DELPHI VCL/FMX

XE7+

Extraits support de Cours

Auteur : Alain Weber

1 Retour sur les principes de programmation par classes et objets

1.1 Comparaison entre Pascal Orienté Objet et POO

Un peu d'histoire. Parlons de ces langages que sont: Smalltalk, Objective C, Pascal Objet

- **Smalltalk** est un langage de programmation orienté objet, réflexif et dynamiquement typé

(*réflexif : capacité d'un programme à examiner, et éventuellement à modifier, ses propres structures internes de haut niveau lors de son exécution*).

Fin1970, début 1980 passage de la programmation structurée à la programmation orientée objet pour résoudre la compexité croissante des programmes.

- **Objective C** se présente comme une surcouche du langage C pour permettre la création et la manipulation d'objets, en reprenant certains concepts du langage Smalltalk-80.
- **Pascal objet** tel qu'implémenté dans **Turbo Pascal** prédécesseur de Delphi reprend donc les techniques et évolutions d' **Objective C**, **Smalltalk, C++**

Aujourd'hui, la **POO** programmation par objets est vue davantage comme un paradigme, le **paradigme objet,** que comme une simple technique de programmation. C'est pourquoi, lorsque l'on parle de nos jours de programmation par objets, on désigne avant tout la partie codage d'un modèle à objets obtenu par Analyse Orientée Object.

Même si ce mode de programmation est aussi possible avec Delphi. La POO permet donc entre autres de penser objet et d'implémenter le code en faisant communiquer entre eux plusieurs OS, plusieurs Machines, plusieurs Langages. Au sein de Delphi **Com/DCom, Corba, Midas et Datasnap** vont dans sens

*Ne pas oublier que dans un programme Oject Pascal tout est décrit dans le bloc de programme. Nous sommes dans un **espace collaboratif** ou tout est décrit et accessible dans l'executable. Dans les autres modèles lié à la POO nous sommes dans une **collaboration distante** avec par exemple des objets contenus dans des librairies externes ou la notion de **sécurité d'accès** a aussi son importance*

Il ne faut donc surtout pas faire de confusion entre les mots réservés Delphi tels que **class** et **object** d'un côté, et les termes **classe** et **objet** issus de la programmation orientée objet (POO) de l'autre.

1.2 Classes et objets

En POO, la description d'un **objet** est une **abstraction** ayant des limites claires et un sens précis dans le contexte d'un problème étudié. Tout objet possède une identité propre et peut être distingué des autres.
La **classe** permet de décrire de manière abstraite les **attributs** et le **comportement** communs d'un ensemble d'objets. Dans une classe les **propriétés** décrivent les attributs et les **méthodes** les comportements. Un **objet** est l'instance d'une **classe**.

Dans Delphi, la classe une fois instanciée via une variable du type de cette classe possède un espace mémoire réservé. La mise en oeuvre des **propriétés** est décrite plus loin. Pour l'instant nous parlerons de **champs**

Remarque : Il faut comprendre la réflexion ayant présidée à l'établissement du modèle objet. En programmation procédurale classique, la manière de travailler est orientée traitement. On bâtit un programme en créant et enchaînant des fonctions. Les données passent au second plan. Le modèle **objet** permet de rétablir l'équilibre entre **données** et **traitements** et de mieux traiter les évolutions progressives du modèle de données et des traitements associés. Il devient alors beaucoup plus simple de maintenir la cohésion du programme.

Trois notions sont associées à la programmation par objet : l'**encapsulation**, l'**héritage** et le **polymorphisme**. Voyons les dans l'univers Pascal Object.

Reprenons notre exemple du vélo. L'**encapsulation** permet de définir le vélo dans une structure fermée.
Un vélomoteur reprend les caractéristiques du vélo avec, en plus, principalement un moteur et un réservoir. Il **hérite** donc des attributs et fonctionnalités de son ancêtre. Mais vélo et vélomoteur n'avancent pas de la même manière. Dans un cas le cycliste fournit l'énergie motrice dans l'autre c'est le moteur qui met en oeuvre cette énergie. En d'autres termes on **avance** mais pas de la même manière. La gestion du **polymorphisme** (Emprunté au grec et qui signifie "qui a plusieurs formes") permet de résoudre cette problématique.
*La gestion du **polymorphisme** dans certain cas particuliers permet même de repasser d'un objet à l'autre. Si le vélomoteur tombe en panne, il ne reste plus qu'à pédaler pour avancer ...*

Voyons en détail ces trois notions…

Encapsulation
Le principe de l'encapsulation en Object Pascal est de regrouper les données et les traitements associés dans une même structure. Il y a deux types d'objets dans Delphi. Le type **object** et le type **class**. Le type **object** a été présenté brièvement plus haut et ne doit normalement plus être utilisé. Nous présentons ici le type **class.** Une classe est normalement définie en deux étapes dans les parties de l'unité respectivement nommées **interface** et **implémentation**. La partie **interface** défini la description de la structure principalement à a l'aide de **propriétés** et de **méthodes.** La partie **implémentation** décrit les traitements réalisés dans ces **méthodes.**

Dans l'exemple suivant la classe TVelo décrit dans un premier temps la manière d'avancer d'un vélo dans un espace à une dimension. Pour définir cette classe contenant données et code on écrira :

```
type
  TVelo = class
    position: integer;
    procedure avancer;
  end;

procedure TVelo.avancer;
begin
  position := position + 1;
end;
```

ex04d_class.dpr
La procédure **avancer** est une **méthode** de la classe TVelo. Cette méthode permet d' accéder au **champ** position de la classe TFigure.

Pour utiliser cette classe vous devez l'instancier. Deux étapes : déclarer une variable du type de cette classe, puis instancier cette variable :

```
var
  monVelo: TVelo;

begin
  monVelo := TVelo.Create;
  monVelo.position := 5;
  monVelo.avancer;
  writeln(monVelo.position);
```

```
  monVelo.Free;
  readln;
end.
```

ex04d_class.dpr

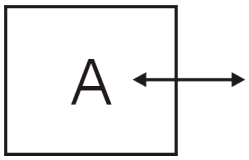
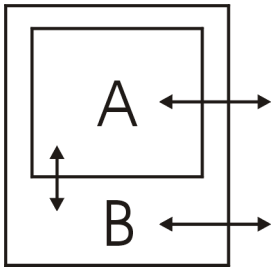
Class et **Class(Tobject)** sont équivalent en Delphi. **Create** est un "**constructor",** une méthode de base fournie par la classeTObject (élément primordial à tous les classes d'objets décrits dans Delphi). Cette fonction qui instancie la classe en un objet en

- réservant l'espace mémoire nécessaire correspondant à l'objet,
- chargeant cet espace avec la structure de l'objet,
- retournant l'adresse de cet objet.

