Créer un variant personnalisé

par Eric GASPARD

Date de publication : 02 Octobre 2010

Dernière mise à jour :

Tutoriel pour vous apprendre comment créer son propre variant au sein de votre programme.

I - Introduction

Dans le langage Delphi, un variant est un type de donnée bien particulier. Dans son utilisation on a l'impression que c'est un type "mutant", changeant de type en cours de route suivant l'utilité qu'on en a alors que derrière la valeur stockée reste la même.
Le OleVariant est également au coeur de la programmation COM en *late-binding*, permettant d'appeler méthodes et propriétés de l'objet COM qui pourtant inconnues au moment de la compilation.

En réalité c'est toute une mécanique de *cast* et de redéfinition des opérateurs pour les 18 types de variant natif plus une mécanique d'invocation similaire à celle de **IDispatch**.
Cependant sortis des types natifs, il n'est pas possible d'affecter d'autres type d'objets afin de bénéficier de cet aspect "mutable" et "maléable".

Le langage Delphi, propose néanmoins un moyen d'étendre ce nombre de type de variant par le biais de la classe abstraite **TCustomVariantType**. Nous allons voir dans ce tutoriel comment implémenter cette dernière.

II - Principes de base

II-A - La structure d'un variant

Un variant est définit par un *record* : le **TVarData**. La taille de cette structure fait 16 octets, les deux premiers octets sont réservés à l'identifieur du type de variant (**TVarType**), les 14 suivants sont dédiés au stockage des données propres au variant.
Un **SizeOf** d'un variant renverra toujours 16, néanmoins la partie donnée du **TVarData** permet notamment de stocker des pointeurs et donc d'étendre la limitation de 14 octets par ce biais là. De fait la taille effective d'un variant peut dépasser les 16 octets à partir du moment où ce dernier garde des références sur des objets ou des *record*.

Le **TVarData** utilise la partie variable des enregistrements pour donner à chaque type de variant natif sa propre implémentation pour sa partie donnée. Par exemple un variant de type entier aura comme **VType** "**varInteger**" et pourra accéder à la valeur entière via le champ **VInteger**.
Les méthodes de la classe **TInvokeableVariantType** utilise le type **TVarData** plutôt que le type **Variant** pour faire transiter les variants, notamment pour éviter des *cast* automatiques qui seraient innoportuns pour les traitements.

II-B - L'identifieur de variant (TVarType)

Chaque type de variant possède son propre identifieur **TVarType** (qui est en réalité un simple *Word*). Votre nouveau type de variant devra lui aussi en avoir un cependant par défaut le système vous en attribuera un automatiquement lorsque votre classe héritant de **TInvokeableVariantType** s'enregistrera auprès du système. Notez que cet identifieur dépendra alors des autres types de variant personnalisés précédemment enregistré par rapport au votre et peut donc varier d'un programme à l'autre.
Il est cependant tout à faire possible d'affecter un identifieur fixe à votre type de variant si vous en avez l'utilité.

Les identifieurs systèmes vont de Ox0000 à Ox0100 (256) à cela Delphi ajoute une marge de 14 ce qui fait que le premier identifieur est Ox010F.
Le système autorise Ox06FF (1791) type de variant personnalisé sans compter les 14 réservés (soit 1777 type de variant 'utilisateur') ce qui le dernier identifieur aura la valeur Ox07FF.

Il existe également **varArray** (Ox2000) et **varByRef** (Ox4000) qui ne sont pas des identifieurs mais des drapeaux venant s'ajouter aux identifieur.
varArray définit que le variant contient un tableau de variant tandis que **varByRef** indique que le variant possède une référence à une valeur du type du variant plutôt que de contenir la valeur directement.
La constante **varTypeMask** (Ox0FFF) est un masque vous permettant de ne garder que les bits attribués à l'identifieur lors de comparaisons.

III - Implémenter un variant personnalisé

Nous allons maintenant nous attaquer au gros du travail. Pour illuster le propos, nous allons créer un variant contenant les coordonnées X et Y d'un point géométrique sur un plan en deux dimensions.

*A l'exception des utilitaires vu en III-C, toute l'implémentation du variant n'a pas besoin d'être visible pour l'utilisateur. Il est donc préférable de mettre cette dernière dans la partie **implementation** de l'unité et de n'exposer que les utilitaires pour l'usage.*
Moins on expose de code, plus on réduit les chances de se tromper pour l'utilisateur.

