POO A GOGO : LES PROPRIETES

Objectifs : dans ce chapitre, vous allez apprendre l'essentiel à propos des propriétés.

Sommaire : Définition – *Getter* et *setter* – Propriétés et variables – Redéfinition d'une propriété – Propriétés par défaut – Les informations de stockage – Les propriétés indexées – Les tableaux de propriétés – Les propriétés de classe – Exemple

Ressources : les programmes de test sont présents dans le sous-répertoire *poo* du répertoire e*xemples*.

QU'EST-CE QU'UNE PROPRIETE ?

Une *propriété* définit l'attribut d'un objet et est avant tout un moyen d'accéder de manière contrôlée à un champ. Si une propriété a l'apparence d'une variable, elle n'en est pas une dans la mesure où elle n'occupe pas forcément de mémoire et qu'aussi bien l'affectation d'une valeur à une propriété que la lecture de sa valeur peuvent déclencher l'exécution d'une méthode. Par ailleurs, membre d'un descendant de ***TComponent*** et insérée dans une section *published*, une propriété deviendra visible dans l'inspecteur d'objet de **Lazarus**.

TRAVAILLER AVEC LES PROPRIETES

LECTURE ET ECRITURE D'UNE PROPRIETE : *GETTER* ET *SETTER*

Il est toujours possible de rendre *public* un champ quelconque. Ainsi la définition d'une classe comme celle-ci est tout à fait correcte :

```
type
  TMyClass = class
  public
    fMyField : string;
  end;
```

L'utilisateur pourra alors affecter une chaîne au champ *fMyField* comme il agirait avec n'importe quelle variable. En supposant que *MyObject* soit une instance de ***TMyClass***, les écritures suivantes seront correctes :

```
MyObject.fMyField := 'affectation correcte' ;
ShowMessage(MyObject.fMyField ) ;
```

Cependant, il est vivement conseillé d'éviter cet accès direct, car il est contraire à l'esprit de la POO. Comprenez bien qu'il ne s'agit pas simplement de croyance ou de purisme, mais de profiter des avantages d'une encapsulation correcte !

Considérez par exemple le cas où le contenu du champ *fMyField* doive toujours apparaître en majuscules dans votre programme. Comme vous le feriez dans le cadre de la programmation procédurale, il vous faudra remplacer toutes les occurrences de votre champ par une expression du genre :

```
UpperCase(MyObject.fMyField)
```

Vous conviendrez que dans un programme complexe et long, réparti dans de nombreuses unités, les risques d'erreurs seront importants. La réutilisation du code et sa maintenance seront aussi très difficiles et fastidieuses.

Les propriétés sont une réponse à ce genre de problème : une propriété permet de déclencher la méthode souhaitée (dans l'exemple en cours, une mise en majuscules). Une propriété, en plus d'accéder au champ visé, peut en effet effectuer les traitements particuliers nécessaires à l'objet auquel elle appartient. Quant à son invocation, elle restera inchangée dans l'ensemble du programme, même si l'implémentation a été modifiée.

L'interface de la classe devrait au minimum ressembler à ceci :

```
type
  TMyClass = class
  strict private
    fMyField : string;
  public
    property MyField: string read fMyField write fMyField;
  end;
```

Une propriété est introduite par le mot réservé *property* suivi de l'identificateur de la propriété, de son type et d'au moins un des deux mots réservés *read* et *write*, eux-mêmes suivis du nom d'un champ ou d'une méthode d'accès.

Le gain paraît nul à ce niveau puisque l'accès se fait directement grâce au nom d'un champ interne, sinon que ce champ est protégé puisqu'il est devenu inaccessible depuis l'extérieur de l'objet.

Une amélioration décisive consistera à utiliser une méthode de lecture (*getter*) et/ou une méthode d'écriture (*setter*) :

```
type
  TMyClass = class
  strict private
    fMyField : string;
    function GetMyField: string;
    procedure SetMyField(const AValue: string);
  public
    property MyField: string read GetMyField write SetMyField;
```

```
end;
```

À présent, en supposant toujours que *MyObject* soit une instance de ***TMyClass***, les écritures suivantes seront correctes :

```
MyObject.MyField := 'affectation correcte' ;
ShowMessage(MyObject.MyField ) ;
```

Quelques conventions sont utilisées de manière à rendre le code source plus lisible. Bien qu'elles n'aient pas de caractère obligatoire, vous devriez vraiment en tenir compte :

- les champs internes ont leur identificateur précédé de la lettre « f » (ou « F ») pour l'anglais *field* ;
- une méthode *getter* porte un nom au préfixe *Get* ;
- une méthode *setter* porte un nom au préfixe *Set*.

Les définitions des deux méthodes d'accès pourraient être celles-ci :

```
{ TMyClass }

function TMyClass.GetMyField: string;
begin
  Result := fMyField;
end;

procedure TMyClass.SetMyField(const AValue: string);
begin
  fMyField := AValue ;
end;
```

Pour le moment, de telles complications ne sont pas justifiées, mais si vous revenez à votre programme complexe, avec ses nombreuses occurrences du champ *MyField* et ses un peu moins nombreuses unités, la transformation du champ en chaîne en majuscules n'exigera que la modification d'une unique ligne de code :

```
procedure TMyClass.SetMyField(const AValue: string);
begin
  fMyField := UpperCase(AValue) ;
end;
```

La modification se propagera dans tout le code sans effort supplémentaire. En fait, tous les traitements légaux sont permis au sein de ces méthodes d'accès.

Afin de montrer l'efficacité des propriétés, vous allez créer une classe chargée de transformer un entier en chaîne de caractères en tenant compte des règles complexes d'accord en français, en particulier pour 80 et 100 qui prennent un « s » lorsqu'ils ne

sont pas suivis d'un autre ordinal et pour le tiret employé ou non systématiquement (suivant les… écoles !).

[Exemple PO-14]

Voici l'interface de cette classe :

```
type

 { TValue2St }

 TValue2St = class
 strict private
   fValue: Integer;
   fStValue: string;
   fWithDash: Boolean;
   fDash: Char;
   procedure SetWithDash(AValue: Boolean);
   procedure SetValue(const AValue: Integer);
 protected
   function Digit2St(const AValue: Integer): string; virtual;
   function Decade2St(const AValue: Integer; Plural: Boolean = True): string; virtual;
   function Hundred2St(const AValue: Integer; Plural: Boolean = True): string; virtual;
   function Thousand2St(const AValue: Integer): string; virtual;
   function Million2St(const AValue: Integer): string; virtual;
 public
   constructor Create;
   property WithDash: Boolean read fWithDash write SetWithDash;
   property Value: Integer read fValue write SetValue;
   property StValue: string read fStValue;
 end;
```

Cette classe appelle les remarques suivantes :

- la section *strict private* abrite les champs et leurs méthodes d'accès : ils sont donc inaccessibles à l'extérieur de la classe ;
- la section *protected* comprend les méthodes qui transforment un entier en chaîne de caractères : cette section ainsi que l'emploi de *virtual* se justifient par le fait que des classes qui descendraient de ***TValue2St*** auraient probablement à modifier ces méthodes afin d'obtenir d'autres résultats ;
- la section *public* comprend un constructeur qui initialisera des données et trois propriétés : *WithDash* qui déterminera l'emploi systématique ou non du tiret, *Value* qui gèrera la valeur entière de travail, et *StValue* pour la chaîne de retour ;
- les propriétés *WithDash* et *Value* accèdent directement aux champs qui les concernent, mais utilisent une méthode pour les définir : *WithDash* modifiera automatiquement la chaîne si elle est elle-même modifiée tandis que *Value* profitera de sa modification pour construire la chaîne correspondante ;
- la propriété *StValue* est en lecture seule : elle accède directement au champ *fStValue* qui aura été calculé en interne ;

- les méthodes *Decade2St* et *Hundred2St* ont toutes deux un paramètre *Plural* défini à *True* par défaut : ce paramètre précisera s'il faut ajouter un « s » et simplifiera l'appel de la fonction si c'est le cas en économisant un paramètre (*Decade2St*(45) est équivalent à *Decade2St*(45, True)).