Ici cette adresse est ensuite stockée dans la variable **monVelo** qui est en réalité un pointeur sur le début de l'espace mémoire réservé. Le **déréférencement** (décrit plus loin) permet d'éviter une écriture du type
monVelo^.avancer
monVelo.Free permet de détruire l'objet, c'est-à-dire libérer l'espace mémoire. **Free** est est un "**destructor",** une méthode de base fournie aussi par la classe **Tobject**.

FreeAndNil (monVelo) peut remplacer **monVelo.Free**. Cette opération permet de mettre à **Nil** la variable **monVelo**. Cette opération est utile lorsque la variable est réutilisée plusieurs fois. Regardez le détail de cette procédure dans l'unité **SysUtils**. Elle est riche d'enseignements

1.3 Héritage

Représentation	Description
	L'objet A communique avec son environnement par le biais de son **Interface** matérialisée dans cette figure par une double flèche. Le rectangle représente l'**encapsulation**
	L'objet B encapsule ici et ses propres propriétés et méthodes et reprend celles de l'objet A. cette reprise ce nomme **héritage**

L'**héritage** est la possibilité de pouvoir bénéficier des propriétés et des méthodes d'une classe ancêtre. Par exemple un vélomoteur est un vélo particulier. **En object Pascal, un objet ne peut avoir qu'un seul ancêtre immédiat, mais il peut avoir plusieurs descendants directs**.

Nous allons définir une nouvelle classe **TVeloMoteur** héritant de la classe **TVelo** par **TVeloMoteur** = class (**TVelo**)

```
type
  TVelo = class
    position: integer;
    procedure avancer;
  end;

  TVeloMoteur = class(TVelo)
    Reservoir : integer ;
    procedure avancer;
  end;

procedure TVelo.avancer;
begin
  position := position + 1;
end;

procedure TVeloMoteur.avancer;
begin
  position := position + 10 ;
  reservoir := reservoir - 1 ;
end;
```

ex04e_heritage.dpr

La classe **TVeloMoteur** a donc deux champs **Reservoir** etle champ **Position** de la classe **TVelo** dont elle hérite. Pour utiliser cette nouvelle classe, il nous faut instancier un objet de type **TVeloMoteur**:

```
var
  monVeloMoteur: TVeloMoteur;

begin
  monVeloMoteur := TVeloMoteur.Create;
  monVeloMoteur.Reservoir := 60 ;
  monVeloMoteur.position := 5;
  monVeloMoteur.avancer;
  writeln(monVeloMoteur.position);
  writeln(monVeloMoteur.Reservoir);
  monVeloMoteur.Free;
  readln;
end.
```

ex04e_heritage.dpr

La méthode **Avancer** de la classe **TVeloMoteur** surcharge la méthode **Avancer** de la classe ancêtre **TVelo**, c'est-à-dire que la méthode **Avancer** de la classe **TVeloMoteur** cache la méthode **Avancer** de la classe ancêtre **TVelo**. Cependant vous pouvez y accéder en appelant explicitement la méthode de la classe ancêtre **TVelo**.**Avancer**, ou mieux encore en utilisant le mot réservé inherited devant la méthode de la classe ancêtre **Tvelo**. Ce qui donnerait dans notre exemple :

procedure TVeloMoteur.avancer;
begin
  inherited avancer ;
  position := position + 10 ;
  reservoir := reservoir - 1 ;
end;
	La valeur de Position est donc ici augmentée de 10+1

ex04e_heritage.dpr

Ce mécanisme tire parti de toutes les fonctionnalités de l'héritage.

1.4 Polymorphisme

1.4.1 Présentation

Rappel : En POO la communication entre objets s'effectue au moyen de **messages**. Un message est l'appel d'une **méthode publique** de l'objet récepteur.

Dans notre exemple la méthode Visualiser devrait être **polymorphe**, c.a.d. s'adapter à l'objet auquel elle est associée. Le **polymorphisme** permet d'adresser un objet indépendamment de son type. Il est ainsi possible d'envoyer un message à un objet, cet objet va traiter différemment ce message suivant son type. Ceci est tout particulièrement utile pour manipules plusieurs objets de type différents mais en relation d'héritage. Ce mécanisme est mis en oeuvre de la manière suivante :
Définir la méthode ançêtre comme étant **virtuelle**, en incluant la directive **virtual** dans la déclaration. Une méthode virtuelle, à la différence d'une méthode statique, peut être surchargée dans les classes dérivées. Quand une méthode surchargée est appelée, c'est le type réel (à l'exécution) du type de classe ou d'objet utilisé dans l'appel de la méthode, et non pas le type déclaré de la variable, qui détermine l'implémentation activée. S**urcharger** les méthodes de même nom dans les objets descendants avec la directive **override**. Une déclaration **override** doit correspondre à la déclaration de l'ancêtre dans l'ordre et le type des paramètres, ainsi que dans le type éventuel du résultat.

1.4.2 Mise en œuvre

L'exemple ci-après reprend les classes TVelo et TVeloMoteur précédemment définies. Pour rendre polymorphe la méthode Avancer, il faut reprendre la déclaration des classes TVelo et TVeloMoteur. Ajouter à la déclaration de la méthode Avancer de la classe TVelo le mot réserve **virtual**, et pour la méthode Avancer de la classe TVeloMoteur **override**. Ainsi le résultat de l'éxécution du programme produira le résultat suivant :

```
TVelo=1
TVeloMoteur=10
```

```
. . .
  TVelo = class
    position: integer;
    procedure avancer; virtual ;
  end;

  TVeloMoteur = class(TVelo)
    Reservoir : integer ;
    procedure avancer; override ;
  end;
```

```
procedure TVelo.avancer;
begin
  position := position + 1;
end;

procedure TVeloMoteur.avancer;
begin
  position := position + 10 ;
  reservoir := reservoir - 1 ;
end;

var
  mesDeuxRoues  : array [0..1] of TVelo;
  i1 : Integer;
begin
  mesDeuxRoues[0] := TVelo.Create ;
  mesDeuxRoues[0].position := 0 ;
  mesDeuxRoues[1] := TVeloMoteur.Create ;
  mesDeuxRoues[1].position := 0 ;
  for i1 := low(mesDeuxRoues) to high(mesDeuxRoues) do
  begin
    mesDeuxRoues[i1].avancer;
    writeln(mesDeuxRoues[i1].ClassName + '=' +
      intToStr(mesDeuxRoues[i1].position))
  end;
  for i1 := low(mesDeuxRoues) to high(mesDeuxRoues) do
    mesDeuxRoues[i1].free;
  readln;
end.
```