III-A - Créer la structure du variant

La première des choses à faire est de créer un record similaire à **TVarData** qui va contenir la structure du nouveau variant.

```
TPointVarData = packed record
  VType: TVarType;
  X: Longint;
  Y: Longint;
  Reserved1, Reserved2, Reserved3: Word;
end;
```

Le premier champ doit toujours être de type **TVarType** pour le reste (la partie donnée) vous pouvez l'organiser comme vous voulez du moment que vous respectez la taille de 14 octets(autant de champs utilisant n'importe quel type simple). **TPointVarData** et **TVarData** faisant tout les deux 16 octets chacun, nous pouvons utiliser la magie du transtypage pour *caster* **TVarType** en **TPointVarType** pour utiliser les membres de ce dernier.

*Le record doit toujours être déclaré avec l'option **packed**.*

III-B - Création de la classe héritant de TCustomVariantType

Chaque type de variant personnalisé doit posséder une instance héritant de la classe TCustomVariantType servant de boîte à outil pour le système afin d'effectuer toutes les opérations possibles avec votre variant personnalisé (copie, transtypage, comparaison, etc...).

```
implementation
uses Variants, SysUtils;

type
  TPointVariantType = class(TCustomVariantType)
  end;

{...}

var
  PointVariantType : TPointVariantType;

{...}

initialization
  PointVariantType := TPointVariantType.Create;
finalization
  FreeAndNil(PointVariantType);
```

La classe est volontairement vide pour le moment, l'implémentation des méthodes se fera lors des chapitres suivants.

Le singleton créé s'auto-enregistrera auprès du système au niveau de son constructeur. C'est à ce moment là que l'identifieur pour le nouveau type de variant sera attribué. Vous pouvez accéder à cet identifiant via la propriété **VarType**.

Si vous voulez utiliser un identifieur fixe, il vous faut le passer en paramètre du constructeur lors de la création du singleton.

III-C - Utilitaires de création du variant

```
interface
uses Types;

// créer un variant de type "Point" représentant un point à partir de ses coordonnées
function VarPointCreate(const X, Y: Integer): Variant; overload;
// créer un variant de type "Point" représentant un point à partir d'un TPoint
function VarPointCreate(const Point: TPoint): Variant; overload;

// accesseur pour l'identifieur du type du variant
function VarPoint: TVarType;
// le variant est-il de type "Point"
function VarIsPoint(const AValue: Variant): Boolean;
// transtyper un variant vers un variant de type "Point"
function VarAsPoint(const AValue: Variant): Variant;

implementation
uses Variants, SysUtils;

{...}

function VarPointCreate(const X, Y: Integer): Variant;
begin
  VarClear(Result);
  TPointVarData(Result).VType := VarPoint;
  TPointVarData(Result).X := X;
  TPointVarData(Result).Y := Y;
end;

function VarPointCreate(const Point: TPoint): Variant;
begin
  Result := VarPointCreate(Point.X, Point.Y);
end;

function VarPoint: TVarType;
begin
  Result := PointVariantType.VarType;
end;

function VarIsPoint(const AValue: Variant): Boolean;
begin
  Result := (TVarData(AValue).VType and varTypeMask) = VarPoint;
end;

function VarAsPoint(const AValue: Variant): Variant;
begin
  if not VarIsPoint(AValue) then
   VarCast(Result, AValue, VarPoint)
  else
   Result := AValue;
end;
```

III-D - Implémenter TCustomVariantType

III-D-1 - Copier et nettoyer le variant

III-D-1-a - Copy

```
public
  procedure Copy(var Dest: TVarData; const Source: TVarData;
      const Indirect: Boolean); override;
```

Copy est l'une des deux méthodes abstraites à redéfinir, elle est notamment appelé lorsque le variant est affecté ou lorsqu'il est passé en paramètre par valeur.

```
procedure TPointVariantType.Copy(var Dest: TVarData; const Source: TVarData;
      const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
  begin
    TPointVarData(Dest).VType := VarType;
    TPointVarData(Dest).X := TPointVarData(Source).X;
    TPointVarData(Dest).Y := TPointVarData(Source).Y;
  end;
end;
```

Lorsqu'une copie est demandée indirecte (Indirect à True) alors cela signifie que Source contient non pas les valeurs directement mais une référence vers les valeurs. De fait la copie ne doit elle-même ne faire qu'une référence sans faire d'allocation.