Voici l'implémentation de cette classe :

```
implementation

const
  CDigit : array[0..9] of string =('zéro','un','deux','trois','quatre',
                                   'cinq','six','sept','huit','neuf');
  CNum1: array[10..19] of string = ('dix','onze','douze','treize','quatorze',
                    'quinze','seize','dix-sept','dix-huit','dix-neuf');
  CNum2: array[1..7] of string = ('vingt','trente','quarante','cinquante',
                    'soixante','soixante-dix','quatre-vingt');
  CHundred = 'cent';
  CThousand = 'mille';
  CMillion = 'million';

{ TValue2St }

procedure TValue2St.SetWithDash(AValue: Boolean);
// *** tiret obligatoire ou non ***
begin
  if fWithDash = AValue then
    Exit;
  fWithDash := AValue;
  if fWithDash then
    fDash := '-'
  else
    fDash := ' ';
  Value := Value; // force la mise à jour du texte associé au nombre
end;

procedure TValue2St.SetValue(const AValue: Integer);
// *** nouvelle valeur ***
begin
  fValue := AValue;
  fStValue := Million2St(fValue); // transformation de l'entier
end;

function TValue2St.Digit2St(const AValue: Integer): string;
// *** chiffres en lettres ***
begin
  Result := CDigit[AValue];
end;

function TValue2St.Decade2St(const AValue: Integer; Plural: Boolean = True): string;
// *** dizaines en lettres ***
begin
  case AValue of
    0..9: Result := Digit2St(AValue);
    10..19: Result := CNum1[AValue];
    20, 30, 40, 50, 60, 70: Result := CNum2[(AValue div 10) - 1];
    21, 31, 41, 51, 61: Result := CNum2[(AValue div 10) - 1] + '-et-un';
```

```
    22..29: Result := CNum2[1] + '-' + Digit2St(AValue - 20);
    32..39: Result := CNum2[2] + '-' + Digit2St(AValue - 30);
    42..49: Result := CNum2[3] + '-' + Digit2St(AValue - 40);
    52..59: Result := CNum2[4] + '-' + Digit2St(AValue - 50);
    62..69: Result := CNum2[5] + '-' + Digit2St(AValue - 60);
    71: Result := 'soixante-et-onze';
    72..79: Result := CNum2[5] + '-' + CNum1[AValue - 60];
    80: if Plural then // cas de 80 avec ou sans s
          Result := CNum2[7] + 's'
        else
          Result := CNum2[7];
    81..89: Result := CNum2[7] + '-' + Digit2St(AValue - 80);
    90..99: Result := CNum2[7] + '-' + CNum1[AValue - 80];
  end;
end;

function TValue2St.Hundred2St(const AValue: Integer; Plural: Boolean = True): string;
// *** centaines en lettres ***
begin
  if (AValue < 100) then
    Result := Decade2St(AValue, Plural)
  else
  if (AValue = 100) then
    Result := CHundred
  else
  if (AValue < 200) then
    Result := CHundred + fDash + Decade2St(AValue - 100, Plural)
  else
  if ((AValue mod 100) = 0 ) then
    Result := Decade2St(AValue div 100, False) + fDash + CHundred
  else
    Result := Decade2St(AValue div 100, False) + fDash + CHundred + fDash +
      Decade2St(AValue - 100 * (AValue  div 100));
  if Plural and ((AValue mod 100) = 0) and (AValue <> 100) and (AValue <> 0) then
    Result := Result + 's'; // cas de 100 multiplié
end;

function TValue2St.Thousand2St(const AValue: Integer): string;
// *** milliers en lettres ***
begin
  if AValue < 1000 then
    Result := Hundred2St(AValue)
  else
  if AValue = 1000 then
    Result := CThousand + fDash
  else
  if AValue < 2000 then
    Result := CThousand + fDash + Hundred2St(AValue - 1000)
  else
  if (AValue mod 1000) = 0 then
    Result := Hundred2St(AValue div 1000, False) + fDash + CThousand + fDash
  else
    Result := Hundred2St(AValue div 1000, False) + fDash + CThousand + fDash
      + Hundred2St(AValue - 1000 * (AValue div 1000));
end;

function TValue2St.Million2St(const AValue: Integer): string;
```

```
// *** millions en lettres ***
begin
 if AValue= 1000000 then
   Result := CDigit[1] + fDash + CMillion
 else
 if AValue < 1000000 then
   Result := Thousand2St(AValue)
 else
 if AValue < 2000000 then
   Result := CDigit[1] + fDash + CMillion + fDash + Thousand2St(AValue - 1000000)
 else
 if(AValue mod 1000000) = 0 then
   Result := Thousand2St(AValue div 1000000)+ fDash + CMillion + 's'
 else
   Result := Thousand2St(AValue div 1000000)+ fDash + CMillion + 's' + fDash +
    Thousand2St(AValue - 1000000 * (AValue div 1000000));
end;

constructor TValue2St.Create;
// *** création de l'objet ***
begin
 inherited Create; // on hérite
 fDash := ' ';
end;
```

La transformation d'un nombre en chaîne a été décomposée en cinq méthodes travaillant respectivement sur les chiffres, les dizaines, les centaines, les milliers et les millions. Cette décomposition de l'entier à traiter évite les méthodes trop longues et alambiquées : *Million2St* va déléguer le travail de précision à ses consœurs.

L'essentiel est de constater que derrière une simple affectation se cache souvent un ensemble complexe d'instructions :

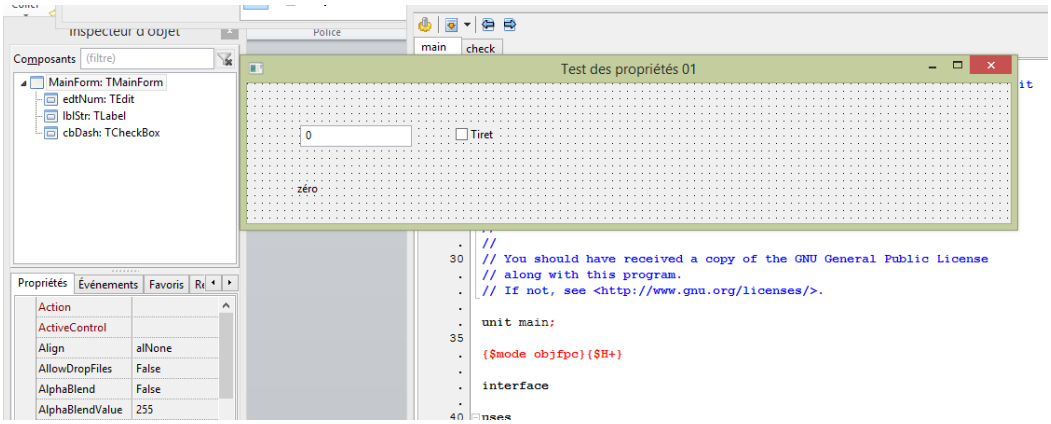
```
MyObject.Value := 123456 ;
```

Apparemment, *Value* de l'objet *MyObject* prend la valeur 123456. En réalité, une série de calculs construira la chaîne « *cent vingt-trois mille quatre cent cinquante-six* ». En passant *WithDash* à *True*, le résultat serait « *cent-vingt-trois-mille-quatre-cent-cinquante-six* » sans aucune autre intervention de l'utilisateur.

Afin de tester votre unité baptisée *check*, procédez comme suit :

- créez une nouvelle application ;
- enregistrez les squelettes créés automatiquement par **Lazarus** sous les noms suivants : *project1.lpi* sous *testproperties01.lpi* et *unit1.pas* sous *main.pas* ;
- ajoutez l'unité *check* à la clause *uses* de l'interface de *main* ;
- changez *Caption* de la fenêtre principale en « *Test des propriétés 01* » ;

- ajoutez un ***TEdit***, un ***TCheckBox*** et un ***TLabel*** à votre fiche principale en les renommant respectivement *edtNum*, *cbDash* et *lblStr* :



L'éditeur prendra la valeur de l'entier à transformer, la case à cocher indiquera si l'emploi des tirets est obligatoire ou non, tandis que l'étiquette contiendra la chaîne calculée.

Continuez votre travail ainsi :

- ajoutez un champ *Value* de type *TValue2St* dans la section *private* de l'interface de la fiche ;
- créez les gestionnaires *OnCreate* et *OnDestroy* de la fiche grâce à l'inspecteur d'objet ;
- faites de même avec les gestionnaires *OnChange* de *edtNum* et *cbDash* ;
- complétez l'unité *main* de cette manière :

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  check; // unité ajoutée

type

  { TMainForm }

  TMainForm = class(TForm)
    cbDash: TCheckBox;
    edtNum: TEdit;
    lblStr: TLabel;
    procedure cbDashChange(Sender: TObject);
    procedure edtNumChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { private declarations }
    Value: TValue2St; // champ de travail
```

```
  public
    { public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.edtNumChange(Sender: TObject);
// *** l'éditeur change ***
var
  Li: Integer;
begin
  if edtNum.Text = '' then // chaîne vide ?
    Exit;
  if TryStrToInt(edtNum.Text, Li) then // nombre entier correct ?
  begin
    Value.Value := Li; // la propriété fait son travail !
    lblStr.Caption:= Value.StValue; // l'étiquette contient la chaîne
  end
  else
    ShowMessage('"'+ edtNum.Text + '" n''est pas un nombre entier correct !');
end;

procedure TMainForm.cbDashChange(Sender: TObject);
// *** avec ou sans tirets ***
begin
  Value.WithDash := cbDash.Checked;
  lblStr.Caption:= Value.StValue; // mise à jour de l'étiquette
end;

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  Value := TValue2St.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
  Value.Free;
end;

end.
```

Vous avez créé une instance de ***TValue2St*** dans la méthode *FormCreate* sans oublier de la libérer dans la méthode *FormDestroy*. Par ailleurs, à chaque fois qu'une modification est apportée à *edtNum*, la validité de l'entrée est vérifiée grâce à *TryStrToInt*, une fonction de la RTL qui essaye de transformer une chaîne en entier : si

cette transformation réussit, l'entier obtenu est affecté à la propriété *Value* de l'instance de ***TValue2St*** avant que la chaîne calculée ne soit affichée dans l'étiquette[1].

[Exemple PO-15]

À présent, un locuteur belge fera remarquer que « *septante* », « *octante* » et « *nonante* » sont des facilités auxquelles il ne voudrait pour rien au monde renoncer. Pour satisfaire ses besoins, il faudrait compléter le tableau *CNum2* :

```
CNum2: array[1..10] of string = ('vingt','trente','quarante','cinquante',
                'soixante','soixante-dix','quatre-vingt','septante', 'octante', 'nonante');
```

Les modifications à apporter à la classe ***TValue2St*** seraient minimes :

```
type
  TValue2StBelg = class(TValue2St)
  protected
    function Decade2St(const AValue: Integer; Plural: Boolean = True): string; override;
  end ;
```

Quant à l'implémentation de la méthode surchargée, elle serait bien plus simple que celle de l'ancêtre :

```
function TValue2StBelg.Decade2St(const AValue: Integer; Plural: Boolean = True): string;
// *** dizaines en lettres – version belge ***
begin
  Result := inherited Decade2St(AValue, Plural); // on hérite de la valeur de l'ancêtre
  case AValue of // on ne modifie que les nouvelles valeurs
    70, 80, 90 : Result := CNum2[(AValue mod 10) + 1] ;
    71, 81, 91 : Result := CNum2[(AValue mod 10) + 1] + 'et-un';
    72..79 : Result := CNum2[8] + '-' + Digit2St(AValue - 70);
    82..89 : Result := CNum2[9] + '-' + Digit2St(AValue - 80);
    92..99 : Result := CNum2[10] + '-' + Digit2St(AValue - 90);
  end;
end;
```

Enfin, dans le programme principal, il faudrait déclarer une variable de type ***TValue2StBelg*** au lieu d'une variable de type ***TValue2St*** :

```
  TMainForm = class(TForm)
    // [...]
  private
    { private declarations }
    Value: TValue2StBelg; // champ de travail modifié
  public
    { public declarations }
  end;
```

[1] Dans un projet plus abouti, on aurait intérêt à inclure ces tests dans la classe elle-même et sans doute à prévoir une entrée du nombre sous forme de chaîne. Les simplifications sont ici à visée pédagogique.

L'instanciation de la classe devrait suivre ce nouveau type :

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  Value := TValue2StBelg.Create; // nouvelle création
end;
```

À peu de frais, vous aurez une version belge de l'application !

