

Partie III : Créer un composant graphique

par Sébastien Doeraene (sjrd.developpez.com)


Date de publication : 01/06/2006

Dernière mise à jour :

Dans cette troisième partie, nous allons construire un composant graphique de toutes pièces. Il existe deux types de composants graphiques : ceux qui ont un handle - les composants Windows - et qui peuvent donc obtenir le focus et intercepter les événements claviers, et ceux qui n'en ont pas, et qui se dessinent donc sur le canevas de leur parent plutôt que sur leur propre canevas.

Dans cette partie, nous verrons le second type ; nous laisserons le premier pour la partie suivante.

 *Ce tutoriel a été écrit avec le support de Delphi 2005 édition Architecte. Si vous possédez une version antérieure ou une édition inférieure, il est possible que vous ne puissiez pas utiliser certaines des fonctionnalités présentées ici.*

 *Certains liens dans ce tutoriel font référence à l'aide locale de Delphi 2005 - ils commencent tous par "ms-help://" -, ces liens ne fonctionnent que si vous avez enregistré ce tutoriel sur votre disque local et que vous possédez une version de Delphi supérieure ou égale à la version 2005.*

Vous êtes invités à **réagir à cet article** sur les blogs. Merci.

Introduction

I - Qu'est-ce qui fait d'un composant un composant visuel ?

II - Choix du composant à développer

III - Comment dessiner le contrôle ?

IV - Collections

IV-A - Qu'est-ce qu'une collection

IV-B - Les classes dont nous aurons besoin

IV-C - La classe TChartQuarterGraphics

IV-D - La classe TChartQuarters

IV-E - La classe TChartQuarter

IV-E-1 - Les indispensables

IV-E-2 - Propriétés de la classe

IV-E-3 - Constructeur et destructeur

IV-E-4 - Méthodes d'accès aux propriétés

IV-E-5 - Méthode Assign

IV-E-6 - Méthode Click

V - Propriétés de TCircleChart

VI - Dessiner les quartiers

VI-A - Un peu d'algorithmique

VI-B - Le code Delphi

VII - Réagir aux clics sur les quartiers

VII-A - Un événement supplémentaire au niveau de TCircleChart

VII-B - Méthodes ClickQuarter et DoClickQuarter

VII-C - Interception des événements souris

VII-D - Les méthodes PointToQuarterIndex et PointToQuarter

VII-D-1 - Méthode PointToQuarter

VII-D-2 - Algorithme de la méthode PointToQuarterIndex

VII-D-3 - Implémentation de la méthode PointToQuarterIndex

VIII - Propriétés héritées

IX - Effet de la propriété AutoSize

X - Améliorer le système de menu pop-up

XI - Afficher un hint selon le quartier pointé

XII - Ajout d'actions personnalisées

XII-A - Système des actions

XII-A-1 - Les classes en présence

XII-A-2 - La classe d'action

XII-A-3 - La classe de lien d'action

XII-A-3.1 - Posséder une référence vers le composant contrôlé

XII-A-3.2 - Déterminer si une propriété donnée est concernée par le lien

XII-A-3.3 - Mettre à jour une propriété du composant contrôlé

XII-B - Support des actions par le composant

XII-B-1 - Propriétés ActionLink et Action

XII-B-2 - Mettre à jour les propriétés lors du changement d'action

XII-B-3 - Down et OnClick : des propriétés à forte interaction

 XII-B-3.1 - SetDown

 XII-B-3.2 - Click

XII-B-4 - Ne pas enregistrer les propriétés liées dans le flux

XII-B-5 - Sécuriser la destruction des actions associées

XIII - Code complet du composant TCircleChart

XIV - Test du composant

XV - Intégration dans la palette des composants

 XV-A - Recenser le composant TCircleChart

 XV-B - Recenser l'action TChartQuarterAction

XVI - Conclusion

XVII - Liens utiles

XVIII - Remerciements

Introduction

À présent que nous connaissons plusieurs techniques de base relatives à la création de composants, nous allons pouvoir attaquer des techniques plus avancées, avec notamment le dessin de composants créés de toutes pièces.

Dans cette partie et la suivante, nous aborderons divers concepts intimement liés à la création de composants, comme les collections et les actions.