III-D-1-b - Clear

```
public
  procedure Clear(var V: TVarData); override;
```

Clear est la seconde méthode abstraite à redéfinir, elle est appelée lorsqu'une instance du variant va être désallouée afin de désallouer toute référence dans le variant (*record*, objet, etc...).

```
procedure TPointVariantType.Clear(var V: TVarData);
begin
  SimplisticClear(V);
end;
```

Ici comme le variant ne contient rien d'autre que deux entiers il n'y a rien de particulier à désallouer, on peut donc utiliser la méthode **SimplisticClear** pour le nettoyage du variant.
Notez que de nombreuses méthodes protégées de la classe permettent d'effectuer bon nombre d'opérations simple, n'hésitez pas à les appeler.

III-D-1-c - IsClear

```
public
  function IsClear(const V: TVarData): Boolean; virtual;
```

IsClear permet de savoir si le variant est dans son état initial ou pas. Son comportement par défaut est False.
Ici nous n'avons pas moyen de déterminer si X et Y sont à leurs valeurs par défaut ou non (O,O étant un point valide), de fait nous ne redéfinirons pas cette méthode ici.
Cette méthode prend plus de sens lorsque le variant contient des pointeurs et permet de déterminer si ces pointeurs sont toujours assignés ou pas.

III-D-2 - Transtyper

III-D-2-a - Cast

```
public
  procedure Cast(var Dest: TVarData; const Source: TVarData); override;
```

Cast permet de transtyper un variant quelconque en notre variant personnalisé.
L'implémentation par défaut de cette méthode cherche à savoir si le variant source est aussi un variant personnalisé et tente alors de le transtyper en notre propre variant personnalisé.

```
procedure TPointVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
 LSource: TVarData;
 // récupère la valeur du variant sous forme d'entier et l'utilise comme X.
 // Le Y dans ce cas là vaut toujours 0.
 procedure DefaultCast();
 var
  LTemp: TVarData;
 begin
  VarDataInit(LTemp);
  try
   // si la valeur ne se transtype pas en entier alors une EVariantTypeCastError
   // se déclenche
   VarDataCastTo(LTemp, LSource, varInteger);
   TPointVarData(Dest).X := LTemp.VInteger;
   TPointVarData(Dest).Y := 0;
  finally
   VarDataClear(LTemp);
  end;
 end;
 // si la chaîne est du format '%d,%d' alors on extrait les deux valeurs X et Y
 // et on créé le variant "Point" avec ces deux valeurs
 procedure CastFromString();
 var
  StrSource: String;
  CommaPos: Integer;
 begin
  StrSource := VarDataToStr(LSource);
  CommaPos := Pos(',', StrSource);
  if CommaPos > 0 then
  begin
   TPointVarData(Dest).X := StrToInt(System.Copy(StrSource, 1, CommaPos-1));
   TPointVarData(Dest).Y := StrToInt(System.Copy(StrSource, CommaPos+1, Length(StrSource)));
  end
  else
   DefaultCast;
 end;
begin
  VarDataInit(LSource);
  try
   // créer une copie du variant source pour éviter tout effet de bord
   VarDataCopyNoInd(LSource, Source);

   if VarDataIsStr(LSource) then
    CastFromString
   else
```

```
    DefaultCast;

   // affecter l'identifiant du variant "Point" au variant de sortie.
    Dest.VType := VarType;
  finally
   VarDataClear(LSource);
  end;
end;
```

Ici notre variant autorise deux cas de transtypage valide.
Si le variant source est une chaîne alors on cherche à voir si il est de la forme "X,Y" auquel cas on extrait les deux valeurs et on remplit notre variant avec.
Sinon notre *cast* par défaut tente de récupérer une valeur entière à partir du variant source et l'utilise pour valuer X, Y valant toujours 0 dns ce cas-là.

Dès lors nous sommes maintenant capable d'utiliser notre utilitaire de transtypage entre variant comme ceci : VarAsPoint('12,15') et de récupérer un variant "Point" correctement initialisé avec les valeurs X=12 et Y=15.

Notez ici que les règles de transtypage sont complètement arbitraire, c'est vous qui décidez quels variant sont transtypables en le votre (et sous quel format ils doivent se présenter) ou pas.