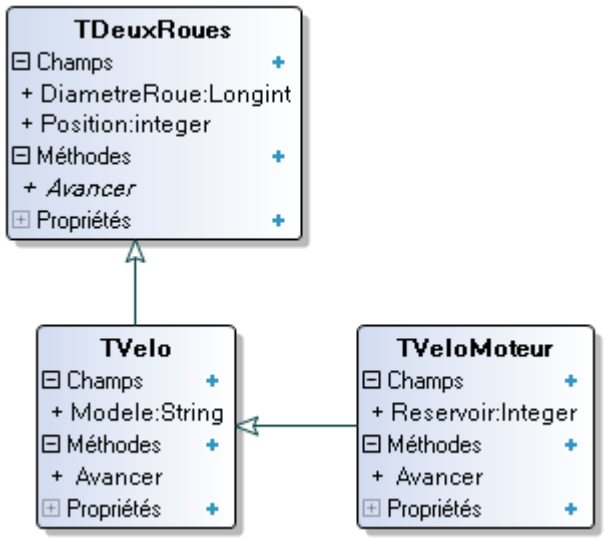
ex04f_polymorphe.dpr

Redéclarer une procédure qui existait déjà dans une classe ancêtre sans utiliser les directives **virtual** et **override**, masque la procédure ancêtre. Les adresses des procédures sont statiques et tout est résolu lors de la compilation. Tant que l'on utilise une variable du type exact de la classe, on n'a aucun problème. La procédure appelée est bien celle que l'on a redéfinie, mais dès que l'on fait un appel à un objet au moyen d'une variable d'un type ancêtre compatible (ce qui est le cas dans l'exemple ou l'on fait appel à un objet TveloMoteur stocké dans un tableau de TVelo grâce au pointeur sur le tableau), la procédure appelée est celle qui correspond à la classe de ce pointeur et les déclarations dans des classes dérivées sont ignorées.
Au contraire, lorsqu'une fonction est déclarée comme virtuelle, les informations permettant son appel sont générées dynamiquement à l'exécution lors de l'instanciation de l'objet, en fonction de son type garantissant l'appel de la bonne procédure, quel que soit le transtypage utilisé.

1.5 Virtualisation et abstraction

Reprise de notre exemple précédent avec la pleine utilisation du polymorphisme par l'utilisation de la directive **abstract** accolée à la directive **virtual**. Nous arrivons donc ici à un **concept abstrait** de deuxRoues qui pourra se décliner soit en vélo, soit en vélomoteur le moment venu.

CODE	DESCRIPTION

...
type
  TDeuxRoues = Class
  DiametreRoue : Longint ;
  Position : integer ;
  Procedure Avancer ; **virtual ; abstract ;**
  end ;
 	
  TVelo = Class (TDeuxRoues)
  Modele : String ;
  Procedure Avancer ; override ;
  end ;
 	
  TVeloMoteur = Class (TVelo)
  Reservoir : Integer ;
  Procedure Avancer ; override ;
  End;
 	
…./…
 	
  procedure TVelo.Avancer;
  begin
  Position := Position + 1 ;
  end;
 	
  procedure TVeloMoteur.Avancer;
  begin
  Position := Position + 10 ;
  end;
 	
var
  MonDeuxRoues : TDeuxRoues ;
 	
procedure FaireQuelqueChose ;
begin
  MonDeuxRoues.Position := 0 ;
  MonDeuxRoues.Avancer ;
  writeln (IntToStr(MonDeuxRoues.Position)) ;
end;
 	
begin
  MonDeuxRoues := TVeloMoteur.Create ;
  FaireQuelqueChose ;
  MonDeuxRoues.Free ;
  MonDeuxRoues := TVelo.Create ;
  FaireQuelqueChose ;
  MonDeuxRoues.Free ;
  readln ;
end.
	Définition du type ancêtre. Nous avons ici la description du concept de **deux roues**. Un ancêtre de ce type pourrait être le concept de **véhicule**.
	virtual ; abstract ; indique une procédure ou une fonction qui ne sera implémentée que dans les descendants.
	
	Diagramme UML généré avec DELPHI.
	 
	
	Cette procédure regroupe la partie commune à l'utilisation soit du vélo soit du vélomoteur.
	
	Pour utiliser un objet, on instancie une variable du type Objet.

ex04g_abstract.dpr

Lors de l'écriture de la description de la structure Maj+Ctrl+C permet de compléter la classe au niveau du curseur c.a.d. créer les éléments nécessaires dans la partie implémentation.

1.5.1 Exploration plus approfondie

L'exercice suivant utilise l'unité **uVisuOctets.pas** décrite plus haut et sert de point de départ pour visualiser la structure interne sous-jacente aux exemples de ce chapître.

```
. . .
uses
  System.SysUtils,
  uVisuOctets in 'uVisuOctets.pas';

const
  KMarque: array [0 .. 2] of AnsiString = ('Algue d"or', 'Idéal Sport',
    'Jacques Anquetil');
  KDiametre: array [0 .. 3] of integer = (712, 664, 640, 600);
type
  TVelo = record
    DiametreRoue: Longint;
    Modele: String[16];
    Position: integer;
  end;
var
  MesVelos: array of TVelo;
  i1, i2, cpt: integer;
begin
  setlength(MesVelos, length(KMarque) * length(KDiametre));
  cpt := low(MesVelos);
  for i1 := low(KMarque) to high(KMarque) do
    for i2 := low(KDiametre) to high(KDiametre) do
    begin
      MesVelos[cpt].Modele := KMarque[i1];
      MesVelos[cpt].DiametreRoue := KDiametre[i2];
      MesVelos[cpt].Position := $21436587 ;
      cpt := cpt + 1 ;
    end;
  for i1 := low(MesVelos) to high(MesVelos) do
    writeln(MesVelos[i1].Modele + ':' + intToStr(MesVelos[i1].DiametreRoue));
  writeln ;
  writeln (VisuXOctets(MesVelos,128)) ;
  readln;
end.
```

ex04h_MesVelos.dpr

2 Pour aller plus loin

2.1 Protection et visibilité des objets

Le système de protection du contenu des objets dans Delphi est assuré par les mots réservés **public, private, protected, published, strict private et strict protected**. On peut placer dans chacun de ces blocs aussi bien donnés membres, méthodes que propriétés. Ce tableau montre la visibilité des déclarations de ces blocs pour les principaux mots réservés :

Visibilité	Au sein de la classe	Dans une classe dérivée	A partir d'une instance
Private	Oui	Non	Non
Protected	Oui	Oui	Non
Public	Oui	Oui	Oui
Published	Oui	Oui	Oui

published permet de visualiser les propriétés et méthodes d'une classe dérivée de TComponent dans l'inspecteur d'objet.

Outre les spécificateurs de visibilité **private** et **protected**, le compilateur Delphi supporte des paramètres de visibilité supplémentaires avec des contraintes d'accès plus importantes. Il s'agit des visibilités **strict private** et **strict protected** qui empêche toute redéclaration dans les descendants.

Au sein d'une même unité, les déclarations private et protected sont accessibles par les descendants de la même manière que les déclarations public. Elles ne sont réellement cachées que pour les autres unités.
Et dans ce dernier cas, il est quand même possible de changer la visibilité d'une fonction ou d'une procédure dans une classe descendante. voici dans cet exmple un moyen pour accéder aux variables privée d'un ancêtre déclaré dans une autre unité.