Ces deux parties sont un niveau au-dessus des deux précédentes, et il est impératif de bien maîtriser le contenu de ces dernières avant de s'aventurer dans celle-ci.

I - Qu'est-ce qui fait d'un composant un composant visuel ?

Un composant visuel est un composant qui dérive de la classe **TControl** (directement ou indirectement). Cependant, on ne dérive jamais directement de cette classe car elle est inutilisable : on dérive toujours d'un de ses deux descendants : **TGraphicControl** et **TWinControl**.

TGraphicControl est utilisé pour des contrôles dits *légers*. Ces contrôles ne peuvent pas recevoir d'entrée clavier ni contenir d'autres contrôles. Un exemple très simple est le composant **TLabel**. Ils se dessinent sur le canevas de leur parent.

Ces composants ne requérant pas la création d'une fenêtre Windows (donc d'un handle), ils nécessitent beaucoup moins de ressources (d'où leur appellation de *légers*).

C'est ce type de création que nous étudierons dans cette partie.

TWinControl, par contre, englobe un handle Windows. Il peut donc contenir d'autres contrôles et intercepter les événements claviers mais est en revanche plus *lourd* car consomme plus de ressources. Généralement, lorsqu'on crée un nouveau contrôle Windows, il est plus simple de descendre de la classe **TCustomControl** qui implémente un canevas vous permettant de le dessiner simplement.

Nous verrons ce type de composants dans la partie suivante.

II - Choix du composant à développer

J'ai choisi de développer avec vous un composant qui représentera un graphique circulaire. Ses différents quartiers seront choisis avec une propriété.

Nous verrons comment on peut utiliser les **collections** pour cette propriété qui devra enregistrer chaque quartier.

De plus, cela permettra de rappeler aux développeurs qui ont quitté l'école depuis longtemps que leurs cours de trigonométrie n'ont pas servi à rien, puisque nous utiliserons les formules trigonométriques pour dessiner les quartiers.


Nous verrons aussi comment on peut détecter dans quel quartier on a cliqué afin d'envoyer des informations détaillées au gestionnaire d'événement.

Finalement, nous verrons, par l'exemple des Hints, comment intercepter un message Windows non standard grâce à la directive de méthode **message**.

Nous nommerons ce composant **TCircleChart**. Créez une nouvelle unité **CircChart.pas** dans le paquet **ComposTutoR.bpl**.

La classe **TCircleChart** devra hériter de la classe **TGraphicControl**. C'est de cette classe qu'il faut hériter lorsqu'on veut créer un composant graphique de toutes pièces.

```
type
  TCircleChart = class(TGraphicControl)
  end;
```

 *À nouveau, nous aurions dû d'abord créer une classe **TCustomCircleChart**, mais nous oublierons cela pour nous concentrer sur le code utile du composant. Cette habitude de programmation sera détaillée dans le prochain composant, car là nous en aurons besoin.*

III - Comment dessiner le contrôle ?

La chose la plus importante dans un composant graphique est sans aucun doute son dessin. **Borland** a bien fait les choses en ce qui concerne le **TGraphicControl**, puisque celui-ci propose une méthode virtuelle **Paint** dans laquelle on peut dessiner sur le composant au moyen de sa propriété **Canvas**.

Cette méthode est déclarée protégée dans **TGraphicControl**, et il n'y a aucune raison de modifier cette visibilité, puisque cette méthode **ne doit jamais** être appelée directement.

```
protected  
procedure Paint; override;
```

Nous allons placer un code simple pour l'illustration. Ce code dessine une ellipse de la couleur indiquée par la propriété **Color**, héritée de **TControl**. Nous l'améliorerons une fois que la propriété pour les quartiers sera créée.

```
procedure TCircleChart.Paint;  
begin  
  with Canvas do  
  begin  
    Brush.Color := Color;  
    Brush.Style := bsSolid;  
    Pen.Style := bsClear;  
    Ellipse(0, 0, Width, Height);  
  end;  
end;
```

IV - Collections

Nous allons donc maintenant créer la propriété qui devra enregistrer les différents quartiers à dessiner. Cette propriété sera une propriété de type *collection*. Nous allons voir pas à pas comment créer tout le système qui tourne autour.