PROPRIETES ET VARIABLES

Mais revenons un peu en arrière. Une ligne du code de la méthode *SetValue* de l'unité *check* vous aura peut-être surpris :

```
Value := Value;
```

En temps ordinaire, avec une variable, cette affectation n'aurait aucun sens : pourquoi affecter à une variable la valeur qu'elle possède déjà ? Il en va différemment avec les propriétés : l'affectation à *Value* va activer la méthode *SetValue* qui va recalculer la valeur de la chaîne et l'affecter au champ *fStValue*. Pour des raisons de lisibilité, une telle écriture n'est pas courante, mais elle illustre bien la différence entre une variable et une propriété.

En fait, une propriété n'occupe pas forcément d'espace en mémoire. Elle n'est même pas forcément en rapport avec un champ interne. On peut par exemple imaginer une propriété en lecture seule qui renverrait un entier tiré au hasard :

```
{ TMyClass }

TMyClass = class
strict private
  function GetMyProp: Integer;
published
  property MyProp: Integer read GetMyProp;
end;
// [...]
implementation

function TMyClass.GetMyProp: Integer;
// *** renvoie un entier de 0 à 99 ***
begin
  Result := Random(100);
end;
```

 Une conséquence de ce mécanisme est qu'une propriété ne peut pas servir de paramètre de type *var* dans une routine. Pas plus vous ne pourrez utiliser @ ou modifier une propriété avec *Inc* ou *Dec*.

LES INFORMATIONS DE STOCKAGE

Il existe trois spécificateurs de stockage : *stored*, *default* et *nodefault*. S'ils n'ont pas d'incidence sur le comportement de la classe, ils modifient les informations stockées lors de l'enregistrement des données de la classe dans un flux.


Ces spécificateurs ne sont pas applicables aux propriétés tableaux définis ci-après[2].

Le spécificateur *stored* permet de préciser si la valeur d'une propriété publiée sera stockée dans le flux de la classe, économisant si nécessaire de la place lors de l'enregistrement d'une fiche au format LFM. Il est suivi d'un booléen obtenu grâce à une constante, une fonction sans paramètre ou un champ de la classe. Si elle n'est pas précisée, sa valeur présumée est *True*.

```
published
  property MyImage : TImage read fImage write SetImage stored False ;
```

Dans l'exemple, la propriété *MyImage* ne sera pas stockée dans le fichier LFM de la fiche à laquelle elle appartient. L'utilisation du spécificateur évite de sauvegarder des données volumineuses comme une image qui ne serait lue qu'à l'exécution.

Le spécificateur *default* permet d'indiquer quelle valeur par défaut sera utilisée pour la propriété concernée. Elle prend comme paramètre une constante du même type qu'elle et n'est autorisée que pour les types scalaires et les ensembles. Les autres types comme les chaînes de caractères, les classes ou les réels ont automatiquement une valeur implicite si bien que *default* ne s'applique pas à eux : les chaînes sont initialisées à la chaîne vide, les classes à *nil* et les réels à 0.

```
property MyProp : Integer read fMyProp write SetMyProp default 100;
property MyString: string read fMyString write SetMyString;
property MyObject: TLabel read fMyObject write SetMyObject;
```

La valeur par défaut de *MyProp* sera 100. Les valeurs de *MyString* et *MyObject* seront respectivement la chaîne vide et *nil*.


Il est important de noter qu'il est de la responsabilité du programmeur d'initialiser la propriété lors de sa création, car le spécificateur ne s'occupe que de l'enregistrement dans le flux et non des initialisations de l'objet instancié.

Le spécificateur *nodefault* permet de redéfinir une valeur de propriété marquée *default* sans spécifier de nouvelle valeur. Il est donc utilisé dans une classe descendant d'une classe ayant défini une valeur par défaut pour une propriété particulière.

[2] Voir page 118

 La valeur 2147483648 étant utilisée pour *nodefault*, elle ne convient pas pour une valeur par défaut.

 L'ensemble fonctionne comme ceci : si le spécificateur *stored* est à *True* et que la propriété en cours a une valeur différente de sa valeur par défaut ou qu'elle n'a pas de valeur par défaut, la valeur est enregistrée dans le flux. Dans un cas contraire, la valeur n'est pas enregistrée.

REDEFINITION D'UNE PROPRIETE

Lors de la définition d'une sous-classe, une propriété peut de nouveau être déclarée sans en préciser le type. Il est ainsi possible de modifier sa visibilité ou ses spécificateurs : par exemple, une propriété déclarée comme protégée sera de nouveau déclarée dans la section publique ou publiée d'une classe enfant.

Cette technique est très utilisée par des classes qui servent de moules à leurs descendants. Ainsi, le composant ***TLabel*** qui est proposé par l'unité *StdCtrls* a-t-il une définition plutôt déconcertante :

```
{ TLabel }

 TLabel = class(TCustomLabel)
 published
   property Align;
   property Alignment;
   property Anchors;
   property AutoSize;
   property BidiMode;
   property BorderSpacing;
   property Caption;
   property Color;
   property Constraints;
   property DragCursor;
   property DragKind;
   property DragMode;
   property Enabled;
   property FocusControl;
   property Font;
 // [...]
```

Le reste de sa définition est constitué uniquement de ce genre de déclarations qui ne prennent leur sens que lorsqu'on sait que ces propriétés sont en fait définies dans la section *protected* de l'ancêtre ***TCustomLabel*** : l'écriture elliptique *property* suivi du nom de la propriété héritée signifie simplement que la visibilité de cette dernière change pour devenir *published*.

Comme il est impossible de restreindre la visibilité d'une propriété, une classe ressemblant à ***TLabel*** qui aurait besoin d'en cacher certaines propriétés se servirait de ***TCustomLabel*** comme ancêtre et ne publierait que les propriétés appropriées.

De la même manière, il est possible d'ajouter un *setter* ou un *getter* que l'ancêtre ne définissait pas, de redéfinir si nécessaire une propriété, ou encore de déclarer une valeur par défaut.

[Exemple PO-16]

Pour illustrer ces possibilités, nous allons créer une unité baptisée *myclasses* qui contiendra trois classes de travail :

```
{ TMyClass }

TMyClass = class
private
  fMyName: string;
  fMyAge: Integer;
  fMyColor: TColor;
  function GetName: string;
  procedure SetMyAge(AValue: Integer);
  procedure SetMyColor(AValue: TColor);
protected
  property MyName: string read GetName;
  property MyAge: Integer read fMyAge write SetMyAge;
  property MyPreferedColor: TColor read fMyColor write SetMyColor;
public
  constructor Create;
end;

{ TMySubClass }

TMySubClass = class(TMyClass)
private
  procedure SetMyName(const AValue: string);
protected
  property MyName: string read GetName write SetMyName;
public
  constructor Create;
  property MyPreferedColor: TColor read fMyColor write SetMyColor default clBlue;
  property MyAge;
end;

{ TMyRedefClass }

TMyRedefClass = class(TMySubClass)
private
  function GetMyAge: string;
  procedure SetMyAge(const AValue: string);
published
  property MyAge: string read GetMyAge write SetMyAge;
  property MyName;
end;
```

Bien que définissant les propriétés *MyName*, *MyAge* et *MyPreferedColor*, l'ancêtre ***TMyClass*** ne rend aucune d'entre elles publiques ou publiées : il est par conséquent impossible d'y accéder en dehors des classes enfants. ***TMySubClass*** rend justement

publiques *MyPreferedColor* et *MyAge* en gratifiant la première d'une valeur par défaut. Comme la couleur par défaut doit être synchronisée entre l'enregistrement de la fiche et la réalité du champ interne de l'objet, il est nécessaire de redéfinir le constructeur *Create*. Par ailleurs, la même classe complète la propriété protégée *MyName* en lui dédiant une méthode d'écriture *SetMyName*. Enfin, ***TMyRedefClass*** redéfinit la propriété *MyAge* afin qu'elle prenne comme paramètre une chaîne de caractères en lieu et place d'un entier, et publie cette propriété modifiée ainsi que *MyName*.

Voici le code source de l'implémentation de ces trois classes :

```
implementation

const
  CDefaultName = 'Pascal';

{ TMyRedefClass }

function TMyRedefClass.GetMyAge: string;
// *** récupération de l'âge ***
begin
  Result := IntToStr(inherited MyAge);
end;

procedure TMyRedefClass.SetMyAge(const AValue: string);
// *** âge en chaîne de caractères ***
begin
  inherited MyAge := StrToInt(AValue);
end;

{ TMySubClass }

procedure TMySubClass.SetMyName(const AValue: string);
// *** nom redéfini ***
begin
  fMyName := AValue;
end;

constructor TMySubClass.Create;
// *** constructeur ***
begin
  inherited Create; // on hérite
  fMyColor := clBlue; // couleur par défaut
end;

{ TMyClass }

function TMyClass.GetName: string;
// *** nom récupéré ***
begin
  Result := fMyName;
end;

procedure TMyClass.SetMyAge(AValue: Integer);
// *** âge déterminé ***
begin
  fMyAge := AValue;
```

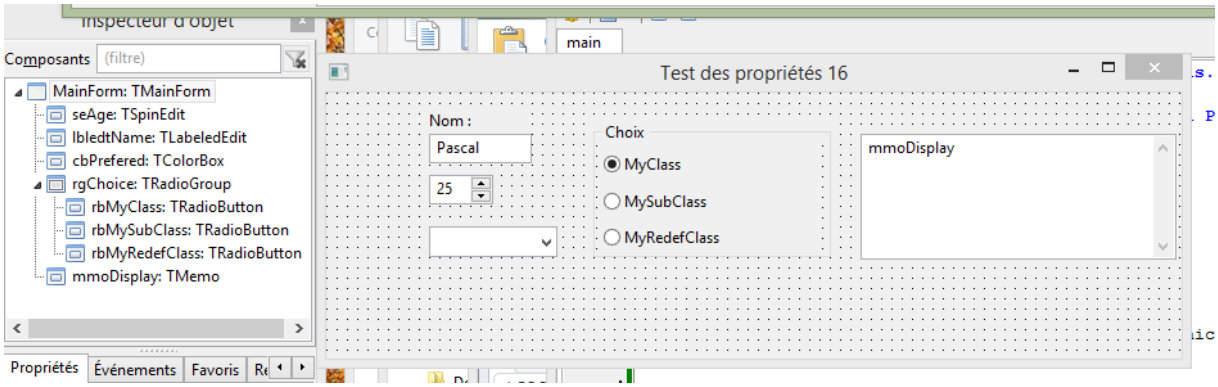
```
end;

procedure TMyClass.SetMyColor(AValue: TColor);
// *** couleur préférée déterminée ***
begin
 fMyColor := AValue;
end;

constructor TMyClass.Create;
// *** constructeur ***
begin
 fMyName := CDefaultName; // nom par défaut
end;
```

Ce code est très simple en dehors des méthodes *GetMyAge* et SetMyAge de ***TMyRedefClass*** : sans avoir besoin du spécificateur *virtual*, les propriétés sont virtuelles et peuvent par conséquent faire appel à leur ancêtre grâce à *inherited*, aussi bien pour accéder au champ *fMyAge* que pour le modifier.