Unité décrivant la classe de base

ex05f_accesPrivate.dpr& ex05f_accesPrivate1.pas

CODE	DESCRIPTION

unit ex05f_accesPrivate1;
 	
interface
 	
uses Classes ;
 	
type
TMaClasse = class
private
fv1: integer;
public
property v1: integer read fv1 write fv1;
function fRetourneValeurPlusUn: integer;
end;
 	
implementation
 	
function TMaClasse.fRetourneValeurPlusUn: integer;
begin
result := fv1 + 1;
end;
 	
end.
	Classe ancêtre
	
	Variable private
	
	Retourne le contenu de la variable private **fv1**

Projet principal

```
program ex05f_accesPrivate;
{$APPTYPE CONSOLE}
{$R *.res}

uses
  System.SysUtils,
  ex05f_accesPrivate1 in 'ex05f_accesPrivate1.pas';

type
  TAccesPrive = class(TObject)
  private
    fv1: integer;
  end;

  TMaClasseDescendante = class(TMaClasse)
  public
    procedure IncrementeValeurPlusUn;
  end;

procedure TMaClasseDescendante.IncrementeValeurPlusUn;
begin
  TAccesPrive(TMaClasse(self)).fv1 :=
  TAccesPrive(TMaClasse(self)).fv1 + 1;
end;

var
  aClass1X: TMaClasseDescendante;

begin
  aClass1X := TMaClasseDescendante.Create;
  aClass1X.v1 := 3;
  aClass1X.IncrementeValeurPlusUn;
  writeln(aClass1X.fRetourneValeurPlusUn.ToString);
  aClass1X.Free;
  readln
end.
```

TAccesPrive Classe décrivant les propriétés privées de l'ancêtre TMaClasse devant être rendue accessibles dans la classe descendante TMaClasseDescendante.
Attention la partie private doit eêtre décrite complètement à l'idantique de la partie private de la classe encêtre

Moyen d'accès à la variable privée **fv1** de l'ancêtre

2.2 Champs et propriétés

2.2.1 Champ d'un objet

Comme nous l'avons vu de la paragraphe consacré au type **record** un **champ** n'est finalement qu'une variable associée à une structure. Les champs peuvent être de type quelconque, y compris de type classe (les champs peuvent donc contenir des références d'objet). Décrits à l'intérieur d'une classe, les champs sont généralement privés.

2.2.2 Propriétés d'un objet

Une **propriété** définit un attribut d'un objet. Une **propriété** associe des actions spécifiques à la lecture et la modification de la donnée sous jacente associée. Les propriétés proposent un moyen de contrôler l'accès aux attributs d'un objet et permettent aussi le calcul des attributs.CLe système de propriétés (**Property**) permet d'accéder plus simplement aux données (**field**) d'un objet en lecture et écriture.

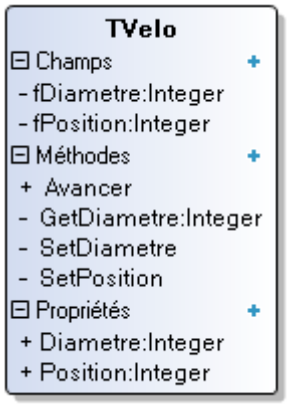
```
type
  TVelo = class
  strict private
    fDiametre : Integer ;
    . . .
    function GetDiametre : Integer;
    procedure SetDiametre(val : Integer);
  public
    . . .
    property Diametre : Integer read GetDiametre write SetDiametre;
    . . .
  end;
```

Read	permet de définir une fonction permettant l'accès indirect en lecture aux données membres de l'objet à travers une propriété (ici accès indirect à la donnée fDiametre avec la propriété Diametre par la fonction GetDiametre).
Write	Permet de définir une fonction permettant l'accès indirect en écriture aux données membres de l'objet (ici accès indirect à la donnée fDiametre avec la propriété Diametre par la procedure SetDiametre).

Le système de propriété est plus riche, car il permet, plutôt que d'accéder directement à la donnée membre, de passer par une fonction qui permettra d'éventuels tests de validité, ou un accès plus facile aux données. Cette technique assure l'intégrité de vos données :

En repartant d'un exemple précédent voici un programme utilisant des propriétés.

CODE	DESCRIPTION

…
type
  TVelo = class
  strict private
  fDiametre : Integer ;
  fPosition : Integer ;
  procedure SetPosition(val : Integer);
  function GetDiametre : Integer;
  procedure SetDiametre(val : Integer);
  public
  property Position : Integer read fPosition write SetPosition;
  property Diametre : Integer read GetDiametre write SetDiametre;
  procedure Avancer;
  end;
 	
procedure TVelo.SetPosition(val: Integer);
begin
  fPosition := abs(val) ;
end;
 	
function TVelo.GetDiametre: Integer;
begin
  result := fDiametre ;
end;
 	
procedure TVelo.SetDiametre(val: integer);
begin
  case val of
  0..619 : fDiametre := 600 ;
  620..661 : fDiametre := 640 ;
  662..687 : fDiametre := 664 ;
  else fDiametre := 712
  end;
end;
…/…
var
  MonVelo : TVelo ;
begin
  MonVelo := TVelo.Create ;
  MonVelo.Diametre := 605 ;
  writeln (MonVelo.Diametre) ;
  MonVelo.Position := -5 ;
  MonVelo.Avancer ;
  writeln (MonVelo.Position) ;
  MonVelo.Free ;
  readln ;
end.
	la fontion SetDiametre permettra par exemple de controler la valeur du diamètre entré parmis celles possibles
	
	Chaque propriété a un spécificateur **read**, un spécificateur **write** ou les deux.
	Dans la classe TVelo, aucune action n'est associée à la lecture de la propriété Position ; l'opération read consiste simplement à obtenir la valeur stockée dans le champ fPosition.
	
	 
	
	Diagramme UML correspondant généré avec DELPHI.
	
	MonVelo.Diametre := 605 ; et
	MonVelo.SetDiametre (605) ; sont équivalents

ex04i_propriete.dpr

2.3 Délégation

Les types de base des mécanisme de réutilisation d'élément d'un programme sont **l'héritage**, **l'agrégation** et la **délégation**

La **délégation** est le système utilisé par Delphi pour passer la main à une procédure en réponse à un événement. Cette technique permet d'éviter de surclasser (hériter) une classe pour surcharger la méthode en réponse à l' événement. En d'autres termes l'objet qui reçoit la requête délègue le traitement de cette demande à une méthode d'un autre objet .