*Pour créer un variant de toute pièce depuis sa structure **TVarData** il est important de l'initialiser par un appel à **VarDataInit** qui doit obligatoirement s'accompagner par la suite d'un appel à **VarDataClear**.*

III-D-2-b - CastTo

```
public
  procedure CastTo(var Dest: TVarData; const Source: TVarData;
      const AVarType: Word); override;
```

CastTo permet de transtyper notre variant vers un autre type de variant.

```
function TPointVariantType.ToString(const V: TVarData): String;
begin
  if V.VType = VarType then
    Result := IntToStr(TPointVarData(V).X) + ',' + IntToStr(TPointVarData(V).Y)
  else
    RaiseCastError;
end;

procedure TPointVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
  const AVarType: Word);
begin
  if Source.VType = VarType then
    case AVarType of
     varString:
      VarDataFromStr(Dest, ToString(Source));
     varOleStr:
      VarDataFromOleStr(Dest, ToString(Source));
     else
      RaiseCastError;
    end
  else
   RaiseCastError;
end;
```

Le seul transtypage de notre variant que nous autorisons ici est vers le type String. A cet effet nous ajouterons une méthode utilitaire **ToString** qui se chargera rendre les données du point sous la forme 'X,Y'.

Notez que ici notre implémentation inclus un test pour savoir si le variant source est bien du même type que notre variant personnalisé, auquel cas on renvoit une exception.

Un tel cas est possible d'après la ***documentation*** *lorsque le variant n'est pas affecté (**Unassigned**).*

III-D-3 - Opérateurs arithmétiques et de comparaison

De la même façon que pour les types standards, vous pouvez (et devez) redéfinir le comportement des opérateurs (+, -, =, etc...).
Les différentes méthodes utilisent les constantes suivantes pour identifier quel opérateur elle doivent traiter :

Constante	Valeur	Opérateur	Exemple
Opérateurs unaires			
opNegate	12	-	X := -X
opNot	13	Not	if not X then
Opérateurs binaires			
opAdd	0	+	X := X + Y
opSubtract	1	-	X := X - Y
opMultiply	2	*	X := X * Y
opDivide	3	/	X := X / Y
opIntDivide	4	Div	X := X div Y
opModulus	5	Mod	X := X mod Y
opShiftLeft	6	Shl	X := X shl Y
opShiftRight	7	Shr	X := X shr Y
opAnd	8	And	if X and Y then
opOr	9	Or	if X of Y then
opXor	10	Xor	if X xor Y then
opCompare	11	Uniquement utilisé par les méthodes **LeftPromotion** et **RightPromotion** pour désigner s'il s'agit d'un opérateur de comparaison	
Opérateurs de comparaison			
opCmpEQ	14	=	if X = Y then
opCmpNE	15	<>	if X <> Y then
opCmpLT	16	<	if X < Y then
opCmpLE	17	<=	if X <= Y then
opCmpGT	18	>	if X > Y then
opCmpGE	19	>=	if X >= Y then

Notez que le terme "opérateur binaire" est un faux-ami, il ne désigne pas les opérateurs dédiés aux opérations bit-à-bit mais à toutes les opérations à deux opérandes (ce qui inclus les opérations bit-à-bit).

III-D-3-a - UnaryOp

```
public
  procedure UnaryOp(var Right: TVarData; const Operator: TVarOp);
```

```
      override;
```

UnaryOp est chargé de modifier la valeur du variant suite à une opération unaire effectuée dessus.
L'implémentation par défaut de cette méthode lève une **EVariantInvalidOpError** exception.

```
procedure TPointVariantType.UnaryOp(var Right: TVarData;
  const Operator: TVarOp);
begin
  if (Right.VType = VarType) and (Operator = opNegate) then
  begin
    TPointVarData(Right).X := -TPointVarData(Right).X;
    TPointVarData(Right).Y := -TPointVarData(Right).Y;
  end
  else
   RaiseInvalidOp;
end;
```

