 procedure InsererApres(cell:TCellDble);
private
 next:TCellDble;
 prec:TCellDble;
end;
```

```
TListeDble=class
 public
  constructor Create;
  destructor Liberer;
  procedure AjouterLeft(elt:string);
  procedure AjouterRight(elt:string);
  procedure InsererLeft(rang:integer;elt:string);
  procedure InsererRight(rang:integer;elt:string);
  procedure SupprimerLeft(rang:integer);
  procedure SupprimerRight(rang:integer);
  function ElementFromLeft(rang:integer):string;
  function ElementFromRight(rang:integer):string;
  function IndexFromLeftOf(elt:string):integer;
  function IndexFromRightOf(elt:string):integer;
  function Tete:TCellDble;
  function Fin:TCellDble;
  function Longueur:integer;
  procedure Clear;
  function ParcourirLeft:string;
 private
  head,tail:TCellDble;
  Long:integer;
  function CellFromLeftAt(rang:integer):TCellDble;
  function CellFromRightAt(rang:integer):TCellDble;

end;
```

```
implementation

 { TCellDble }
```

```
constructor TCellDble.Créer (avant,apres:TCellDble;
                             elt:string );
 begin
  self.next:=apres;
  self.prec:=avant;
  self.info:=elt;
 end;

 procedure TCellDble.InsererApres(cell:TCellDble);
 var cellNext:TCellDble;
 begin
  cellNext:= cell.next;
  self.next:=cellNext;
  cell.next:=self;
  cellNext.prec:=self;
  self.prec:=cell
 end;
```

```
procedure TCellDble.InsererAvant(cell:TCellDble);
 var cellPrec:TCellDble;
 begin
  cellPrec:= cell.prec;
  self.prec:=cellPrec;
  cell.prec:=self;
  cellPrec.next:=self;
  self.next:=cell
 end;
```

```
constructor TListeDble.Create; { TListeDble }
 begin
  head:=TCellDble.Créer (nil,nil,'***');
  tail:=TCellDble.Créer (nil,nil,'###');
  head.next:=tail;
  tail.prec:=head;
  Long:=0;
 end;

 destructor TListeDble.Liberer;
 begin
  self.Clear;
  self.head.Free;
  self.tail.Free;
 end;

 function TListeDble.ElementFromLeft(rang: integer):
string;
 var L,Loc:TCellDble;
  compte:integer;
 begin
  if (rang<=Long)and(rang>0) then
   result:=CellFromLeftAt(rang).info
  else
   result:='?'
 end;

 function TListeDble.ElementFromRight(rang:
integer): string;
 var L,Loc:TCellDble;
  compte:integer;
 begin
  if (rang<=Long)and(rang>0) then
   result:=CellFromRightAt(rang).info
  else
   result:='?'
 end;

function TListeDble.IndexFromLeftOf(elt: string):
integer;
 var i,index:integer;
  L:TCellDble;
 begin
  index:=0;
  L:=self.head;
  for i:=1 to long+1 do   begin
   if L.info=elt then
    break
   else
   begin
    L:=L.next;
    index:=index+1
   end
  end;
  if index>long then
   index:=0;
  result:=index
 end;
```

```
function TListeDble.IndexFromRightOf(elt: string):
integer;
 var i,index:integer;
  L:TCellDble;
 begin
  index:=0;
  L:=self.tail;
  for i:=1 to long+1 do   begin
   if L.info=elt then
    break
   else
   begin
    L:=L.prec;
    index:=index+1
   end
  end;
  if index>long then
   index:=0;
  result:=index
 end;

procedure TListeDble.InsererLeft(rang: integer;
          elt: string);
var Loc:TCellDble;
begin
 if (rang<=Long)and(rang>0) then
 begin
  Loc:=TCellDble.Creer(nil,nil,elt);
  Loc.InsererAvant(CellFromLeftAt(rang));
  Long:=Long+1;
 end
end;

 procedure TListeDble.InsererRight(rang: integer; elt:
string);
 var Loc:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   Loc:=TCellDble.Creer(nil,nil,elt);
   Loc.InsererApres(CellFromRightAt(rang));
   Long:=Long+1;
  end
 end;

 procedure TListeDble.SupprimerLeft(rang: integer);
 var Loc,Lprec,Lnext:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   Loc:=CellFromLeftAt(rang);
   Lnext:=Loc.next;
   Lprec:=Loc.prec;
   Lnext.prec:=Lprec;
   Lprec.next:=Lnext;
   Loc.Free;
   Long:=Long-1
  end
 end;
```