Afin de tester cette unité, créez à présent un nouveau projet que vous baptiserez *testproperties16* :

- renommez la fiche principale en *MainForm* et l'unité la contenant en *main* ;
- déposez un composant ***TLabelEdit*** sur la fiche et renommez-le *lbledtName* ;
- changez la propriété *Caption* de la propriété *EditLabel* de *lbledtName* en « *Nom :* » ;
- changez la propriété *Text* du même composant en « *Pascal* » ;
- déposez un composant ***TSpinEdit*** sur la fiche et renommez-le *seAge* ;
- donnez la valeur « *25* » à la propriété *Value* de *seAge* ;
- déposez un composant ***TColorBox*** sur la fiche et renommez-le *cbPrefered* ;
- déposez un composant ***TRadioGroup*** sur la fiche et renommez-le *rgChoice* ;
- donnez la valeur « *Choix* » à la propriété *Caption* de *rgChoice* ;
- ajoutez trois ***TRadioButton*** dans *rgChoice* que vous baptiserez *rbMyClass*, *rbMySubClass* et *rbMyRedefClass* ;
- donnez respectivement les valeurs « *MyClass* », « *MySubClass* » et « *MyRedefClass* » aux propriétés *Caption* de ces boutons radio ;
- passez la propriété *Checked* du premier radio bouton à *True* ;
- déposez un composant ***TMemo*** sur la fiche et renommez-le *mmoDisplay* ;
- modifiez la propriété *ScrollBars* de *mmoDisplay* pour qu'elle vaille *ssAutoBoth*.

Voici ce que vous devriez obtenir à la conception :



- ajoutez l'unité *myclasses* à la clause *uses* de la partie interface de *main* afin d'avoir accès aux classes définie précédemment ;
- ajoutez trois variables à la fiche principale afin d'instancier les trois classes définies :

```
  private
    { private declarations }
    MyClass: TMyClass;
    MySubClass: TMySubClass;
    MyRedefClass: TMyRedefClass;
  public
```

- créez les gestionnaires *OnCreate* et *OnDestroy* de la fiche principale :

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  MyClass := TMyClass.Create; // on crée les instances
  MySubClass := TMySubClass.Create;
  MyRedefClass := TMyRedefClass.Create;
  // nettoyage du mémo
  mmoDisplay.Lines.Clear;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
  MyClass.Free; // on libère les objets
  MySubClass.Free;
  MyRedefClass.Free;
end;
```

- créez enfin l'ensemble des gestionnaires *OnChange* des composants de la fiche principale :

```
procedure TMainForm.cbPreferedChange(Sender: TObject);
// *** couleur changée ***
begin
  MySubClass.MyPreferedColor := cbPrefered.Selected;
```

```
  MyRedefClass.MyPreferedColor := cbPrefered.Selected;
end;

procedure TMainForm.lbledtNameChange(Sender: TObject);
// *** nom changé ***
begin
  MyRedefClass.MyName := lbledtName.Text;
end;

procedure TMainForm.rbMyClassChange(Sender: TObject);
// *** choix de TMyClass ***
begin
  with mmoDisplay.Lines do
  begin
    Add(MyClass.ClassName + ' ----------'); // nom de la classe
    Add('Rien d''autre n''est visible !');
    Add('----------');
    Add('');
  end;
end;

procedure TMainForm.rbMyRedefClassChange(Sender: TObject);
// *** choix de TMyRedefClass ***
begin
  with mmoDisplay.Lines do
  begin
    Add(MyRedefClass.ClassName + ' ----------'); // nom de la classe
    Add('Age : ' + MyRedefClass.MyAge); // pas de transformation en chaîne !
    Add('Couleur préférée : ' + IntToStr(MyRedefClass.MyPreferedColor));
    if MyRedefClass.MyPreferedColor = clBlue then
      Add('=> couleur par défaut !');
    Add('');
  end;
end;

procedure TMainForm.rbMySubClassChange(Sender: TObject);
// *** choix de TMySubClass ***
begin
  with mmoDisplay.Lines do
  begin
    Add(MySubClass.ClassName + ' ----------'); // nom de la classe
    Add('Age : ' + IntToStr(MySubClass.MyAge));
    Add('Couleur préférée : ' + IntToStr(MySubClass.MyPreferedColor));
    if MySubClass.MyPreferedColor = clBlue then
      Add('=> couleur par défaut !');
      Add('Nom : ' + MyRedefClass.MyName);
    Add('');
  end;
end;

procedure TMainForm.seAgeChange(Sender: TObject);
// *** âge changé ***
begin
  MySubClass.MyAge := seAge.Value; // un entier
  MyRedefClass.MyAge := IntToStr(seAge.Value); // une chaîne !
end;
```

Le programme affiche dans le mémo les éléments publics des trois objets instanciés. Comme attendu, ***TMyClass*** est une classe sans utilité pratique, ***TMySubClass*** a rendu publiques les propriétés *MyAge* et *MyPreferedColor*, et ***TMyRedefClass*** a redéclaré *MyAge* afin qu'elle accepte une chaîne de caractères comme paramètre au lieu d'un entier.

LES PROPRIETES INDEXEES

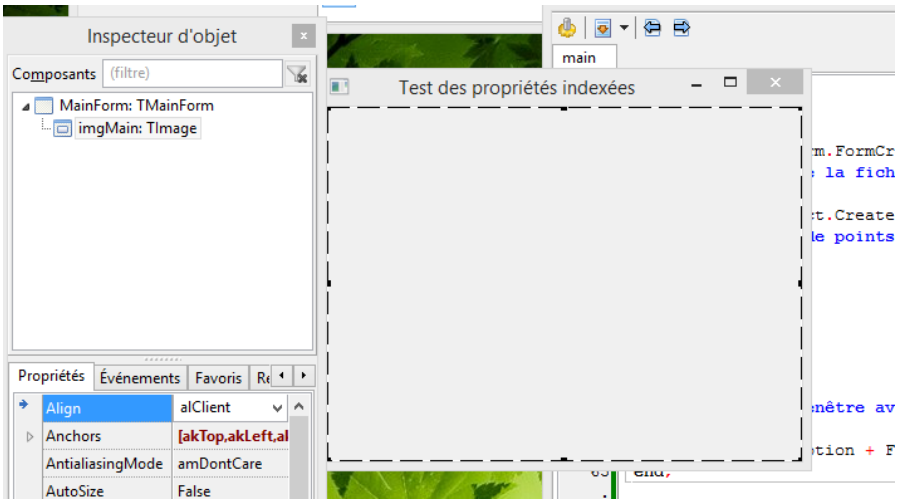
Il est aussi possible de lire et d'écrire plusieurs propriétés à partir d'une même méthode à condition qu'elles soient du même type. Dans ce cas, chaque déclaration de type de propriété sera suivie du mot réservé *index* lui-même suivi d'un entier précisant le rang de l'index. Les méthodes *getter* et *setter* seront forcément une fonction et une procédure.

[Exemple PO-17]

Pour expérimenter cette possibilité, vous allez construire une classe capable de traiter les coordonnées d'un rectangle. Ces coordonnées seront accessibles individuellement, mais partageront un même tableau en interne.

Procédez comme suit :

- créez une nouvelle application baptisée *testindexedproperties* dont l'unité principale sera renommée *main* ;
- renommez la fiche principale en *MainForm* ;
- modifiez la propriété *Caption* de cette fiche pour qu'elle affiche « *Test des propriétés indexées* » ;
- déposez un composant ***TImage*** sur la fiche principale, baptisez-le *imgMain* et changez sa propriété *Align* à *alClient* :



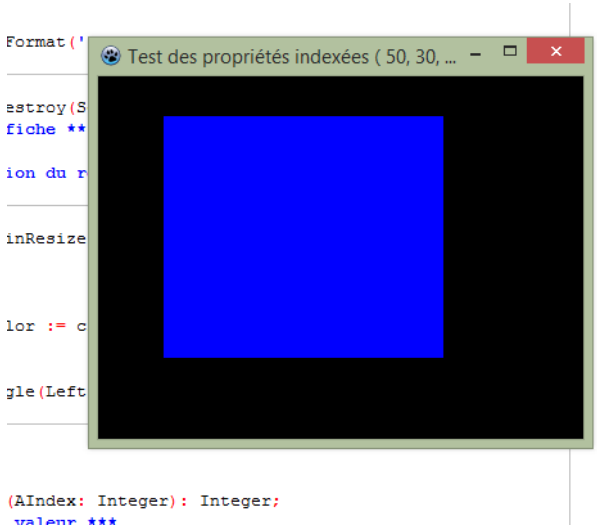
- créez un gestionnaire *OnCreate* pour la fiche principale et un gestionnaire *OnResize* pour le composant ***TImage*** ;

- complétez le code ainsi :

```
type

 { TMyRect }

 TMyRect = class
 strict private
  fValues: array[0..3] of Integer;
  function GetValue(AIndex: Integer): Integer;
  procedure SetValue(AIndex: Integer; AValue: Integer);
 public
  property Left: Integer index 0 read GetValue write SetValue;
  property Top: Integer index 1 read GetValue write SetValue;
  property Width: Integer index 2 read GetValue write SetValue;
  property Height: Integer index 3 read GetValue write SetValue;
 end;


 { TMainForm }

 TMainForm = class(TForm)
  imgMain: TImage;
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure imgMainResize(Sender: TObject);
 private
  { private declarations }
  MyRect: TMyRect;
 public
  { public declarations }
 end;

var
 MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
 MyRect := TMyRect.Create; // rectangle créé
 // affectation de points
 with MyRect do
 begin
  Left:= 50;
  Top := 30;
  Width := 320;
  Height := 250;
 end;
 // en-tête de fenêtre avec les coordonnées
 with MyRect do
  Caption := Caption + Format(' ( %d, %d, %d, %d)', [Left, Top, Width, Height]);
```