IV-A - Qu'est-ce qu'une collection

Une collection est une classe qui hérite de la classe **TCollection**. Cette classe est déclarée dans l'unité **Classes**.

Cette classe n'est jamais utilisée directement, tout comme **TComponent**. On en dérive toujours pour créer une classe plus spécialisée.

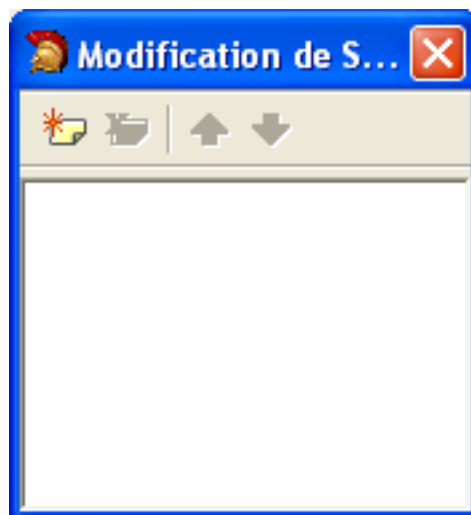
Chaque collection est une liste d'objets de types **TCollectionItem**. On peut ainsi assimiler **TCollection** à **TStrings** et **TCollectionItem** à **string**, si vous voulez. Tout comme on n'utilise jamais directement **TCollection**, on crée toujours un descendant de **TCollectionItem** aussi pour l'utiliser.

À la création d'un **TCollection**, on doit indiquer la sous-classe de **TCollectionItem** à utiliser pour les éléments. À chaque sous-classe de **TCollection** correspond donc une sous-classe de **TCollectionItem** et inversement.

La classe enfant de **TCollection** sera le type de la propriété du composant que l'on écrit. La classe enfant de **TCollectionItem** est le type des différents éléments de la collection.

Un exemple de collection que vous utilisez certainement régulièrement est la collection **TStatusPanels** dont les éléments sont des **TStatusPanel** et qui est utilisée dans la propriété **Panels** du **TStatusBar**.

En mode conception, on utilise une collection en double-cliquant dessus. L'éditeur de collection apparaît alors :



Éditeur de collection

Vous pouvez alors utiliser les boutons de la barre d'outils pour ajouter, supprimer et déplacer des éléments de la collection. Vous pouvez définir les propriétés de chaque éléments *via* l'inspecteur d'objets.

IV-B - Les classes dont nous aurons besoin

Comme nous l'avons dit plus haut, une collection est caractérisée par deux classes : une pour la collection et une pour les éléments de la collection.

Ainsi, nous définirons une classe **TChartQuarter** héritée de **TCollectionItem** et une classe **TChartQuarters** héritée de **TCollection** :

```
type
  TChartQuarter = class(TCollectionItem)
  end;

  TChartQuarters = class(TCollection)
  end;
```

À cela, nous ajouterons une déclaration avancée (*forward*) de la classe **TCircleChart**, car nous devons y faire référence dans ces deux classes :

```
type
  TCircleChart = class;

  TChartQuarter = class(TCollectionItem)
  end;

  TChartQuarters = class(TCollection)
  end;
```

Finalement, nous aurons besoin d'une petite classe (qui n'a rien à voir avec les collections) pour les informations de dessin de chaque quartier : **TChartQuarterGraphics** héritée de **TPersistent**.

```
type
  TCircleChart = class;

  TChartQuarterGraphics = class(TPersistent)
  end;

  TChartQuarter = class(TCollectionItem)
  end;

  TChartQuarters = class(TCollection)
  end;
```

*

IV-C - La classe TChartQuarterGraphics

Puisque cette classe est petite et n'a rien à voir avec les collections, nous allons la construire rapidement ici.

Cette classe sert à représenter les trois données graphiques (fond du quartier, fond du texte, et police du texte) de chaque quartier dans chacun de ses trois états (normal, enfoncé ou désactivé).

Elle est composée uniquement de trois propriétés. Deux de type **TBrush** et une de type **TFont**.