```
procedure TListeDble.SupprimerRight(rang: integer);
 var Loc,Lprec,Lnext:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   rang:=long-rang+1;
   self.SupprimerLeft(rang)
  end
 end;
function TListeDble.Fin: TCellDble;
 begin
  result:=self.tail
 end;

 function TListeDble.Tete: TCellDble;
 begin
  result:=self.head;
 end;

 procedure TListeDble.Clear;
 var L,Loc:TCellDble;
 begin
  if Long>0 then
  begin
   L:=self.head;
   while Assigned(L) do   begin
    Loc:=L;
    L:=L.next;
    if (Loc<>head)and(Loc<>tail)then
   //on n'efface pas la tête et la fin
    begin
     Loc.Free ;
    end
   end;
   head.next:=tail;
   tail.prec:=head;
   Long:=0;
  end
 end;

 function TListeDble.Longueur: integer;
 begin
  result:=Long
 end;

function TListeDble.ParcourirLeft:string;
 var L:TCellDble;
  sortie:string;
  begin
   L:=self.head;
   sortie:='';
   while Assigned(L) do   begin
    sortie:=sortie+L.info;
    L:=L.next;
   end;
   result:=sortie
  end;
```

```
procedure TListeDble.AjouterLeft(elt: string);
 var L,Loc:TCellDble;
 begin
  L:=self.head;
  Loc:=TCellDble.Creer(L,L.next,elt);
  L.next.prec:=Loc;
  L.next:=Loc;
  Long:=Long+1;
 end;

 procedure TListeDble.AjouterRight(elt: string);
 var L,Loc:TCellDble;
 begin
  L:=self.tail;
  Loc:=TCellDble.Creer(L.prec,L,elt);
  L.prec.next:=Loc;
  L.prec:=Loc;
  Long:=Long+1;
 end;

 function TListeDble.CellFromLeftAt(rang: integer):
TCellDble;
 var L,Loc:TCellDble;
  compte:integer;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   L:=self.head;
   for compte:=1 to rang+1 do begin
  //la tete compte pour une cellule
    Loc:=L;
    L:=L.next;
   end;
   result:=Loc
  end
  else
   result:=nil
 end;

 function TListeDble.CellFromRightAt(rang: integer):
TCellDble;
 var L,Loc:TCellDble;
  compte:integer;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   L:=self.tail;
   for compte:=1 to rang+1 do begin
  //la fin compte pour une cellule
    Loc:=L;
    L:=L.prec;
   end;
   result:=Loc
  end
  else
   result:=nil
 end;
end. // fin unit UDblListChn
```

Ex-5 Code solution pratique : hiérarchie de quadrilatères

```
unit UClassQuadrilat;

 interface
 uses stdctrls;
```

```
type
 cotes=string;
 angles=string;

 quadrilatere=class
private
 AB,BC,CD,DA:cotes;
public
 function PropCotes (x:cotes):string; virtual;
 //--statique car elle sert à tous :
 procedure AfficheCotes (List:TlistBox);
 procedure AfficheAngles (List:TlistBox); virtual;
 constructor create; virtual;
end;
```

```
Parallelogramme = class (quadrilatere)
 private
  ABC,BCD,CDA,DAB:angles;
 public
  function PropCotes(x:cotes):string;override;
  function PropAngles(x:angles):string; virtual;
  procedure AfficheAngles(List:TlistBox);override;
  constructor create;override;
end;

rectangles = class (parallelogramme)
//-- rectangle est déjà une function existante en Delphi!
 public
  function PropAngles(x:angles):string;override;
end;

carre = class (rectangles)
 public
  function PropCotes(x:cotes):string;override;
end;
```

```
implementation
```

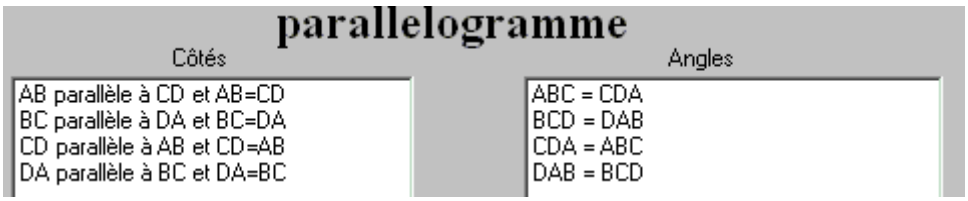
```
////////////////// CLASSE QUADRILATERE ////////////////
 constructor quadrilatere.create;
 begin
  AB:='AB';
  BC:='BC';
  CD:='CD';
  DA:='DA';
 end;