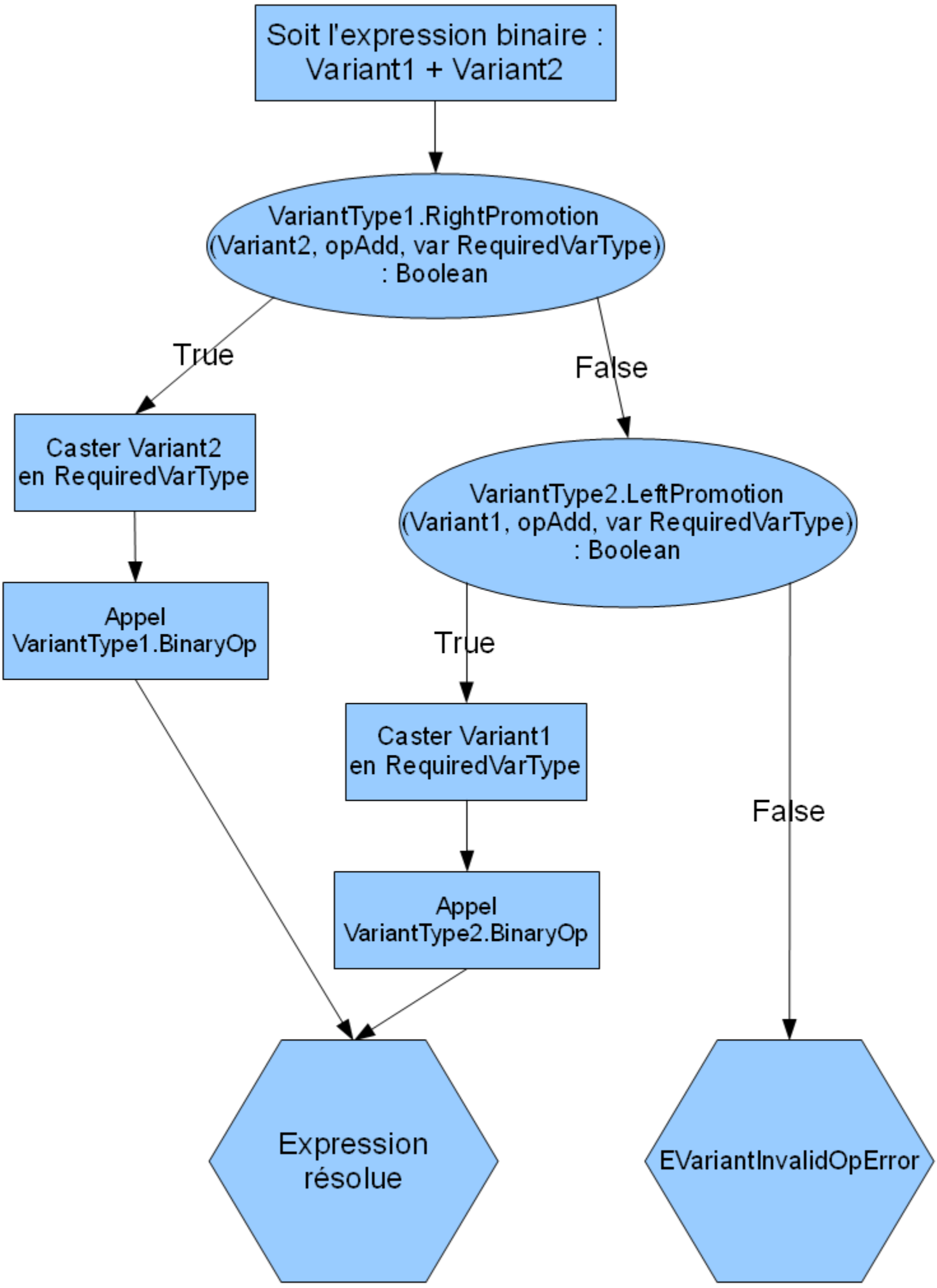
Ici seul l'opérateur -X est supporté ce qui a pour effet d'inverser le signe des coordonnées X et Y du point.

III-D-3-b - Promotion des opérandes

Lors de la résolution d'une expression on en aboutit à une suite d'opérations binaires de la forme "OpérandeGauche Opérateur OpérandeDroite" déterminée suivant la priorité des opérateurs et les parenthèses.
Une opération binaire en elle-même est résolue de la gauche vers la droite.
Lorsqu'une opération binaire est résolue entre variant, avant de passer la main aux méthodes opérateurs il faut déterminer quel type de variant va s'occuper de l'opération. C'est à cela que serve les méthodes **RightPromotion** et **LeftPromotion** qui permettent en plus de forcer le variant à être transtyper en un type de variant donné.
De fait vous pouvez filter à ce niveau les opérations effectuées par votre variant, les types de variant que vous aurez à traiter (grace au cast forcé) et cela pour chaque côté de l'opération où apparait votre variant.