À cela, on ajoutera un constructeur, un destructeur, et une surcharge de la méthode **Assign** déclarée dans **TPersistent**.

```

type
  TChartQuarterGraphics = class(TPersistent)
  private
    FBackgroundBrush : TBrush;
    FTextBrush : TBrush;
    FFont : TFont;
    procedure SetBackgroundBrush(New : TBrush);
    procedure SetTextBrush(New : TBrush);
    procedure SetFont(New : TFont);
  public
    constructor Create(AOnChange : TNotifyEvent = nil);
    destructor Destroy; override;
    procedure Assign(Source : TPersistent); override;
  published
    property BackgroundBrush : TBrush read FBackgroundBrush write SetBackgroundBrush;
    property TextBrush : TBrush read FTextBrush write SetTextBrush;
    property Font : TFont read FFont write SetFont;
  end;
    
```

Voici aussi l'implémentation de cette classe :

```

////////////////////////////////////
/// Classe TChartQuarterGraphics ///
////////////////////////////////////

constructor TChartQuarterGraphics.Create(AOnChange : TNotifyEvent = nil);
begin
  inherited Create;
  FBackgroundBrush := TBrush.Create;
  FBackgroundBrush.OnChange := AOnChange;
  FTextBrush := TBrush.Create;
  FTextBrush.OnChange := AOnChange;
  FFont := TFont.Create;
  FFont.OnChange := AOnChange;
end;

destructor TChartQuarterGraphics.Destroy;
begin
  FFont.Free;
  FTextBrush.Free;
  FBackgroundBrush.Free;
  inherited Destroy;
end;

procedure TChartQuarterGraphics.SetBackgroundBrush(New : TBrush);
begin
  FBackgroundBrush.Assign(New);
end;

procedure TChartQuarterGraphics.SetTextBrush(New : TBrush);
begin
  FTextBrush.Assign(New);
end;

procedure TChartQuarterGraphics.SetFont(New : TFont);
begin
  FFont.Assign(New);
end;

procedure TChartQuarterGraphics.Assign(Source : TPersistent);
    
```

```
begin
  if Source is TChartQuarterGraphics then
  begin
    with TChartQuarterGraphics(Source) do
    begin
      Self.FBackgroundBrush.Assign(FBackgroundBrush);
      Self.FTextBrush.Assign(FTextBrush);
      Self.FFont.Assign(FFont);
    end;
  end else inherited;
end;
```

IV-D - La classe TChartQuarters

Nous allons maintenant développer la classe **TChartQuarters**. Comme dit plus haut, cette classe héritera de **TCollection** et sera donc la classe de collection.

Il y a plusieurs choses indispensables à réaliser lorsqu'on crée une collection. Tout d'abord, le constructeur doit passer en paramètre au constructeur hérité le type de classe des éléments. Commençons donc par le constructeur de **TChartQuarters**. Ce constructeur acceptera un paramètre de type **TCircleChart** qui identifiera le composant auquel la collection appartient :

```
constructor TChartQuarters.Create(ACircleChart : TCircleChart);
begin
  inherited Create(TChartQuarter);
  FCircleChart := ACircleChart;
end;
```

La variable **FCircleChart** est bien entendu une variable privée de type **TCircleChart**.

Nous pouvons remarquer que nous avons passé **TChartQuarter** en paramètre au constructeur hérité, afin de désigner cette classe comme étant celle des éléments de la collection.

La classe de collection doit également surcharger la méthode **GetOwner** qui doit renvoyer le propriétaire de la collection, à savoir la valeur de **FCircleChart**.

```
protected
function GetOwner : TPersistent; override;
```

L'implémentation en est tout ce qu'il y a de plus simple :

```
function TChartQuarters.GetOwner : TPersistent;
begin
  Result := FCircleChart;
end;
```

Ce sont les deux seules *obligations* de l'écriture d'une collection. Toutefois, il est utile de définir quelques méthodes spécialisées pour la manipulation côté programmation :

```
private
...
function GetItem(Index : integer) : TChartQuarter;
procedure SetItem(Index : integer; Value : TChartQuarter);
public
...
function Add : TChartQuarter;
function AddItem(Item : TChartQuarter; Index : integer) : TChartQuarter;
function Insert(Index : integer) : TChartQuarter;

property Items[index : integer] : TChartQuarter read GetItem write SetItem; default;
```