 function quadrilatere.PropCotes(x:cotes):string;
 begin
  result:=x+' : quelconque'
 end;

 procedure quadrilatere.AfficheCotes(List:TlistBox);
 begin
  List.Clear;
  List.Items.Add(PropCotes(AB));
  List.Items.Add(PropCotes(BC));
  List.Items.Add(PropCotes(CD));
  List.Items.Add(PropCotes(DA));
 end;

 procedure quadrilatere.AfficheAngles(List:TlistBox);
 begin
  List.Clear;
  List.Items.Add('angles quelconques')
 end;
```

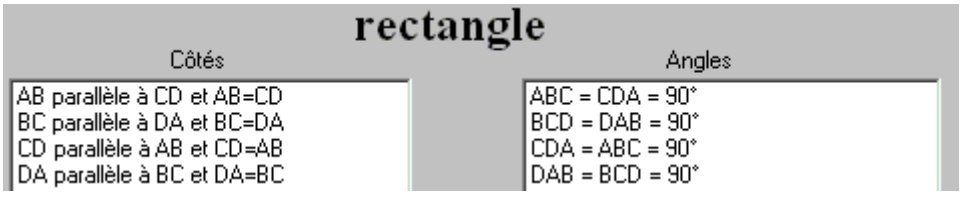
```
//////// CLASSE PARALLELOGRAMME  ///////////
 constructor parallelogramme.create;
 begin
  inherited;
  ABC:='ABC';
  BCD:='BCD';
  CDA:='CDA';
  DAB:='DAB';
 end;

 function parallelogramme.PropCotes(x:cotes):string;
 begin
  if x='AB' then
   result:=x+' parallèle à CD et AB=CD'
  else
   if x='BC' then
    result:=x+' parallèle à DA et BC=DA'
   else
    if x='CD' then
     result:=x+' parallèle à AB et CD=AB'
    else
     if x='DA' then
      result:=x+' parallèle à BC et DA=BC'
 end;
```

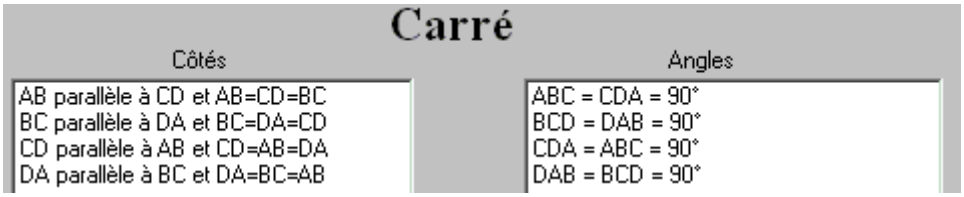
```
// CLASSE PARALLELOGRAMME  (suite) //
function parallelogramme.PropAngles
                              (x:angles):string;
 begin
  if x='ABC' then
   result:='ABC = CDA'
  else
   if x='BCD' then
    result:='BCD = DAB'
   else
    if x='CDA' then
     result:='CDA = ABC'
    else
     if x='DAB' then
      result:='DAB = BCD'
 end;

 procedure parallelogramme.AfficheAngles
                              (List:TlistBox);
 begin
  List.Clear;
  List.Items.Add(PropAngles(ABC));
  List.Items.Add(PropAngles(BCD));
  List.Items.Add(PropAngles(CDA));
  List.Items.Add(PropAngles(DAB));
 end;
```

```
////////////////// CLASSE RECTANGLE ///////////////////
 function rectangles.PropAngles(x:angles):string;
 var s:string;
 begin
  s:=inherited PropAngles(x);
  result:=s+' = 90°'
 end;

////////////////// CLASSE CARRE ///////////////////
 function carre.PropCotes(x:cotes):string;
 var s:string;
 begin
  s:=inherited PropCotes(x); {celui de parallélogramme:
première classe ancêtre où il est surchargé ou défini }
  if (x='AB') then
   result:=s+'=BC'
  else
   if(x='bc')then
    result:=s+'=CD'
   else
    if(x='cd')then
     result:=s+'=DA'
    else
     if(x='da') then
      result:=s+'=AB'
 end;

 end.
```

Exemples d'affichages obtenus pour un objet quadrilatère instancié dans chaque type :







Ex-6 Code solution pratique : un TMemoFlash qui clignote

Une unité d'IHM qui instancie un TmemoFlash lorsque l'on clique sur le bouton Button1 :