```
end;

procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
  MyRect.Free; // libération du rectangle
end;

procedure TMainForm.imgMainResize(Sender: TObject);
// *** dessin ***
begin
  // couleur bleue
  imgMain.Canvas.Brush.Color := clBlue;
  // dessin du rectangle
  with MyRect do
    imgMain.Canvas.Rectangle(Left, Top, Width - Left, Height - Top);
end;

{ TMyRect }

function TMyRect.GetValue(AIndex: Integer): Integer;
// *** récupération d'une valeur ***
begin
  Result := fValues[AIndex];
end;

procedure TMyRect.SetValue(AIndex: Integer; AValue: Integer);
// *** établissement d'une valeur ***
begin
  fValues[AIndex] := AValue;
end;

end.
```

L'exécution du programme donne ceci :



Dans l'exemple, l'index renvoie à celui d'un tableau, mais cela n'a rien d'obligatoire. Il aurait été possible de déclarer quatre champs privés et d'y accéder depuis une seule méthode grâce à une construction de type *case…of*. La déclaration aurait alors ressemblé à ceci :

```
strict private
  fLeft, fTop, fWidth, fHeight : Integer ;
```

Dans ce cas, les méthodes d'accès auraient eu cette allure :

```
{ TMyRect }

function TMyRect.GetValue(AIndex: Integer): Integer;
// *** récupération d'une valeur ***
begin
  case AIndex of
    0: Result := fLeft ;
    1: Result := fTop ;
    2: Result := fWidth ;
    3: Result := fHeight;
end;

procedure TMyRect.SetValue(AIndex: Integer; AValue: Integer);
// *** établissement d'une valeur ***
begin
  case AIndex of
    0: fLeft := AValue ;
    1: fTop := AValue;
    2: fWidth := AValue;
    3: fHeight := AValue;
end;
```

LES PROPRIETES TABLEAUX

Les *propriétés tableaux* ressemblent aux tableaux de **Pascal** et sont comme eux indicées. Leur déclaration comprend une liste de paramètres placés entre crochets et spécifiés par un nom et un type quelconque.

Voici par exemple une définition possible d'un damier :

```
const
  // *** nom des pièces ***
  CCircle = 'cercle';
  CSquare = 'carré';

type
  // *** taille du damier ***
  TSize8 = 0..7;

  // *** définition d'une case ***
  TSquare = record
    Piece: string;
    Empty: Boolean;
```

```
  end;

  { TMyBoard }

  TMyBoard = class
  strict private
    fBoard: array[TSize8, TSize8] of TSquare;
    fColors: array[0..1] of TColor;
    function GetColor(const Name: string): TColor;
    function GetUsed(X, Y : TSize8): Boolean;
    function GetName(X, Y: TSize8): string;
    procedure SetColor(const Name: string; AValue: TColor);
    procedure SetUsed(X, Y : TSize8; AValue: Boolean);
    procedure SetName(X, Y: TSize8; AValue: string);
  public
    procedure Clear;
    function Count: Integer;
    property Used[X, Y: TSize8]: Boolean read GetEmpty write SetEmpty;
    property PieceName[X, Y: TSize8]: string read GetName write SetName;
    property Color[const Name: string]: TColor read GetColor write SetColor;
  end;
```

Dans l'exemple proposé, après la déclaration du type *TSize8* comme intervalle des entiers *0..7*, viennent celles de *Used* qui est une propriété dont la tâche est de gérer l'occupation des cases d'un damier et de *PieceName* qui s'occupe du nom de la pièce présente dans une case. Enfin, la propriété *Color* associe une couleur à un type de pièce.

En plus des propriétés sont définies deux méthodes très fréquemment associées aux propriétés tableaux : la procédure *Clear* qui remet à zéro le tableau et la fonction *Count* qui en renvoie le nombre d'éléments. Leur existence s'explique par le fait que les fonctions telles que *Length* ne sont pas applicables aux propriétés tableaux.

Les propriétés indicées doivent obligatoirement définir un *getter* et un *setter*. Il est par ailleurs important de rappeler que les propriétés *ressemblent* à des variables, mais qu'elles n'en sont pas : c'est pourquoi s'il est tout à fait possible d'indicer une propriété par une chaîne de caractères, ce qu'un tableau ordinaire n'accepterait pas, les fonctions utilisées avec un tableau ne sont pas autorisées avec une propriété tableau.

À partir des déclarations faites, des écritures comme celles qui suivent seraient autorisées :

```
if MyBoard.Used[2, 4] then
  MyBoard.Color['cercle'] := clRed ;
```

Il est aussi possible de définir une propriété privilégiée dont le nom pourra être omis lors de son invocation : elle est appelée *propriété par défaut*. Pour la définir ainsi, il suffit d'ajouter *default* après sa déclaration, sans oublier de la séparer avec un point-virgule :

```
property Color[const Name: string]: TColor read GetColor write SetColor; default;
```

À présent, la propriété *Color* peut être utilisée de deux façons :

```
MyBoard.Color['carré'] := clGreen ;
MyBoard['carré'] := clGreen ;
```


Ces deux écritures sont strictement équivalentes. Une seule propriété peut être définie ainsi, mais rien n'empêche de la redéfinir dans une classe descendante.

[Exemple PO-18]

Afin d'illustrer l'utilisation des propriétés tableaux, vous allez créer un damier dont certaines cases seront occupées par des cercles ou des carrés de couleur.

Commencez par créer un nouveau projet :

- nommez-le *testpropertiesarrays* ;
- déclarez la classe ***TMyBoard*** comme proposée ci-avant et définissez-en les méthodes :

```
{ TMyBoard }

function TMyBoard.GetColor(const Name: string): TColor;
// *** couleur en fonction du nom ***
begin
 if Name = CCircle then
   Result := fColors[0]
 else
 if Name = CSquare then
   Result := fColors[1];
end;

function TMyBoard.GetUsed(X, Y : TSize8): Boolean;
// *** case vide ? ***
begin
 Result := fBoard[X, Y].Used;
end;

function TMyBoard.GetName(X, Y: TSize8): string;
// *** nom associé à la case ***
begin
 Result := fBoard[X, Y].Piece;
end;

procedure TMyBoard.SetColor(const Name: string; AValue: TColor);
// *** définition de la couleur d'une pièce ***
begin
 if Name = CCircle then
   fColors[0] := AValue
 else
 if Name = CSquare then
   fColors[1] := AValue;
end;

procedure TMyBoard.SetUsed(X, Y : TSize8; AValue: Boolean);
```

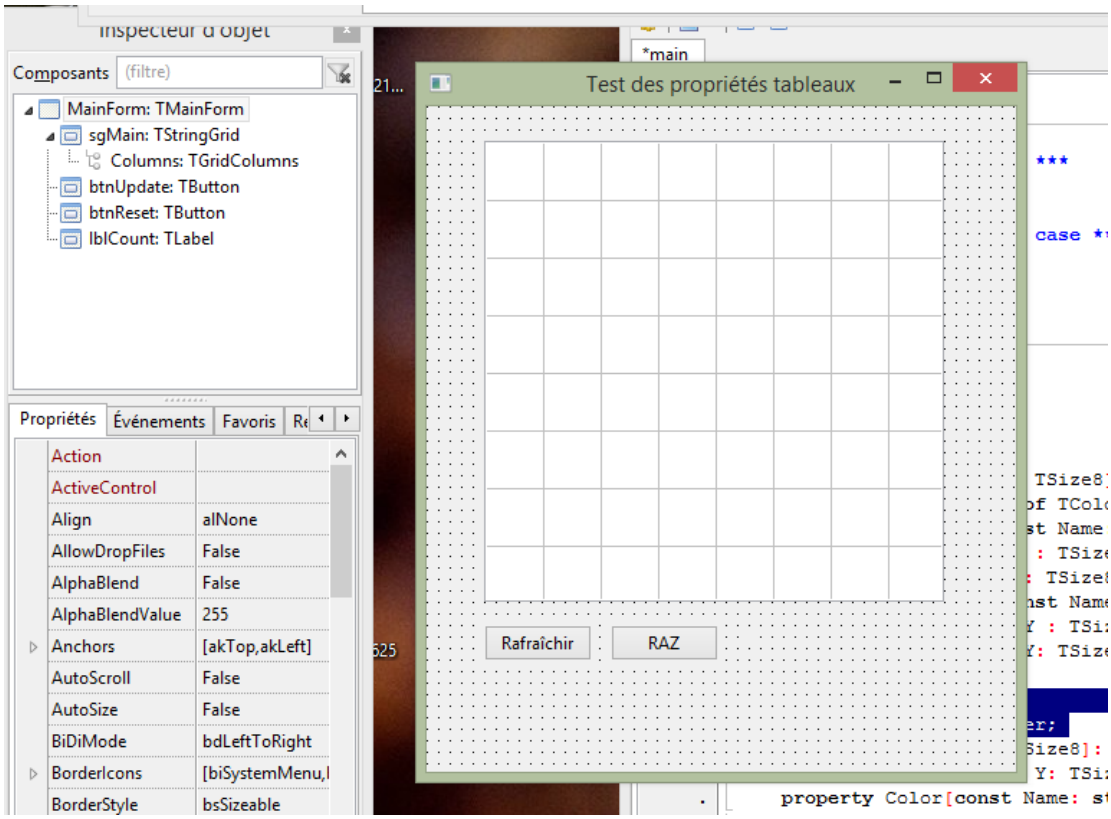
```
// *** mise à jour de l'occupation d'une case ***
begin
 fBoard[X, Y].Used := AValue;
end;

procedure TMyBoard.SetName(X, Y: TSize8; AValue: string);
// *** mise à jour du nom associé à la case ***
begin
 fBoard[X, Y].Piece := AValue;
end;

procedure TMyBoard.Clear;
// *** nettoyage du damier ***
var
 Li, Lj: Integer;
begin
  // on parcourt tout le damier
 for Li := Low(TSize8) to High(TSize8) do
  for Lj := Low(TSize8) to High(TSize8) do
  begin
   // remise à zéro de chaque élément
   fBoard[Li, Lj].Piece := '';
   fBoard[Li, Lj].Used := False;
  end;
end;

function TMyBoard.Count: Integer;
// *** nombre d'éléments du damier ***
var
 Li, Lj: Integer;
begin
 Result := 0; // résultat par défaut
 // on parcourt tout le damier
 for Li := Low(TSize8) to High(TSize8) do
  for Lj := Low(TSize8) to High(TSize8) do
   if fBoard[Li, Lj].Used then // case occupée ?
    Inc(Result); // résultat incrémenté
end;
```

La fiche principale comprendra un composant ***TStringGrid*** baptisé *sgMain*, une étiquette ***TLabel*** nommée *lblCount* et deux boutons ***TButton*** nommés *btnUpdate* et *btnReset*. La grille servira à représenter le damier. L'étiquette contiendra le nombre d'éléments du damier. Le premier bouton autorisera la modification du damier tandis que le second le remettra à zéro :

- passez les propriétés *ColCount* et *RowCount* de *sgMain* à 8 ;
- passez les propriétés *DefaultColWidth* et *DefaultRowHeight* à 40 pour obtenir des cases carrées ;
- passez les propriétés *FixedCols* et *FixedRows* à 0 afin d'éliminer la colonne et la rangée de référence ;
- passez la propriété *ScrollBars* à *ssNone* pour éviter l'affichage de barres de défilement ;
- définissez la légende de *btnRest* à « *RAZ* » ;

- définissez la légende de *btnUpdate* à « *Rafraîchir* ».

Voici à quoi devrait ressembler votre travail :



Il vous reste à coder le gestionnaire de création de la fiche et celui de sa destruction, les gestionnaires pour les boutons et celui qui dessinera chacune des cellules pour rendre compte de l'état de la case correspondante du damier.

Voici le programme proposé :