Rappelez-vous cependant que bon nombre d'opération sont équivalante quelque soit l'ordre des opérandes. De fait un utilisateur ne comprendrait pas pourquoi son addition marche quand il utilise votre variant à gauche et pas à droite.

La logique d'exécution est la suivante :


Soit l'expression binaire :
Variant1 + Variant2
VariantType1.RightPromotion
(Variant2, opAdd, var RequiredVarType)
: Boolean
True
False
Caster Variant2
en RequiredVarType
Appel
VariantType1.BinaryOp
VariantType2.LeftPromotion
(Variant1, opAdd, var RequiredVarType)
: Boolean
True
Caster Variant1
en RequiredVarType
Appel
VariantType2.BinaryOp
False
Expression
résolue
EVariantInvalidOpError

Dans le cas d'une opération binaire entre un variant natif et un variant personnalisé, si aucun des deux variants ne peut traiter l'opération alors une tentative de dernière chance est effectuée :
On tente de caster le variant personnalisé vers le même type que le variant natif et on relance le processus.

III-D-3-b-i - RightPromotion

```
protected
  function RightPromotion(const V: TVarData; const Operator: TVarOp;
      out RequiredVarType: TVarType): Boolean; override;
```

RightPromotion est appelé lorsque votre variant se trouve sur l'opérande de gauche et doit répondre si il est capable de traiter l'opération avec l'opérande de droite.
L'implémentation par défaut renvoit True et demande à transtyper l'opérande de droite vers le type de variant personnalisé.

```
function TPointVariantType.RightPromotion(const V: TVarData; const Operator: TVarOp;
      out RequiredVarType: TVarType): Boolean;
begin
  RequiredVarType := VarType;
  Result := ((V.VType = VarType) or VarDataIsNumeric(V));
end;
```

Ici on exige que l'opérande de droite soit transtypé vers notre type mais qu'on ne sera capable de résoudre l'opération que si l'opérande de droite est de notre type ou bien une valeur numérique.

III-D-3-b-ii - LeftPromotion

```
protected
    function LeftPromotion(const V: TVarData; const Operator: Integer;
      out RequiredVarType: Word): Boolean; override;
```

LeftPromotion est appelé lorsque votre variant se trouve sur l'opérande de droite et doit répondre si il est capable de traiter l'opération avec l'opérande de gauche.
L'implémentation par défaut renvoit True et demande à transtyper l'opérande de gauche vers le type de variant personnalisé.

```
function TPointVariantType.LeftPromotion(const V: TVarData;
  const Operator: Integer; out RequiredVarType: Word): Boolean;
begin
  if (Operator = opAdd) and VarDataIsStr(V) then
    RequiredVarType := varString
  else
    RequiredVarType := VarType;

  Result := True;
end;
```

Ici on exige que l'opérande de gauche soit transtypé vers notre type sauf en cas de concaténation de chaîne, dans ce cas le variant est autorisé à conserver son type.
Notez qu'on ne filtre pas les opérations lorsque notre variant se trouve à droite, cela étant dit rien ne nous empêchera de lancer une exception dans la résolution de l'opérateur par la suite.

III-D-3-c - BinaryOp

```
public
  procedure BinaryOp(var Left: TVarData; const Right: TVarData;
      const Operator: TVarOp); override;
```

BinaryOp est appelé pour résoudre une opération binaire sauf s'il s'agit d'une comparaison.
L'implémentation par défaut de cette méthode lève une **EVariantInvalidOpError** exception.

```
procedure TPointVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
      const Operator: TVarOp);
begin
{
  Supporte :
    VarPoint  + VarPoint            => VarPoint
    VarPoint  + VarPoint(Numérique) => VarPoint
    VarPoint  - VarPoint            => VarPoint
    VarPoint  - VarPoint(Numérique) => VarPoint
    String    + VarPoint            => String
}
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd:  Variant(Left) := Variant(Left) + ToString(Right);
        else
          RaiseInvalidOp;
        end;
    else
      if Left.VType = VarType then
        case Operator of
          opAdd:
          begin
            TPointVarData(Left).X := TPointVarData(Left).X + TPointVarData(Right).X;
            TPointVarData(Left).Y := TPointVarData(Left).Y + TPointVarData(Right).Y;
          end;
          opSubtract:
          begin
            TPointVarData(Left).X := TPointVarData(Left).X - TPointVarData(Right).X;
            TPointVarData(Left).Y := TPointVarData(Left).Y - TPointVarData(Right).Y;
          end;
        else
          RaiseInvalidOp;
        end
      else
        RaiseInvalidOp;
    end
  else
   RaiseInvalidOp;
end;
```