```
unit UClassclickMemo;

 interface

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

 type
  TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
 private
 { Private declarations }
 public
 { Public declarations }
end;

TMemoFlash=class(TMemo)
 private
  Tempo:TTimer;
  Timing:integer;
  procedure flashMemo(Sender : TObject);
  procedure ClickMemo(Sender : TObject);
 public
  constructor Create(AOwner: TComponent);
  destructor Destroy;
end;

 var
  Form1: TForm1;

implementation

 {$R *.dfm}
 constructor TMemoFlash.Create(AOwner: TComponent);
 begin
  inherited;
  self.Parent:=TWinControl(AOwner);
  self.Lines.Append('adresse Owner = '+inttostr((AOwner as TWincontrol).Handle));
  self.Lines.Append('adresse Memo = '+inttostr(self.Handle));
  Tempo:=TTimer.Create(AOwner);
```

```
 Tempo.Enabled:=false;
 Tempo.Interval:=100;
 Tempo.OnTimer:= flashMemo;
 self.OnClick:= ClickMemo;
end;

destructor TMemoFlash.Destroy;
begin
 Tempo.Free;
 inherited;
end;

procedure TMemoFlash.flashMemo(Sender : TObject);
begin
 Timing:=Timing+1;
 if Timing<10 then
  if self.color=clyellow then
   self.color:=claqua
  else
   self.color:=clyellow
 else
  tempo.enabled:=false
end;

procedure TMemoFlash.ClickMemo(Sender : TObject);
begin
 Timing:=0;
 Tempo.Enabled:=true;
end;

//-----------------------------------------------------------------//
procedure TForm1.Button1Click(Sender: TObject);
var x:TMemoFlash;
begin
 x:=TMemoFlash.Create(self);
end;

end.
```

Ex-7 Code solution pratique : Filtrage avec KeyPress

```
unit UKeyPress;
//permet le filtrage de caractères en mode clavier à partir d'un TEdit
//(effacement possible seulement à la fin du texte entré)
interface

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

 type
  TForm1 = class(TForm)
  EditSaisie: TEdit;
  MemoApresFiltrage: TMemo;
  EditFiltre: TEdit;
  Label1: TLabel;
  Label2: TLabel;
```

```
 Label3: TLabel;
 procedure EditSaisieKeyPress(Sender: TObject; var Key: Char);
 procedure EditSaisieClick(Sender: TObject);
 procedure FormCreate(Sender: TObject);
private
{ Déclarations privées }
public
{ Déclarations publiques }
 LeTexte:string; //contient le texte entré au clavier
 procedure TexteSansBackSp(var Entree:string;car:char);
 procedure TexteAvecBackSp(var Entree:string;car:char);
end;

var
 Form1: TForm1;

implementation

{$R *.dfm}


procedure TForm1.TexteSansBackSp(var Entree:string;car:char);
//ne traite pas le backspace
begin
 if car<>#13 then
 begin
  if car in ['a'..'z']+['A'..'Z'] then
  begin
   Entree:=Entree+car;
   EditFiltre.Text:=Entree
  end
 end
 else
 begin
  MemoApresFiltrage.Lines.Add(Entree);
  Entree:='';
  EditSaisie.Text:=''
 end
end;
```


```
procedure TForm1.TexteAvecBackSp(var Entree:string;car:char);
//traitement du backspace à partir de la fin du texte
begin
 if (ord(car)=8)and(length(EditSaisie.Text)<>0) then
  if EditSaisie.Text[length(EditSaisie.Text)] in ['a'..'z']+['A'..'Z'] then
  begin
   delete(Entree,length(Entree),1); //on efface le dernier caractère
   EditFiltre.Text:=Entree        //on recopie le nouveau texte
  end;
 TexteSansBackSp(Entree,car)
end;

{-------------------------------------------------------------------------------}
procedure TForm1.EditSaisieKeyPress(Sender: TObject; var Key: Char);
//filtrage des lettres non accentuées seulement
begin
 TexteSansBackSp(LeTexte,Key);
 //TexteAvecBackSp(LeTexte,Key);
end;
```

```
procedure TForm1.EditSaisieClick(Sender: TObject);
begin
 EditSaisie.SelStart:= length(EditSaisie.text) //replace systématiquement le curseur à la fin
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 EditSaisie.SetFocus
end;

End.
```