```
{ TMainForm }

 TMainForm = class(TForm)
   btnUpdate: TButton;
   btnReset: TButton;
   lblCount: TLabel;
   sgMain: TStringGrid;
   procedure btnResetClick(Sender: TObject);
   procedure btnUpdateClick(Sender: TObject);
   procedure FormCreate(Sender: TObject);
   procedure FormDestroy(Sender: TObject);
   procedure sgMainDrawCell(Sender: TObject; aCol, aRow: Integer;
     aRect: TRect; aState: TGridDrawState);
 private
   { private declarations }
   Board: TMyBoard;
 public
   { public declarations }
   procedure UpdateGrid;
```

```
  end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  // création du damier
  Board := TMyBoard.Create;
  // couleur du cercle (accès complet)
  Board.Color[CCircle] := clRed;
  // couleur du rectangle (accès raccourci)
  Board[CSquare] := clBlue;
  // dessin de la grille
  UpdateGrid;
end;

procedure TMainForm.btnUpdateClick(Sender: TObject);
// *** rafraîchissement de l'affichage ***
begin
  UpdateGrid; // grille modifiée
  sgMain.Repaint; // affichage
end;

procedure TMainForm.btnResetClick(Sender: TObject);
// *** remise à zéro ***
begin
  Board.Clear; // damier réinitialisé
  sgMain.Repaint; // affichage
  // nombre d'éléments affichés
  lblCount.Caption := IntToStr(Board.Count);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
// *** destruction de la fiche ***
begin
  Board.Free; // libération du damier
end;

procedure TMainForm.sgMainDrawCell(Sender: TObject; aCol, aRow: Integer;
  aRect: TRect; aState: TGridDrawState);
// *** dessin d'une cellule ***
begin
  with (Sender as TStringGrid) do // travail avec la grille
    if Board.Used[aCol, aRow] then // un élément à l'emplacement ?
    begin
      // couleur du fond
      Canvas.Brush.Color :=
        Board.Color[Board.PieceName[aCol, aRow]]; // couleur d'un cercle ou d'un carré
        // Board[Board.PieceName[aCol, aRow]]; // autre possibilité
```

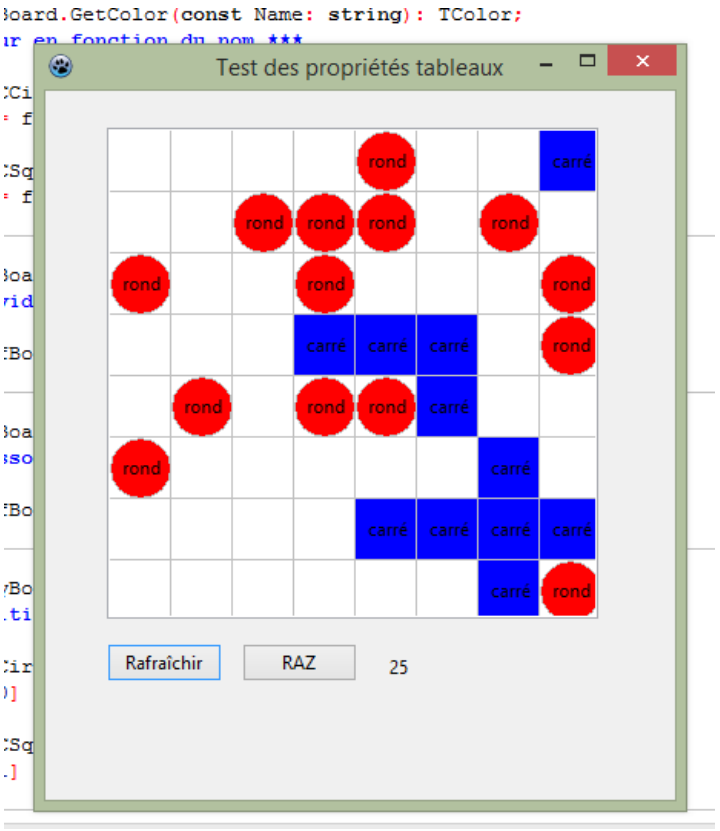
```
      if Board.PieceName[aCol, aRow] = CCircle then
        with aRect do
          Canvas.Ellipse(aRect) // cercle dessiné
      else
        Canvas.FillRect(aRect); // ou un rectangle
      // nom de la forme
      Canvas.TextOut(aRect.Left + 8, aRect.Top + 12 ,
        Board.PieceName[aCol, aRow]);
    end;
  end;

  procedure TMainForm.UpdateGrid;
  var
    Li, LX, LY: Integer;
  begin
    // quelques cercles et rectangles
    for Li := 1 to 10 do
    begin
      // coordonnées
      LX := random(8);
      LY := random(8);
      Board.PieceName[LX, LY] := CCircle; // un cercle
      Board.Used[LX, LY] := True; // case occupée
      // nouvelles coordonnées (peut-être recouvrantes)
      LX := random(8);
      LY := random(8);
      Board.PieceName[LX, LY] := CSquare; // un carré
      Board.Used[LX, LY] := True; // case occupée
    end;
    // nombre d'éléments affichés
    lblCount.Caption := IntToStr(Board.Count);
  end;
```

La méthode *UpdateGrid* tire au hasard les emplacements des carrés et des cercles. Pour simplifier le problème, aucun contrôle n'est opéré pour éviter le recouvrement d'une forme par une autre.

La méthode *sgMainDrawCell* est celle en charge de dessiner le contenu des cases. Cette méthode est appelée automatiquement pour dessiner chacune des cellules de la grille. C'est par conséquent à cet endroit qu'on décide si l'on doit dessiner un cercle, un carré ou rien du tout.

Vous aurez noté la ligne *Board.Color[Board.PieceName[aCol, aRow]];* qui peut être remplacée par une formule plus courte où le nom de la propriété par défaut aura été omis. Le même mécanisme est mis en œuvre dans le gestionnaire *OnCreate* de la fiche principale.

Une fois exécuté, le programme affichera des damiers comme celui-ci :



PROPRIETES DE CLASSE

Comme les méthodes de classe, les *propriétés de classe* sont accessibles sans référence d'objet et doivent être déclarées avec *class* en premier lieu :

```
class property Version : Integer read fVersion write SetVersion ;
class property SubVersion: Real read fSubBersion write SetSubVersion;
```

Les méthodes et les champs auxquels la propriété fait référence doivent être des méthodes statiques de classe et des champs de classe. La déclaration d'une classe comprenant les deux propriétés de classe définies plus haut pourrait être :

```
TMyClass = class
strict private
  class var
    fVersion: Integer ;
    fSubVersion: Real;
  class procedure SetVersion(const AValue: Integer); static;
  class procedure SetSubVersion(const AValue: Real); static;
  // [...]
public
  // [...]
  class property Version : Integer read fVersion write SetVersion ;
  class property SubVersion: Real read fSubBersion write SetSubVersion;
end;
```

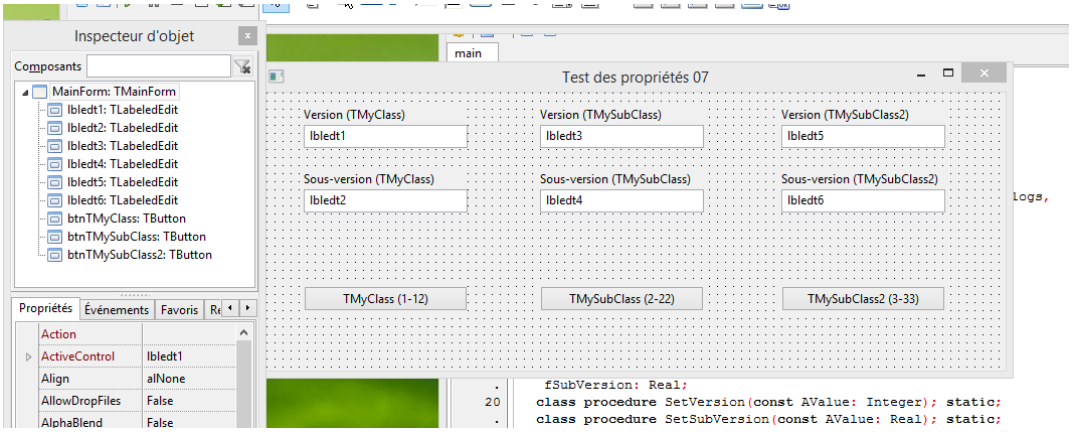
Une propriété de classe étant toujours associée à une classe particulière, il est impossible d'utiliser comme *setter* ou *getter* une méthode de classe non statique : en cas de surcharge de la méthode, la propriété de classe n'aurait plus accès aux données qui lui sont nécessaires. Par conséquent, il est obligatoire de préciser *static* à la fin de la ligne de déclaration des méthodes utilisées par une propriété.

[Exemple PO-19]

Afin de tester les propriétés de classe, vous allez créer un nouveau programme baptisé *testproperties07.lpi*. Les objectifs seront de montrer que les classes n'ont pas à être instanciées pour être accessibles *via* les propriétés de classe et que ces dernières réagissent bien de manière statique, suivant les derniers changements opérés dans les champs de classe.

Procédez donc comme suit :

- modifiez la fiche principale de telle façon qu'elle ressemble à ceci (le composant ***TLabelEdit*** est présent dans la page « *Additional* » de la palette) :



- créez les gestionnaires *OnCreate* de la fiche principale et *OnClick* des trois boutons ;
- reproduisez le code suivant dans l'unité *main* de la fiche principale (*MainForm*) :