Voici un exemple de délégation

Définition d'un type de procédure, les valeurs transmises sont prévue pour définir l'auteur 'Sender' et autre chose comme ici une valeur de type **Integer**.

```
type
TValeurChangeEvent = Procedure(Sender : TObject; aVal : Integer ) of Object;
```

Description d'une classe qui comporte un élément de type **Integer** un champ "**Fvaleur**" de type entier et sa propriété associée "**Valeur**". C'est la modification **Valeur** via son 'setter' **SetValeur** qui va déclencher la procédure liée à la notification

```
TValeur = class
Private
  FValeur: Integer;
  FValeurChange: TValeurChangeEvent;
  Procedure SetValeur(aValeur: Integer);
Public
  Constructor Create;
Published
  Property Valeur: Integer read FValeur write SetValeur;
  Property OnValeurChange: TValeurChangeEvent read FValeurChange
    write FValeurChange;
end;
```

Chargement du champ **fValeur** et appel de la procédure qui a été assignée ailleurs (voir FormActivate ci après) et délégation du travail à réaliser

```
procedure TValeur.SetValeur(aValeur: Integer);
begin
  if Self.FValeur <> aValeur then
  begin
    Self.FValeur := aValeur;
    if Assigned(FValeurChange) then
      FValeurChange(Self, FValeur);
  end;
end;
```

Dans cet exemple la procédure **AfficheValeur** liée à la classe Tform1 reprend les paramètres du modèle **TvaleurChangeEvent**.

```
TForm1 = class(TForm)
  ListBox1: TListBox;
  procedure FormActivate(Sender: TObject);
public
  procedure AfficheValeur(Sender: TObject; aValeur: Integer);
end;
```

Exemple de procédure

```
procedure TForm1.AfficheValeur(Sender: TObject; aValeur: Integer);
begin
  ListBox1.Items.Add(Sender.ClassName);
  ListBox1.Items.Add(aValeur.ToString);
end;
```

Création d'une instance de **TValeur** , affectation de la méthode **OnValeurChange**

```
procedure TForm1.FormActivate(Sender: TObject);
var
  aValeur: TValeur;
begin
  aValeur := TValeur.Create;
  aValeur.FValeurChange := Self.AfficheValeur;
  aValeur.Valeur := 3 ;
  aValeur.Free
end;
```

Voici un autre exemple du principe de délégation. Lorsque l'on veut placer du code en réponse à l'evènement déclenché lorsqu'on relève le bouton de la souris (**mouseUp**) sur un composant visuel d'une fiche. On place ce code dans "l'événement OnMouseUp". En réalité ce qui est fait par Delphi, c'est une nouvelle méthode qui est créée dans la classe dérivée de TForm, et la donnée membre en réponse à l'événement OnClick pointe sur cette nouvelle méthode :

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

La délégation permet aussi de partager le code d'une même méthode en réponse à différents évènements de différents composants. Ainsi le code ButtonClick peut être réutilisé par différents autres composants Tbutton en réponse à l'événement OnClick comme notre exemple sur la calculatrice.

```
property OnClick: TNotifyEvent read FOnClick write FOnClick stored IsOnClickStored;
```

Cette définition se trouve dans la description de l'objet **TControl** de l'unité **Vcl.Controls**. Le type **TNotifyEvent = procedure(Sender: TObject) of object** de l'unité **System.Classes** définit le paramètre qui doit être passé à la procédure associée à l'événement. La directive **stored** est un spécificateur de stockage. Elle n'a aucun effet sur l'exécution du programme mais contrôle la manière dont Delphi gère les informations de type à l'exécution (RTTI). Plus précisément, un spécificateur de stockage spécifie si Delphi enregistre la valeur des propriétés publiées dans les fichiers (.DFM). La directive **stored** doit être suivie par

True, False, le nom d'un champ Boolean ou par le nom d'une méthode sans paramètre qui renvoie une valeur de type Boolean.

2.4 Pointeurs déréférencés

Le symbole **^** est appelé **opérateur de déréférencement**, c'est-à-dire qu'il renvoie la valeur stockée à l'adresse mémoire de la variable dont le nom est donné avant le symbole. Ex. v1^ signifie : à l'adresse de v1

Le compilateur déréférence automatiquement lorsque cela est judicieux un objet qui est adressé sans utiliser '^'. Ce mécanisme est fourni de base par Delphi lorsque vous utilisez des objets qui sont déclarés avec le mot réservé **class**. Ce mécanisme permet d'utiliser les pointeurs de façon transparente.

En fait, dans Delphi, une Classe instanciée un pointeur sur la structure décrite dans la classe. Le fait de déclarer une variable de type Class déclare en fait un pointeur. Ensuite la méthode Create alloue l'espace mémoire nécessaire à l'objet créé et stocke l'adresse de cet espace dans ce pointeur.

2.5 RTTI

Les RTTI sont les **Run Time Type Information**, c'est-à-dire les informations de type à l'exécution. On peut ainsi tester à l'exécution le type d'un objet (grâce à l'opérateur **is**). Ces informations de type sont très puissantes dans Delphi, au point de pouvoir créer une nouvelle instance d'un objet à partir de ses informations de type. Le mot réservé **as** permet de faire un transtypage dynamique (dynamic cast)

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   if Sender is TButton then
      (Sender as TButton).Caption := 'OK' ;
end;
```

Sender is TButton	Permet de tester si Sender est de type TButton ou d'un type compatible.

2.6 Intercepter les messages Windows

Rien de plus simple avec Delphi VCL. Qu'il s'agisse d'un message utilisateur ou bien d'un message Windows, vous utiliserez le mot réservé message en spécifiant ensuite la constante de message désirée. Le reste ne pose aucune difficulté.

CODE	DESCRIPTION

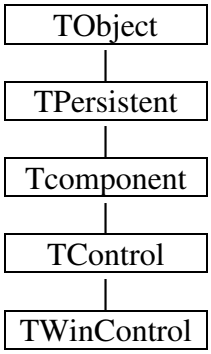
interface
 	
type
  TFigure = class(TControl)
  protected
  procedure WMTimer(var Msg : Tmessage); message WM_TIMER;
  end;
 	
implementation
 	
procedure TFigure.WMTimer(var Msg : Tmessage);
begin
  inherited;
end;
	Message - Suivi d'une constante, indique le type de message à intercepter.

Il est possible d'appeler la méthode de la classe ancêtre juste avec inherited. Les paramètres sont passés automatiquement. Les paramètres du message peuvent être décomposés automatiquement en utilisant les types de messages structurés de Delphi, tel TWMKeyDown, définis dans l'unité **Winapi.Messages**.

Il est possible de générer ses propres message Windows.
Voir le chapitre "Exemple VCL d'un système de recherche de chaînes de caractères" pour une description plus complète.

2.7 Arborescence des classes dans Delphi.

Voici les premiers niveaux pour la VCL:



2.8 Interfaces

2.8.1 Rappel

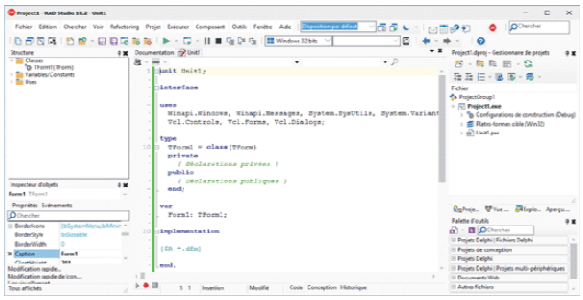
Le terme interface est utilisé au moins à trois endroits : L'interface utilisateur, le mot interface qui va définir une zone de l'unité et l'interface d'un objet dont nous allons parler.

INTERFACES

INTERFACE dans une unité