Comme vous le voiyez nous n'effectuons que peu d'opérations (addition ou soustraction de point et concaténation de chaînes).
Notre implémentation de **RightPromotion** nous assure que nous n'aurons que des valeurs transtypés en notre type de variant pesonnalisé tandis que **LeftPromotion** fait de même sauf pour l'addition de String.
Cependant vous remarquerez qu'on ne traite le cas que d'une chaîne à gauche et d'un variant "Point" à droite pourtant si vous faites un VarPointCreate(1,2) + ' Point' vous obtiendrez bien une chaîne et votre variant personnalisé sera bien transtypé en String correctement alors comment est-ce que cela fonctionne ?
Si l'on reprend la logique d'exécution vu dans le paragraphe sur la promotion des opérandes, **RightPromotion** est appelé sur notre type de variant qui retourne False puisqu'il s'agit d'une chaîne et pas d'un numérique ou d'un variant "Point". La main passe alors au **LeftPromotion** du type de variant String qui répond True mais à condition que notre variant personnalisé soit transtypé en String.

De fait l'exécution qui suit transtype notre variant personnalisé (via un **CastTo**) en String puis appelle le BinaryOp du type de variant String qui effectue la concaténation entre les deux chaînes.

III-D-3-d - Compare

```
public
  procedure Compare(const Left: TVarData; const Right: TVarData;
      var Relationship: TVarCompareResult); override;
```

Compare est appelé pour déterminer si le variant de gauche est inférieur, égal ou supérieur à celui de droite.
L'implémentation par défaut de cette méthode lève une **EVariantInvalidOpError** exception.

```
procedure TPointVariantType.Compare(const Left: TVarData; const Right: TVarData;
      var Relationship: TVarCompareResult);
begin
  if (Left.VType = VarType) and (Right.VType = VarType) then
  begin
    if (TPointVarData(Left).X = TPointVarData(Right).X)
      and (TPointVarData(Left).Y = TPointVarData(Right).Y) then
      Relationship := crEqual
    else if (TPointVarData(Left).X < TPointVarData(Right).X)
      and (TPointVarData(Left).Y < TPointVarData(Right).Y) then
      Relationship := crLessThan
    else
      Relationship := crGreaterThan;
  end
  else
    RaiseInvalidOp;
end;
```

Ici notre implémentation indique que les points sont égaux si leurs coordonnées sont égales. Autrement le Point à gauche est inférieur à celui de droite si ses deux coordonnées sont inférieures à celles du point de droite.

***Compare** requiert que vous tranchiez si l'opérande de gauche est inférieure, égale ou supérieure à celle de droite. Cependant déterminer si votre type est inférieur ou supérieur à un autre peut ne pas avoir de sens, en outre la valeur de retour de cette méthode ne permet pas de renvoyer "pas égal". Dans une telle situation vous devrez redéfinir **CompareOp**.*

III-D-3-e - CompareOp

```
public
  function CompareOp(const Left, Right: TVarData;
      const Operator: TVarOp): Boolean; virtual;
```

CompareOp est appelé pour effectuer une comparaison entre les deux opérandes, Operator étant la nature de la comparaison.
L'implémentation par défaut appelle la méthode **Compare** pour savoir si les deux opérandes sont égales, inférieures ou supérieures et en déduit son résultat par logique.
Si vous êtes capable de dire si votre variant est inférieur, supérieur ou égal à un autre alors ne redéfinissez pas cette méthode (comme c'est le cas dans notre exemple), redéfinissez **Compare**. Autrement répondez à chacune des 6 actions de comparaisons de manière appropriée.