```
type

 { TMyClass }

 TMyClass = class
 strict private
 class var
  fVersion : Integer ;
  fSubVersion: Real;
 class procedure SetVersion(const AValue: Integer); static;
 class procedure SetSubVersion(const AValue: Real); static;
 // [...]
 public
```

```
  // [...]
  class property Version : Integer read fVersion write SetVersion;
  class property SubVersion: Real read fSubVersion write SetSubVersion;
end;

  { TMySubClass }

  TMySubClass = class(TMyClass)
  strict private
    class procedure SetSubVersion(const AValue: Real); static;
    class procedure SetVersion(const AValue: Integer); static;
  public
    class property Version : Integer read fVersion write SetVersion ;
    class property SubVersion: Real read fSubVersion write SetSubVersion;
  end;

    { TMySubClass2 }

  TMySubClass2 = class(TMyClass)
  strict private
    class procedure SetSubVersion(const AValue: Real); static;
    class procedure SetVersion(const AValue: Integer); static;
  public
    class property Version : Integer read fVersion write SetVersion ;
    class property SubVersion: Real read fSubVersion write SetSubVersion;
  end;

  { TMainForm }

  TMainForm = class(TForm)
    btnTMyClass: TButton;
    btnTMySubClass: TButton;
    btnTMySubClass2: TButton;
    lbledt1: TLabeledEdit;
    lbledt2: TLabeledEdit;
    lbledt3: TLabeledEdit;
    lbledt4: TLabeledEdit;
    lbledt5: TLabeledEdit;
    lbledt6: TLabeledEdit;
    procedure btnTMyClassClick(Sender: TObject);
    procedure btnTMySubClassClick(Sender: TObject);
    procedure btnTMySubClass2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
    procedure UpdateEditors;
  public
    { public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.lfm}
```

```
{ TMySubClass2 }

class procedure TMySubClass2.SetSubVersion(const AValue: Real);
// *** numéro de sous-version (autre sous-classe) ***
begin
  fSubVersion := AValue / 100;
end;

class procedure TMySubClass2.SetVersion(const AValue: Integer);
// *** numéro de version (autre sous-classe) ***
begin
  fVersion := AValue * 100;
end;

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
begin
  TMyClass.Version := 1; // initialisation des propriétés
  TMyClass.SubVersion := 11;
  UpdateEditors; // mise à jour
end;

procedure TMainForm.btnTMyClassClick(Sender: TObject);
// *** choix de TMyClass ***
begin
  TMyClass.Version := 1;
  TMyClass.SubVersion := 11;
  UpdateEditors;
end;

procedure TMainForm.btnTMySubClassClick(Sender: TObject);
// *** choix de TMySubClass ***
begin
  TMySubClass.Version := 2;
  TMySubClass.SubVersion := 22;
  UpdateEditors;
end;

procedure TMainForm.btnTMySubClass2Click(Sender: TObject);
// *** choix de TMySubClass2 ***
begin
  TMySubClass2.Version := 3;
  TMySubClass2.SubVersion := 33;
  UpdateEditors;
end;

procedure TMainForm.UpdateEditors;
// *** mise à jour des éditeurs ***
begin
  lbledt1.Caption := IntToStr(TMyClass.Version);
  lbledt2.Caption := FloatToStr(TMyClass.SubVersion);
  lbledt3.Caption := IntToStr(TMySubClass.Version);
  lbledt4.Caption := FloatToStr(TMySubClass.SubVersion);
  lbledt5.Caption := IntToStr(TMySubClass2.Version);
  lbledt6.Caption := FloatToStr(TMySubClass2.SubVersion);
```

```
end;

{ TMySubClass }

class procedure TMySubClass.SetSubVersion(const AValue: Real);
// *** numéro de version (sous-classe) ***
begin
  fSubVersion := AValue / 10;
end;

class procedure TMySubClass.SetVersion(const AValue: Integer);
// *** numéro de sous-version (sous-classe) ***
begin
  fVersion := AValue * 10;
end;

{ TMyClass }

class procedure TMyClass.SetVersion(const AValue: Integer);
// *** numéro de version ***
begin
  fVersion := AValue;
end;

class procedure TMyClass.SetSubVersion(const AValue: Real);
// *** numéro de sous-version ***
begin
  fSubVersion := AValue;
end;

end.
```

Trois classes sont définies (***TMyClass***, ***TMySubClass***, ***TMySubClass2***) dont l'une est l'ancêtre des deux autres (***TMyClass***). Chacune de ses classes définit ses propres propriétés de classe et ses propres méthodes statiques de classe associées. On s'aperçoit à l'exécution qu'il n'est jamais nécessaire d'instancier les classes et que c'est toujours le dernier changement d'une propriété qui est pris en compte, y compris entre classes sœurs. C'est aussi uniquement à travers les méthodes de la classe invoquée que les propriétés sont modifiées[3].

Les propriétés de classes ne peuvent ni être du type *published* ni avoir de définition *stored* ou *default*.

[3] Afin de bien se rendre compte du phénomène, chaque méthode a une particularité : valeur laissée telle quelle, multipliée ou divisée par 10, multipliée ou divisée par 100.

EXEMPLE D'UTILISATION DES METHODES ET PROPRIETES DE CLASSE

Fort de votre nouvelle expérience, vous devriez comprendre le fonctionnement du programme qui va suivre : imaginez une application qui utiliserait une série de descendants de ***TMemo*** pour, par exemple, gérer telle ou telle langue.

En programmation procédurale, vous pourriez créer des tests avec *if…then…else* pour savoir quel descendant utiliser. Avec la POO, une option plus souple et plus sûre est disponible : les classes s'enregistreront elles-mêmes de manière à éviter toute erreur et le corps de la fiche principale ne sera pas affecté par l'ajout d'un nouvel objet. Mieux : la fiche principale ne saura rien des classes instanciées en dehors du fait que ce sont des descendantes d'une classe de base.

[Exemple PO-20]

Le principe est de créer une classe dérivée de la classe à créer dynamiquement. Procédez ainsi :

- créez un nouveau programme nommée *testclassmethods* ;
- sauvegardez la fiche principale sous le nom *MainForm* ;
- ajoutez une nouvelle unité au projet et baptisez-la *submemos* ;
- référencez cette unité dans la clause *uses* de la section *interface* de la fiche principale.

L'ancêtre des classes à utiliser sera défini ainsi dans l'unité *submemos* :

```
{ TSubmemo }

 TSubMemo = class(TMemo)
 strict private
  class var
    fmemos: TList;
 protected
  class procedure NewMemo;
 public
  class constructor Create;
  class destructor Destroy;
  class property Memos: TList read fMemos;
  class function ClassText: string; virtual;
  class function Version: Integer; virtual;
  class function ClassColor: TColor; virtual;
 end;
```

La propriété de classe *Memos* est associée à une variable de classe qui est une liste : elle comprendra toutes les classes qui se seront enregistrées. La liste ***TList*** sera créée et détruite automatiquement grâce à la déclaration d'un constructeur de classe *Create* et d'un destructeur de classe *Destroy*[4].

[4] Pour rappel : ce constructeur et ce destructeur sont invoqués respectivement avant la section *initialization* et après la section *finalization*, sans aucun appel explicite.

ClassText, *Version* et *ClassColor* sont des méthodes de classe qui renverront respectivement une chaîne de caractères, un entier et une couleur. Le choix s'est porté sur des méthodes afin de bénéficier des avantages de la virtualité : dans le cas contraire, des propriétés auraient été identiques pour toutes les classes définies.

La méthode *NewMemo* enregistrera la classe. Comme aucun appel à un constructeur pour instancier cette classe n'aura lieu, il faudra prévoir d'invoquer cette méthode dans la partie *initialization* de l'unité abritant la classe.

L'implémentation ne pose pas vraiment de problèmes :

```
{ TSubMemo }

class procedure TSubMemo.NewMemo;
// *** nouveau mémo ***
begin
  // mémo non enregistré ?
  if (fMemos.IndexOf(Self) = -1) then
    // on l'ajoute
    fMemos.Add(Self);
end;

class constructor TSubMemo.Create;
// *** création ***
begin
  // on crée la liste de travail
  fMemos := TList.Create;
end;

class destructor TSubMemo.Destroy;
// *** destruction ***
begin
  // la liste de travail est libérée
  fMemos.Free;
end;

class function TSubMemo.ClassText: string;
// *** légende 0 ***
begin
  Result := 'Mémo ANCETRE';
end;

class function TSubMemo.Version: Integer;
// *** version 0 ***
begin
  Result := 0;
end;

class function TSubMemo.ClassColor: TColor;
// *** couleur 0 ***
begin
  Result := clRed;
end;
```

Il reste à définir des descendants. Vous en créerez trois à titre d'exemples.

Voici comment compléter l'unité avec leurs déclarations :

```
{ TSubMemoOne }

 TSubMemoOne = class(TSubMemo)
 public
  class function ClassText: string; override;
  class function Version: Integer; override;
  class function ClassColor: TColor; override;
 end;

 { TSubMemoTwo }

 TSubMemoTwo = class(TSubMemo)
 public
  class function ClassText: string; override;
  class function Version: Integer; override;
  class function ClassColor: TColor; override;
 end;

 { TSubMemoThree }

 TSubMemoThree = class(TSubMemo)
 public
  class function ClassText: string; override;
  class function Version: Integer; override;
 end;
```

On voit que chacune des classes se contente de modifier les fonctions de classe. Seule ***TSubMemoThree*** se dispense de toucher à *ClassColor* pour montrer qu'elle héritera de cette méthode virtuelle grâce à ***TSubMemo***.

L'implémentation sera faite ainsi :

```
{ TSubMemoThree }

class function TSubMemoThree.ClassText: string;
// *** légende 3 ***
begin
  Result := inherited ClassText + ' + 3 !';
end;

class function TSubMemoThree.Version: Integer;
// *** version 3 ***
begin
  Result := inherited Version + 30;
end;

{ TSubMemoTwo }

class function TSubMemoTwo.ClassText: string;
// *** légende 2 ***
begin
  Result := 'Mémo DEUX';
end;
```