```
Unit Unit1 ;
Interface
Uses
        Windows, Messages, SysUtils, Classes, Graphics ;
Type
        Tform1=class(Tform)
        Public
        Procedure PasserVitesse (vitesse : integer) ;
        End ;
Var
        Form1 : Tform1;

Implementation
Procedure TForm1.PasserVitesse (vitesse : integer) ;
...
End.
```

INTERFACE UTILISATEUR



Object Pascal est un langage dit de 4ème génération (LAG). Dans une unité (unit) la section interface commence par le mot réservé **interface** et se poursuit jusqu'à la section **implémentation**. La section **interface** déclare les constantes, types, variables, procédures et fonctions accessibles aux autres entités (unités ou programmes) qui utilisent l'unité. ces entités sont dite publiques car une entité peut y accéder comme si elles étaient déclarées dans l'entité elle-même. La déclaration dans l''interface d'une procédure ou d'une fonction ne contient que l'en-tête. Le bloc de la procédure ou de la fonction se trouve dans la section implémentation.

2.8.2 Interfaces de base

Une interface énumère la liste de toutes les procédures et fonctions qui doivent être décrites et implémentées dans une classe qui utilise cette interface.
Le mot clé **interface** a été introduit spécialement pour Delphi. Il est possible aussi d'utiliser une classe abstraite (Class Abstract). Mais dans une interface les méthodes sont obligatoirement publiques et abstraites et il est posssible d'associer un identificateur unique (GUID).
Elle permet de décrire ce qu'il y à faire et non comment le faire (pas d'implémentation).
Ici le mot **Interface** est utilisé pour la classe IInterface qui est la type de base pour toutes les interfaces définies en code Delphi. Voic comment elle est décrite dans l'unité system

system.pas

```
...
IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
...
```

Cet exemple décrit une première approche de la Notion d'interface

ex06a_iinterface.dpr

...
type
  IVehicule = interface(IInterface)
  Procedure Avance;
  End;
 	
  TVelo = Class(TObject, IVehicule)
  position: Integer;
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
  Procedure Avance;
  end;
 	
procedure TVelo.Avance;
begin
  position := position + 1;
end;
 	
function TVelo.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
  Result := S_OK
  else
  Result := E_NOINTERFACE;
end;
 	
function TVelo._AddRef: Integer;
begin
  Result := -1;
end;
 	
function TVelo._Release: Integer;
begin
  Result := -1;
end;
 	
var
  MonIVehicule : IVehicule;
  MonVelo: TVelo;
begin
  MonVelo := TVelo.Create;
  MonVelo.Demarre;
  MonVelo.Avance;
  writeln('Position := ' + MonVelo.position.ToString);
  If assigned(MonIVehicule) then
  MonIVehicule.Avance;
  writeln('Position := ' + MonVelo.position.ToString);
  MonIVehicule := MonVelo;
  If assigned(MonIVehicule) then
  MonIVehicule.Avance;
  writeln('Position := ' + MonVelo.position.ToString);
  MonIVehicule := Nil;
  MonVelo.Free;
  readln;
	pas d'implémentation dans une interface ! – les fonctions QueryInterface, _ AddRef, _Release doivent être implémentées
	
	Plusieurs interfaces peuvent être décrites à la suite
	
	functions de IInterface IVehicule dérites dans system.pas à redéclarer pour les implémenter.
	Note : si les modules du projet sont écrits dans des langages différents alors funtion et procedure doivent être déclarées avec la convention **stdcall**
	
	QueryInterface renvoie une référence à l'interface, spécifiée par le paramètre IID, sous le paramètre Obj. Si la fiche ne gère pas l'interface, le paramètre Obj a la valeur nil
	
	compteur de référence non géré
	
	compteur de référence non géré
	
	La position n'a pas bougé. MonIVehicule n'est pas assigné
	Assigne l'interface (Déclenche la fonction _AddRef)
	MonIVehicule est maintenant assigné et associé à l'instance MonVelo
	
	Déclenche la fonction _Release

La déclaration d'une interface est souvent assimilée à un contrat. Si une interface est mentionnés dans une classe alors toutes les méthodes décrites dans cette interface et dans ses ancêtres doivent être décrites et implémentées dans la classe.

2.8.3 Classes dérivées de TInterfacedObject

Le compilateur Delphi vous fournit l'essentiel de la gestion mémoire IInterface grâce à son implémentation de l'interrogation et du comptage de références de l'interface. Par conséquent, pour un objet qui est créé et détruit via ses interfaces, le comptage de références dérivant de TinterfacedObject peut être utilisé. Si vous choisissez d'utiliser le comptage de références, vous devez faire attention à ne manipuler l'objet que sous la forme d'une référence d'interface et à être cohérent dans votre comptage de références. Par ailleurs, les compilateurs mobiles Delphi supportent le comptage automatique des références pour les classes. Pour de plus amples informations, voir Comptage automatique des références dans les compilateurs mobiles Delphi.

Dans cet exemple Tvelo est hérité de TinterfacedObject. les fonctions de comptage de référence sont déjà intégrées

ex06b_iinterface.dpr

```
...
 TVelo = Class(TInterfacedObject, IVehicule)
    position: Integer;
    Procedure Avance;
  end;

procedure TVelo.Avance;
begin
  position := position + 1;
end;

Procedure TVelo.Demarre;
begin
  position := 0
end;

var
  MonIVehicule: IVehicule;
  MonVelo: TVelo;

begin
  MonVelo := TVelo.Create;
  MonVelo.Demarre;
  MonVelo.Avance;
  MonIVehicule := MonVelo;
  MonIVehicule.Avance;
  writeln ('Position := '+MonVelo.position.ToString) ;
  MonIVehicule := Nil;

  readln;
end.
end.
```

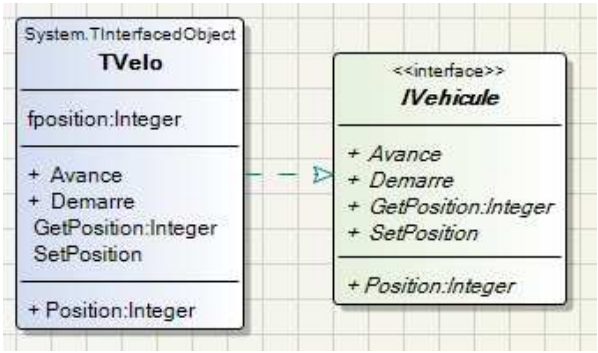
Association instance - interface

Entraîne le destruction de l'instance MonVelo
Donc pas de MonVelo.free !

En tenant compte de ces informations notre exemple est modifié comme suit. Nous introduisons ici la propriété "Position" en lieu et place du field "Position : integer"

ex06c_iinterface.dpr