IV - Etendre les fonctionnalités de votre variant

Jusqu'ici nous avons vu comment implémenter les fonctionnalités nécessaires pour que votre variant puisse interagir correctement avec les autres. Néanmoins il est possible d'étendre les fonctionnalités que propose votre variant, notamment en lui donnant la possibilité de répondre lorsque l'utilisateur appelle des propriétés ou des méthodes survoter variant.

IV-A - Implémenter TInvokeableVariantType

TInvokeableVariantType est une classe abstraite héritant de **TCustomVariantType** et vous permettant d'implémenter des méthodes et des propriétés accessibles à l'utilisateur.

Les différentes à implémenter introduite par cette classe prennent toutes un paramètre *Name* et renvoyent un booléen. *Name* est le nom de la propriété ou méthode appelée par l'utilisateur ; la valeur de retour permet d'ndiquer au système si le nom de propriété ou de méthode demandée est supportée par votre variant ou non.

Si votre but est d'encapsuler une classe à l'intérieur d'un variant et d'exposer toutes ses propriétés published alors référrez-vous plutôt au chapitre IV-B sur l'implémentation de ***TPublishableVariantType*** *qui est plus adapté à votre situation.*

IV-A-1 - Propriétés

IV-A-1-a - GetProperty

```
public
  function GetProperty(var Dest: TVarData; const V: TVarData;
      const Name: string): Boolean; override;
```

GetProperty est appelé lorsque l'utilisateur veut accéder à une propriété du variant.
L'implémentation par défaut ne fait rien du tout et renvoit *False*.

```
function TPointVariantType.GetProperty(var Dest: TVarData; const V: TVarData;
      const Name: string): Boolean;
begin
  Result := V.VType = VarType;
  if Result then
    if (Name = 'X') or (Name = 'x') then
      Variant(Dest) := TPointVarData(V).X
    else if (Name = 'Y') or (Name = 'y') then
      Variant(Dest) := TPointVarData(V).Y
    else
      Result := False;
end;
```

Cette implémentation permet à l'utilisateur d'écrire directement VarPt.X ou VarPt.Y pour récupérer la valeur d'une des deux coordonnées.

IV-A-1-b - SetProperty

```
public
  function SetProperty(const V: TVarData; const Name: string;
      const Value: TVarData): Boolean; virtual;
```

SetProperty est appelé lorsque l'utilisateur veut modifier la valeur d'une des propriété du variant.
L'implémentation par défaut ne fait rien du tout et renvoit *False*.

```
function TPointVariantType.SetProperty(const V: TVarData; const Name: string;
      const Value: TVarData): Boolean;
var
  LTemp: TVarData;
begin
  Result := (V.VType = VarType) and VarDataIsNumeric(Value);
  if Result then
  begin
    VarDataInit(LTemp);
    try
     VarDataCastTo(LTemp, Value, varInteger);
     if (Name = 'X') or (Name = 'x') then
      TPointVarData(V).X := LTemp.VInteger
    else if (Name = 'Y') or (Name = 'y') then
      TPointVarData(V).Y := LTemp.VInteger
    else
      Result := False;

    finally
     VarDataClear(LTemp);
    end;
  end;
end;
```

Ceci aurait pût être une implémentation valable pour permettre à l'utilisateur de mettre à jour une des coordonnées du variant sauf qu'elle ne compile pas.
En effet le variant source (V) est ici passé en *const*, de fait il est impossile de le modifier lui ou un de ses membres.
Une manière de faire aurait de stocker non pas les valeurs directement mais un pointeur (ou référence) dessus, par exemple un pointeur sur TPoint. De fait même si le *record* représentant le variant est en *const* la valeur pointée elle reste toujours modifiable.

IV-A-2 - Méthodes

IV-A-2-a - DoFunction

```
public
  function DoFunction(var Dest: TVarData; const V: TVarData;
      const Name: string; const Arguments: TVarDataArray): Boolean; virtual;
```

DoFunction est appelé lorsque l'utilisateur appelle une méthode du variant qui doit renvoyer une valeur.
L'implémentation par défaut ne fait rien du tout et renvoit *False*.

IV-A-2-b - DoProcedure

```
public
  function DoProcedure(const V: TVarData; const Name: string;
      const Arguments: TVarDataArray): Boolean; virtual;
```

DoProcedure est appelé lorsque l'utilisateur appelle une méthode du variant qui ne renvoie pas de valeur.
L'implémentation par défaut ne fait rien du tout et renvoit *False*.