Ces cinq méthodes sont très simples à implémenter puisqu'elles ne font qu'appeler les méthodes héritées de **TCollection** avec quelques transtypages, excepté **AddItem** mais cette dernière ne mérite pas vraiment d'explications non plus.

```
function TChartQuarters.GetItem(Index : integer) : TChartQuarter;
begin
  Result := TChartQuarter(inherited GetItem(Index));
end;

procedure TChartQuarters.SetItem(Index : integer; Value : TChartQuarter);
begin
  inherited SetItem(Index, Value);
end;

function TChartQuarters.Add : TChartQuarter;
begin
  Result := TChartQuarter(inherited Add);
end;

function TChartQuarters.AddItem(Item : TChartQuarter; Index : integer) : TChartQuarter;
begin
  if Item = nil then
    Result := Add
  else
    Result := Item;
  if Assigned(Result) then
    begin
      Result.Collection := Self;
      if Index < 0 then
        Index := Count - 1;
      Result.Index := Index;
    end;
end;

function TChartQuarters.Insert(Index : integer) : TChartQuarter;
begin
  Result := AddItem(nil, Index);
end;
```

Finalement, il est souvent utile de savoir qu'un des éléments de la collection a été modifié. Cela peut être fait en surchargeant la méthode **Update** de **TCollection**. Son paramètre de type **TCollectionItem** identifie l'élément qui a été modifié. Si ce paramètre vaut **nil**, alors la modification porte sur plusieurs éléments. Dans notre cas, cela importe peu puisque nous demandons simplement un rafraîchissement du contrôle.

```
protected
  procedure Update(Item : TCollectionItem); override;
```

L'implémentation effectue juste un **Repaint** du composant propriétaire :

```

procedure TChartQuarters.Update(Item : TCollectionItem);
begin
    FCircleChart.Repaint;
end;
    
```

IV-E - La classe TChartQuarter

La classe **TChartQuarter** est la classe des éléments de la collection. Il est donc évident que c'est la plus importante. Comme nous l'avons déjà dit plus haut, cette classe doit hériter de la classe **TCollectionItem**.

IV-E-1 - Les indispensables

Tout comme pour la classe **TChartQuarters**, la classe **TChartQuarter** devra respecter deux obligations, mais elles sont différentes de celles de la collection.

La première est que le constructeur doit être la surcharge du constructeur **Create** déclaré dans **TCollectionItem** :

```

public
    constructor Create(Collection : TCollection); override;
    
```

Le paramètre **Collection** identifie évidemment la collection à laquelle appartient l'élément. Sa valeur sera accessible en dehors du constructeur par la propriété **Collection**.

La seconde obligation n'est pas, paradoxalement, nécessaire, mais si vous voulez obtenir une collection un minimum correcte, je vous conseille de la remplir. Il s'agit de surcharger la méthode **GetDisplayName** de **TCollectionItem** afin de renvoyer une chaîne de caractère qui apparaîtra dans l'éditeur de collection.

Cette méthode sans paramètre et renvoie une chaîne :

```

protected
    function GetDisplayName : string; override;
    
```

Nous verrons plus loin comment implémenter le constructeur et **GetDisplayName**.

IV-E-2 - Propriétés de la classe

Les propriétés publiées que nous donnerons à la classe **TChartQuarter** seront celles qui apparaîtront dans l'inspecteur d'objets. Pour l'exemple, nous utiliserons les propriétés suivantes :

Nom de la propriété	Type	Description
AutoCheck	boolean	Détermine s'il faut automatiquement inverser la propriété Down lorsqu'on clique sur ce quartier

Text	string	Libellé du quartier (indication de valeur non comprise)
Percent	Single	Valeur en pourcent du quartier
Down	boolean	Indique si le quartier est abaissé
Enabled	boolean	Détermine si le quartier est activé (pour les clics de souris)
GroupIndex	integer	Indique le groupe de quartiers (même fonctionnement que pour les TToolButton)
Hint	string	Indique le conseil du quartier
ShowText	boolean	Indique s'il faut afficher le libellé et le pourcentage
Graphics	TChartQuarterGraphics	Informations graphiques pour l'état normal
DownGraphics	TChartQuarterGraphics	Informations graphiques pour l'état abaissé (Down = True)
DisabledGraphics	TChartQuarterGraphics	Informations graphiques pour l'état désactivé (Enabled = False)

Toutes ces propriétés auront un accès direct (variable FXXX) en lecture et un accès indirect (méthode SetXXX) en écriture excepté **AutoCheck** et **Hint** qui auront un accès direct en écriture aussi. Nous ajoutons également une méthode **GraphicsChange** de type **TNotifyEvent** pour intercepter les changements des graphiques.