```
...
type
  IVehicule = interface
    procedure SetPosition(val: Integer);
    function GetPosition(): Integer;
    property Position: Integer read GetPosition write SetPosition;
    Procedure Demarre;
    Procedure Avance;
  End;

  TVelo = Class(TInterfacedObject, IVehicule)
  private
    fposition: Integer;
    procedure SetPosition(val: Integer);
    function GetPosition(): Integer;
  public
    property Position: Integer read GetPosition write SetPosition;
    Procedure Demarre;
    Procedure Avance;
  end;

procedure TVelo.Avance;
begin
  fposition := fposition + 1;
end;

Procedure TVelo.Demarre;
begin
  fposition := 0
end;

function TVelo.GetPosition: Integer;
begin
  result := fposition ;
end;

procedure TVelo.SetPosition(val: Integer);
begin
  fposition := val ;
end;

var
  MonIVehicule: IVehicule;

begin
  MonIVehicule := TVelo.Create;
  MonIVehicule.Demarre;
  MonIVehicule.Avance;
  writeln ('Position := '+MonIVehicule.position.ToString) ;
  MonIVehicule := Nil;
  readln;
end.
```



2.9 Methodes anonymes

En Object Pascal une **méthode anonyme** est en réalité ce que l'on appelle dans d'autres langages informatiques une **fermeture** (closure). Une **fermeture** est donc créée lorsqu'une fonction est définie dans le corps d'une autre fonction et fait référence à des arguments ou des variables locales à la fonction dans laquelle elle est définie.
Dans la clause **type** une description incluant "**reference to function**" permet de déclarer une valeur de retour de type **funcion** à une fonction. Un usage, hors **méthode anonyme**, en est présenté dans l'unité **System.Classes.** Voir **TListSortCompareFunc = reference to function …**

CODE	DESCRIPTION
. . . type TRefFonc1 = reference to function(x: Integer): string;	Définit une équivalence entre TRefFonc1 et la référence à une fonction avec un ou des paramètres et une valeur de retour particuliers.

function Ajouter(y: Integer): TRefFonc1;
begin
Writeln ('Y = '+intToStr(y));
Result := function(x: Integer) : String
begin
Writeln ('X = '+intToStr(x));
Result := 'X + Y = '+intToStr(x + y);
end;
end;
	Définit la fonction **ajouter** avec le type **TRefFonc1** comme valeur de retour
	
	Result de la fonction **Ajouter -** Définit un fonction sans nom (anonyme) avec les mêmes paramètre et une valeur de retour que ceux décrit dans le type **TRefFonc1**
	Result de la fonction anonyme

var
v1 : TRefFonc1 ;
begin
v1 := Ajouter(3);
Writeln (v1(5));
Writeln (v1(7));
readln ;
end.. . .
	v1 est initialisée en même temsp que **y**
	l'appel de la fonction reprend la valeur de **y** et lui ajoute la valeur de **x**
	
	Résultat produit :
	
	Y = 3
	X = 5
	X + Y = 8
	X = 7
	X + Y = 10

ex05a_closure.dpr

Dans cet exemple Y joue le rôle de variable persistente

Quelques infos complémentaires sur internet avec les thèmes suivants :
Closures Sussman Steele 1975 - Scheme programming language

Note : Les descriptions et l'exemple ci-dessus utilisent des **fonctions**. Le principe est le même avec des **procédures**.

2.10 Class function, class property et autres

Au même titre qu'une procédure classique décrite dans un programme une Class procédure (ou tout autre Class élément) est chargée au lancement du programme. Elle est donc accessible directement une fois le programme lancé. On emploie le terme d'élément statique. L'exemple ci après montre le positionnement d'une procédure et d'une classe procédure. L'avantage d'une Class procédure est lié à son encapsulation au sein d'une structure comme on peut en trouver au sein des unités standards de Delphi comme System.IOUtils (TDirectory, TPath, …). Elle peut être assimilée à un **singleton**.

CODE	DESCRIPTION

. . .
type
  TMyObject = record
  public
  class procedure proc1 ; static ;
  end;
 	
procedure proc2 ;
begin
  writeln ('P2') ;
end;
 	
function adresse(aPt: pointer): string;
begin
  result := '$' + intToHex(integer(pointer(aPt)), 8);
end;
 	
class procedure TMyObject.proc1;
begin
  writeln ('P1') ;
end;
 	
begin
  writeln('Proc1 ' + adresse(@TMyObject.proc1));
  writeln('Proc2 ' + adresse(@proc2));
  TMyObject.proc1 ;
  proc2 ;
  readln ;
end.
	Proc1 $0041AE90
	Proc2 $0041ADEC
	P1
	P2

ex05b1_classFunc.dpr

Il est préférable d'utiliser une structure de type **record** plutôt qu'un type **class** qui peut prêter à confusion. En effet même si les Class éléments peuvent cohabiter avec d'autres élément dans la description d'une classe, les "Class éléments" ne peuvent pas communiquer avec les autres éléments non "Class élément" au sein de l'implémentation des classes. **Donc de préférence banir ce type de structure**

2.11 Class operator – surcharge d'opérateur

A propos de la surcharge des opérateurs il est possible de surcharger certaines fonctions, ou "opérateurs", dans les déclarations d'enregistrement. Le nom de la fonction opérateur correspond à une représentation symbolique dans le code source. Par exemple, l'opérateur Add correspond au symbole **+**.

CODE	DESCRIPTION

…
uses
  Windows, SysUtils, Classes;
 	
type
  TUnPoint2D = record
    X: Integer;
    Y: Integer;
  public
    class operator Add(const P1, P2: TUnPoint2D): TUnPoint2D;
    class operator Subtract(const P1, P2: TUnPoint2D): TUnPoint2D;
  end;
 	
class operator TUnPoint2D.Add(const P1, P2: TUnPoint2D): TUnPoint2D;
begin
  Result.X := P1.X + P2.X;
  Result.Y := P1.Y + P2.Y;
end;
 	
class operator TUnPoint2D.Subtract(const P1, P2: TUnPoint2D): TUnPoint2D;
begin
  Result.X := P1.X - P2.X;
  Result.Y := P1.Y - P2.Y;
end;
 	
var
  P1, P2, P3: TUnPoint2D;
 	
begin
  P1.X := 4;
  P1.Y := 5;
  P2.X := 2;
  P2.Y := 1;
  P3 := P1 + P2;
  Writeln(format('P1 (%d,%d) + P2 (%d,%d) = P3 (%d,%d)', [P1.X, P1.Y, P2.X,
    P2.Y, p3.X, p3.Y]));
  P3 := P1 - P2;
  Writeln(format('P1 (%d,%d) + P2 (%d,%d) = P3 (%d,%d)', [P1.X, P1.Y, P2.X,
    P2.Y, p3.X, p3.Y]));
  readln;
end.
	Cet exemple décrit une structure d'enregistrement de deux entiers avec les fonctions d'addition et de soustraction appliqués à cette structure.
	
	Utilisation de l'opérateur **+**
	
	Utilisation de l'opérateur **-**
	
	Résultat :
	P1 (4,5) + P2 (2,1) = P3 (6,6)
	P1 (4,5) + P2 (2,1) = P3 (2,4)