```
class function TSubMemoTwo.Version: Integer;
// *** version 2 ***
begin
  Result := 2;
end;

class function TSubMemoTwo.ClassColor: TColor;
// *** couleur 2 ***
begin
  Result := clGreen;
end;

{ TSubMemoOne }

class function TSubMemoOne.ClassText: string;
// *** légende 1 ***
begin
  Result := 'Mémo UN';
end;

class function TSubMemoOne.Version: Integer;
// *** version 1 ***
begin
  Result := 1;
end;

class function TSubMemoOne.ClassColor: TColor;
// *** couleur 1 ***
begin
  Result := clBlue;
end;
```

Afin de bien reconnaître les classes invoquées, chacune disposera d'une couleur, d'un texte et d'un numéro de version propres. En guise de variantes, ***TSubMemoThree*** utilise l'héritage pour récupérer le texte et le numéro de version de son ancêtre avant de les modifier à sa manière.

Pour terminer cette unité, il manque deux éléments. La section *initialization* devra ressembler à ceci :

```
initialization
  // enregistrement des classes
  TSubMemoOne.NewMemo;
  TSubMemoTwo.NewMemo;
  TSubMemoThree.NewMemo;
end.
```

Vos trois classes seront ainsi enregistrées, *NewMemo* les ajoutant dans la liste interne *fMemos*.

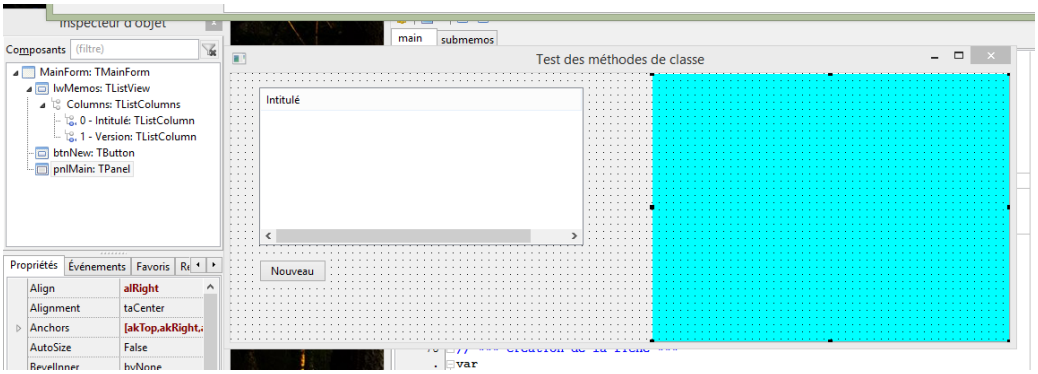
Le dernier élément à déclarer est un type :

```
TSubMemoClass = class of TSubMemo;
```

Il s'agit de déclarer une *métaclasse* : avec ***TSubMemoClass***, on ne s'intéressera pas aux objets à instancier, mais uniquement aux champs, méthodes et propriétés de classe qui viennent d'être définis.

Il est à présent temps de s'occuper de la fiche principale :

- ajoutez un ***TPanel*** aligné à droite, au fond bleu clair que vous nommerez *pnlMain* ;
- ajoutez un ***TListView*** que vous nommerez *lwMemos* ;
- cliquez sur la propriété *columns* de ce dernier pour obtenir deux éléments en cliquant deux fois sur « *ajouter* » ;
- passez à *True* la propriété *AutoSize* de deux éléments obtenus ;
- passez à « *Intitulé* » la propriété *Caption* du premier élément et à « *Version* » celle du second élément :



- passez à *True* les propriétés *RowSelect* et *ReadOnly* de *lwMemos* ;
- passez à *vsReport* la propriété *ViewStyle* de *lwMemos* afin de voir des lignes comprenant les éléments et les sous-éléments ;
- créez le gestionnaire *OnCreate* de la fiche principale :

```
procedure TMainForm.FormCreate(Sender: TObject);
// *** création de la fiche ***
var
  LPtr: Pointer;
  LNewItem: TListItem;
begin
  // remplissage de la liste
  lwMemos.Items.BeginUpdate; // évite les scintillements
  try
    // on parcourt la liste des mémos enregistrés
    for LPtr in TSubMemo.Memos do
    begin
      LNewItem := lwMemos.Items.Add; // position de l'ajout
      // légende récupérée
      LNewItem.Caption := (TSubMemoClass(LPtr)).ClassText;
```

```
      // idem pour la version
      LNewItem.SubItems.Add(IntToStr(TSubMemoClass(LPtr).Version));
      // enregistrement de la classe
      LNewItem.Data := LPtr;
    end;
  finally
    lwMemos.Items.EndUpdate; // affichage
  end;
end;
```

La propriété *Memos* de ***TSubMemo*** ayant été initialisée au démarrage du programme par une série de pointeurs vers des classes de ce type, on récupère les pointeurs et on complète la liste d'éléments du contrôle *lwMemos* avec les données appropriées : la chaîne de caractères dans *Caption*, la version dans *SubItems* et le pointeur vers la classe dans *Data*. Le tout est protégé dans une instruction *try…finally…end* afin de s'assurer que la suspension d'affichage que provoque *BeginUpdate* soit toujours levée, même en cas d'erreur, par *EndUpdate*.

Les données sont obtenues en transtypant les pointeurs avec ***TSubMemoClass*** : en effet, il s'agit de manipuler les classes en tant que telles et non des instances de ces classes.

Continuez ainsi :

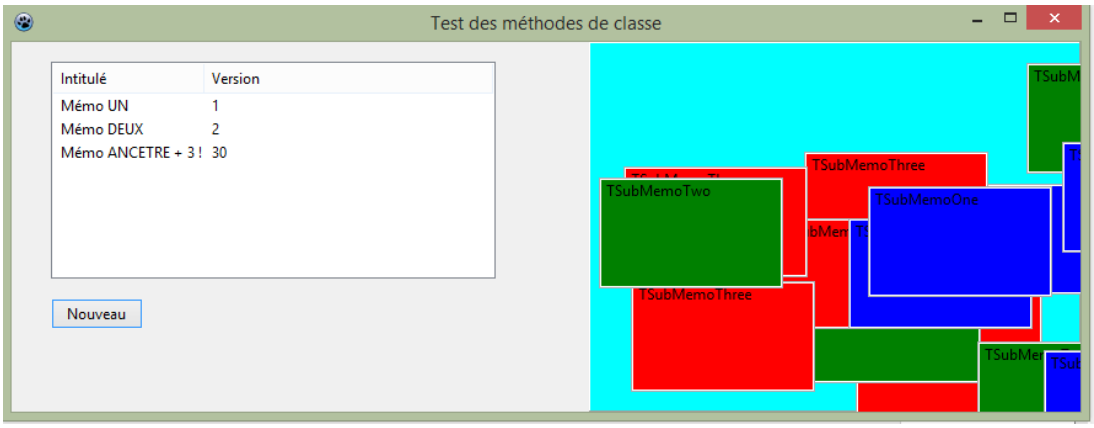
- ajoutez un bouton nommé *btnNew* ;
- associez à ce bouton le gestionnaire *OnClick* suivant :

```
procedure TMainForm.btnNewClick(Sender: TObject);
// *** nouveau mémo ***
var
  LNewMemo: TMemo;
begin
  if lwMemos.SelCount <> 0 then // des mémos enregistrés ?
  begin
    // on crée le mémo à partir de l'élément choisi dans la liste
    LNewMemo := TSubMemoClass(lwMemos.Selected.Data).Create(pnlMain);
    // position aléatoire
    LNewMemo.Left := random(400);
    LNewMemo.Top := random(300);
    // nom affiché
    LNewMemo.Caption := LNewMemo.ToString;
    // couleur de fond dépendant de la classe
    LNewMemo.Color := TSubMemoClass(lwMemos.Selected.Data).ClassColor;
    // le parent est le panneau pour l'affichage
    LNewMemo.Parent := pnlMain;
  end;
end;
```

La méthode va créer une instance de la classe qui descend de ***TMemo*** suivant l'élément sélectionné du ***TListView***. Pour ce faire, elle transtype le pointeur contenu dans la propriété *Data* pour invoquer le constructeur *Create* avec le panneau *pnlMain*

pour propriétaire. Elle définit ensuite une série de propriétés pour un affichage adapté de l'instance et termine en désignant le panneau comme *Parent*[5].

En exécutant ce programme, vous créerez autant de mémos que vous le voudrez. Ils seront du type sélectionné dans le ***TListView***. La libération des ressources se fera automatiquement puisque vous avez désigné *pnlMain* comme le propriétaire en charge de chacune des instances créées.



Que le mécanisme vous paraisse un peu complexe n'aurait rien d'étonnant, mais relevez son efficacité : la fiche principale est indépendante de l'unité en charge des classes à manipuler si bien qu'il est possible d'ajouter et de retirer des classes à volonté sans toucher quoi que ce soit à l'unité principale.

Amusez-vous ainsi :

- transformez en commentaire une ou plusieurs des classes de l'unité *submemos* et examinez le changement de comportement du programme :

```
initialization
  // enregistrement des classes
  TSubMemoOne.NewMemo;
  //  TSubMemoTwo.NewMemo; <= la classe est inaccessible
  TSubMemoThree.NewMemo;
end ;
```

- au contraire, utilisez la classe ancêtre :

```
initialization
  // enregistrement des classes
  TSubMemoOne.NewMemo;
  TSubMemoTwo.NewMemo;
  TSubMemoThree.NewMemo;
  TSubMemo.NewMemo ; // la classe ancêtre
end ;
```

[5] Désigner un *Parent* pour créer dynamiquement un contrôle est une obligation afin qu'il sache où il doit être dessiné.

Encore mieux, créez votre propre descendant de ***TSubMemo***, en lui ajoutant si nécessaire de nouvelles propriétés et méthodes. Vous constaterez que vous n'aurez jamais à modifier votre fiche principale qui saura manipuler les nouvelles classes sans même savoir ce pour quoi elles sont faites !

BILAN

Dans ce chapitre, vous avez appris à :

- ✓ déclarer, définir, et modifier une propriété ;
- ✓ distinguer une propriété d'une variable ;
- ✓ manipuler les *getters* et les *setters* ;
- ✓ connaître les spécificateurs de stockage ;
- ✓ reconnaître et traiter les différents types de propriétés (simples, indexées, tableaux, de classe).