De plus, nous aurons besoin d'une variable privée de type **TCircleChart** qui indiquera le composant auquel appartient ce quartier.

```

private
    FCircleChart : TCircleChart;
    FAutoCheck : boolean;
    FText : string;
    FPercent : Single;
    FDown : boolean;
    FEnabled : boolean;
    FGroupIndex : integer;
    FHint : string;
    FShowText : boolean;
    FGraphics : TChartQuarterGraphics;
    FDownGraphics : TChartQuarterGraphics;
    FDisabledGraphics : TChartQuarterGraphics;

    procedure SetText(const New : string);
    procedure SetPercent(New : Single);
    procedure SetDown(New : boolean);
    procedure SetEnabled(New : boolean);
    procedure SetGroupIndex(New : integer);
    procedure SetGraphics(New : TChartQuarterGraphics);
    procedure SetDownGraphics(New : TChartQuarterGraphics);
    procedure SetDisabledGraphics(New : TChartQuarterGraphics);
    procedure SetShowText(New : boolean);
    procedure GraphicsChange(Sender : TObject);
published
    property AutoCheck : boolean read FAutoCheck write FAutoCheck default False;
    property Text : string read FText write SetText;
    property Percent : Single read FPercent write SetPercent;
    
```

```

property Down : boolean read FDown write SetDown default False;
property Enabled : boolean read FEnabled write SetEnabled default True;
property GroupIndex : integer read FGroupIndex write SetGroupIndex default 0;
property Hint : string read FHint write FHint;
property ShowText : boolean read FShowText write SetShowText default True;
property Graphics : TChartQuarterGraphics read FGraphics write SetGraphics;
property DownGraphics : TChartQuarterGraphics read FDownGraphics write SetDownGraphics;
property DisabledGraphics : TChartQuarterGraphics read FDisabledGraphics write
SetDisabledGraphics;
    
```

À tout cela nous ajouterons un événement **OnClick** de type **TNotifyEvent** :

```

private
    FOnClick : TNotifyEvent;
published
    property OnClick : TNotifyEvent read FOnClick write FOnClick;
    
```


IV-E-3 - Constructeur et destructeur

Les constructeur et destructeur de **TChartQuarter** n'ont rien de bien particulier. Ils initialisent les variables de la classes et les libèrent, c'est tout :

```

constructor TChartQuarter.Create(Collection : TCollection);
begin
    inherited;
    FCircleChart := TChartQuarters(Collection).FCircleChart;
    FText := '';
    FPercent := 0.0;
    FDown := False;
    FEnabled := True;
    FGroupIndex := 0;
    FHint := '';
    FShowText := True;
    FGraphics := TChartQuarterGraphics.Create(GraphicsChange);
    FDownGraphics := TChartQuarterGraphics.Create(GraphicsChange);
    FDisabledGraphics := TChartQuarterGraphics.Create(GraphicsChange);
end;

destructor TChartQuarter.Destroy;
begin
    FDisabledGraphics.Free;
    FDownGraphics.Free;
    FGraphics.Free;
    inherited;
end;
    
```

 On peut être certain que **Collection** est de type **TChartQuarters**, car nous avons dit que chaque classe de collection correspond à une classe d'éléments et inversement ; et il n'y a aucune raison pour qu'une autre classe de collection crée des éléments de type **TChartQuarter**.

IV-E-4 - Méthodes d'accès aux propriétés

Les méthodes d'accès aux diverses propriétés sont très simples. Elles ne font que modifier la variable privée correspondante et appeler la méthode **Changed** pour indiquer à la collection qu'un élément a changé. Le paramètre **AllItems** indique si la modification implique tous les éléments de la collection ou pas.