ex05c_classOperator.dpr

2.12 Record helper, class helper

Une assistance de classe ou d'enregistrement est un type qui, lorsqu'il est associé à une autre classe ou un autre enregistrement, introduit des méthodes et propriétés supplémentaires. Ce mécanisme permet d'étendre une classe sans avoir recours à l'héritage, ce qui est aussi utile pour les enregistrements qui n'acceptent pas du tout l'héritage. Les assistances de classes et d'enregistrements permettent d'étendre un type, mais elles ne doivent pas être considérées comme un outil de conception à utiliser lors du développement d'un nouveau code. Pour du nouveau code, vous devez toujours vous baser sur l'héritage de classe normale et les implémentations d'interfaces.
Remarque : Ce mécanisme est incompatible avec la surcharge d'opérateurs.

CODE	DESCRIPTION

. . .
uses
Windows, SysUtils, Classes;
 	
type
TIntegerHelper = record helper for
Integer
function EnTexte: string;
end;
 	
{ TIntegerHelper }
 	
function TIntegerHelper.EnTexte: string;
begin
Result := IntToStr (self);
end;
 	
var
i1 : Integer ;
begin
writeln ($12345.EnTexte) ;
i1 := 'Bonjour'.length ;
writeln ('Bonjour'.Substring (3)) ;
writeln (i1.EnTexte) ;
readln;
end.
	Description d'un record helper pour le type entier avec une fonction permettant de retourner un nombre sous forme d'un texte.
	
	Length, substring sont eux même décrits de cette manière dans l'unité sysutils
	La fonction Entexte est applicable aussi à une constante à condition que le compilateur la considère au minimum comme un entier.
	1.Entexte par exemple produira une erreur
	
	Résultat :
	74565 (la valeur passée en paramètre est hexadécimale)
	
	Jour - (Indice du 1er caractère = 0 !)
	7

ex05d_RecordHelper.dpr

Remarque sur **System.SysUtils.TstringHelper** - Les compilateurs mobiles Delphi (DCCIOS32 et DCCIOSARM) introduisent l'indexation basée sur 0 pour les chaînes alors que les compilateurs pour MS-Windows offrent toujours le support des chaînes à base 1. L'utilisation du **TstringHelper** va permettre de traiter cette problématique. Ce qui explique le résultat de **Substring (3)** ci-dessus.

Voici un exemple appliqué à une classe. Un class helper peut être un moyen d'accéder à un champ privé d'une classe.

CODE	DESCRIPTION

…
uses
  Windows, SysUtils, Classes;
 	
type
  TClass1 = class
  i1 : Integer ;
  procedure Proc1;
  function Func1: Integer;
  end;
 	
  TClass1Helper = class helper for TClass1
  procedure MessageAccueil;
  function Func1: Integer;
  end;
 	
function TClass1.Func1: Integer;
begin
  Result := i1+1 ;
end;
 	
procedure TClass1.Proc1;
begin
  i1 := Func1;
end;
 	
procedure TClass1Helper.MessageAccueil;
begin
  Writeln ('Bonjour') ;
end;
 	
function TClass1Helper.Func1: Integer;
begin
  Result := i1+5 ;
end;
 	
var
  O1: TClass1;
 	
begin
  O1 := TClass1.Create;
  O1.MessageAccueil;
  O1.i1 := 5 ;
  O1.Proc1;
  Writeln (O1.i1.ToString) ;
  readln;
end.
	Description d'un class helper pour le type TClass1.
	
	Redefini Func1 pour TCLass1. Notez que la fonction d'assistance de classe Func1 est appelée, car l'assistance de classe est prioritaire sur le type de classe réel.
	
	Cette fonction Func1 va surcharger la précédente. 5 sera ajouté à i1 et non plus 1
	
	Résultat :
	
	Bonjour
	
	10

ex05e_classHelper.dpr

3 Conventions d'appel des procédures et fonctions

Extrait à partir du document accessible à cette adresse https://beta.hackndo.com/conventions-d-appel/

3.1 Conventions d'appel dans le monde Intel x86

Il existe différentes conventions pour les appels de fonctions. Pour une partie la fonction appelante nettoyant la pile après l'appel d'une fonction (CDECL, FASTCALL) ce qui n'est pas le cas par exemple de fonction avec la convention STDCALL. Il y a également différentes manières de passer des arguments à une fonction, par la pile ou par les registres. Dans ce qui suit deux parties Une première partie historique qui énumère les principales conventions d'appel pour les architectures 32 bits, puis une deuxième qui présente la convention par défaut sur les architectures 64 bits.

3.2 Univers 32 bits

Dans les architectures 32 bits, il y a quelques registres disponibles, et quelques conventions d'appel existantes. Il y a un prologue et un épilogue dans chaque fonction pour gérer la construction et la destruction de la pile nécessaire au bon fonctionnement de cette fonction.Voici les principales.

- **CDECL** (Standard C Calling Convention) - Cette convention est celle utilisée par défaut par la plupart des compilateurs C/C++. Ici, c'est la fonction appelante qui nettoie la pile pour la fonction qu'elle va appeler.
- **STDCALL** (Standard Call) - Les arguments sont poussés sur la pile avant l'appel de la fonction, et comme la fonction appelée est normalement responsable du nettoyage de la pile, une fois celle-ci est terminée lorsque le programme reprend son exécution après l'appel de la fonction, la pile semble ne pas avoir été modifiée.
- **FASTCALL** (Fast Calling Convention) - Dans cette convention d'appel, les arguments ne sont plus poussés sur la pile, mais enregistrés dans des registres. Cependant, dans une architecture 32 bits, seuls 2 registres sont utilisables pour cette convention. S'il y a plus de deux arguments à; passer à; la fonction, les suivants seront poussés sur la pile, comme dans les deux conventions d'appel vues précédemment. *Cette convention est théoriquement plus rapide à; l'exécution car elle réduit le nombre de cycles d'instructions*.

3.3 Univers 64 bits

Suite à l'arrivée de 8 nouveaux registres dans les architectures 64 bits (r8 à r15), c'est la convention FASTCALL qui est devenue celle par défaut. Elle permet alors de passer 4 arguments via les registres, au lieu de 2 comme pour les architectures 32 bits. S'il y a plus de 4 arguments, les suivants sont poussés sur la pile.

3.4 Résumé

Convention	Mode	Transmission des arguments	Responsabilité maintenance pile
CDECL	32Bits	Poussés sur la pile	Fonction appelante
STDCALL	32Bits	Poussés sur la pile	Fonction appelée
FASTCALL	32Bits	2 registres puis sur la pile	Fonction appelée
FASTCALL	64Bits	4 registres puis sur la pile	Fonction appelée

Table des matières