Dans notre exemple, seul le *setter* de **Percent** devra passer **True** à cette fonction, puisque le fait de modifier le pourcentage modifie également le décalage d'affichage des éléments suivants. Toutefois cela n'est que purement théorique, car nous n'avons pas utilisé la paramètre **Item** de **TChartQuarters.Update**.

J'inclus dans ce morceau de code la méthode **GraphicsChange**, qui a plus ou moins le même rôle.

```
procedure TChartQuarter.SetText(const New : string);
begin
  FText := New;
  Changed(False);
end;

procedure TChartQuarter.SetPercent(New : Single);
begin
  if New < 0.0 then exit;
  FPercent := New;
  Changed(True);
end;

procedure TChartQuarter.SetDown(New : boolean);
var I : integer;
    Quarter : TChartQuarter;
begin
  if New = FDown then exit;
  FDown := New;
  if (FGroupIndex > 0) and FDown then
    for I := 0 to Collection.Count-1 do
      begin
        Quarter := TChartQuarters(Collection)[I];
        if (Quarter <> Self) and
            (Quarter.FGroupIndex = FGroupIndex) then
          Quarter.Down := False;
        end;
      Changed(False);
    end;
end;

procedure TChartQuarter.SetEnabled(New : boolean);
begin
  FEnabled := New;
  Changed(False);
end;

procedure TChartQuarter.SetGroupIndex(New : integer);
begin
  FGroupIndex := New;
end;

procedure TChartQuarter.SetGraphics(New : TChartQuarterGraphics);
begin
  FGraphics.Assign(New);
end;

procedure TChartQuarter.SetDownGraphics(New : TChartQuarterGraphics);
begin
  FDownGraphics.Assign(New);
end;

procedure TChartQuarter.SetDisabledGraphics(New : TChartQuarterGraphics);
begin
  FDisabledGraphics.Assign(New);
```

```
end;

procedure TChartQuarter.SetShowCaption(New : boolean);
begin
  FShowCaption := New;
  Changed(False);
end;

procedure TChartQuarter.GraphicsChange(Sender : TObject);
begin
  Changed(False);
end;
```

*

Seule la méthode **SetDown** mérite des explications. Celle-ci, au cas où **Down** est positionné à **True** et que **GrouplIndex** est différent de 0, doit positionner à **False** la propriété **Down** de tous les autres éléments de la collection dont la propriété **GrouplIndex** a la même valeur, tout comme le fait la méthode **SetDown** de **TToolButton**.

IV-E-5 - Méthode Assign

Comme tout bon composant qui se respecte, nous allons redéfinir la méthode **Assign** de **TPersistent**.

```
public
  procedure Assign(Source : TPersistent); override;
```

Voici comment l'implémenter, il n'y a rien de particulier là-dedans :

```
procedure TChartQuarter.Assign(Source : TPersistent);
var ChartQuarterSource : TChartQuarter;
begin
  if Source is TChartQuarter then
  begin
    ChartQuarterSource := TChartQuarter(Source);
    FText := ChartQuarterSource.FText;
    FPercent := ChartQuarterSource.FPercent;
    FDown := ChartQuarterSource.FDown;
    FEnabled := ChartQuarterSource.FEnabled;
    FGroupIndex := ChartQuarterSource.FGroupIndex;
    FHint := ChartQuarterSource.FHint;
    FShowCaption := ChartQuarterSource.FShowCaption;
    FGraphics.Assign(ChartQuarterSource.FGraphics);
    FDownGraphics.Assign(ChartQuarterSource.FDownGraphics);
    FDisabledGraphics.Assign(ChartQuarterSource.FDisabledGraphics);
    Changed(True);
  end else inherited;
end;
```

*

IV-E-6 - Méthode Click

Nous allons ajouter une méthode **Click** semblable à la méthode **Click** de **TButton**, qui simulera un click de souris sur le quartier.

```
public
  procedure Click;
```

Cette méthode, si **Enabled** est définie à **True**, doit déclencher un événement **OnClick** puis appeler la méthode **ClickQuarter** de **TCircleChart** que nous définirons tout à l'heure.

De plus, si **AutoCheck** est définie à **True**, il faut inverser la propriété **Down** (à moins que **Down** vaille **True** et que **GroupIndex** soit différent de 0, auquel cas on ne doit pas l'inverser car il n'y aurait plus de quartier abaissé dans ce groupe).

```
procedure TChartQuarter.Click;
begin
  if Enabled then
  begin
    if AutoCheck and ((not Down) or (GroupIndex = 0)) then
      Down := not Down;
    if Assigned(FOnClick) then
      FOnClick(Self);
    FCircleChart.ClickQuarter(Index);
  end;
end;
```

Nous en avons fini avec la classe **TChartQuarter**. Nous allons pouvoir revenir à la classe **TCircleChart** et la terminer.

V - Propriétés de TCircleChart

Nous définirons cinq propriétés dans **TCircleChart** :

Nom	Type	Description
Spoke	integer	Rayon du disque
Brush	TBrush	Brush du fond du disque (sous les quartiers)
Pen	TPen	Pen utilisé pour les traits (de circonférence et de séparation entre deux quartiers)
Quarters	TChartQuarters	Collection des différents quartiers
BaseAngle	integer	Angle (en degrés) à partir duquel dessiner les quartiers (0 = à droite ; 90 = en haut)

Ces cinq propriétés doivent avoir une méthode d'accès en écriture puisqu'un changement implique qu'il faille redessiner le contrôle. Comme tout à l'heure, nous ajoutons une méthode **GraphicsChange** pour les modifications de **Brush** et de **Pen**.

```
private
    FSpoke : integer;
    FBrush : TBrush;
    FPen : TPen;
    FQuarters : TChartQuarters;
    FBaseAngle : integer;

    procedure SetSpoke(New : integer);
    procedure SetBrush(New : TBrush);
    procedure SetPen(New : TPen);
    procedure SetQuarters(New : TChartQuarters);
    procedure SetBaseAngle(New : integer);
    procedure GraphicsChange(Sender : TObject);
```

Voici les déclarations publiées de ces cinq propriétés :

```
property Spoke : integer read FSpoke write SetSpoke default 100;
property Brush : TBrush read FBrush write SetBrush;
property Pen : TPen read FPen write SetPen;
property Quarters : TChartQuarters read FQuarters write SetQuarters;
property BaseAngle : integer read FBaseAngle write SetBaseAngle default 90;
```

L'implémentation des six méthodes est triviale, excepté **SetBaseAngle** qui s'assure que la nouvelle valeur est comprise entre 0 (inclus) et 360 (exclus). À cause de l'implémentation non mathématique de l'opérateur **mod**, nous sommes obligés d'ajouter 360 au résultat s'il est négatif.

```
procedure TCircleChart.SetSpoke(New : integer);
begin
    if New <= 0 then exit;
    FSpoke := New;
    Invalidate;
```

```
end;

procedure TCircleChart.SetBrush(New : TBrush);
begin
  FBrush.Assign(New);
end;

procedure TCircleChart.SetPen(New : TPen);
begin
  FPen.Assign(New);
end;

procedure TCircleChart.SetQuarters(New : TChartQuarters);
begin
  FQuarters.Assign(New);
end;

procedure TCircleChart.SetBaseAngle(New : integer);
begin
  FBaseAngle := New mod 360;
  if FBaseAngle < 0 then inc(FBaseAngle, 360);
  Invalidate;
end;

procedure TCircleChart.GraphicsChange(Sender : TObject);
begin
  Invalidate;
end;
```

Finalement, voici l'implémentation des constructeur et destructeur pour initialiser et finaliser ces propriétés. Notez également l'affectation de **clNone** à la propriété **Color** que nous offre **TControl**.

```
constructor TCircleChart.Create(AOwner : TComponent);
begin
  inherited;
  Color := clNone;
  FSpoke := 100;
  FBrush := TBrush.Create;
  FBrush.OnChange := GraphicsChange;
  FPen := TPen.Create;
  FPen.OnChange := GraphicsChange;
  FQuarters := TChartQuarters.Create(Self);
  FBaseAngle := 90;
end;

destructor TCircleChart.Destroy;
begin
  FQuarters.Free;
  FPen.Free;
  FBrush.Free;
  inherited;
end;
```

Nous avons maintenant tous les outils en main pour dessiner les différents quartiers.

VI - Dessiner les quartiers

Maintenant que nous pouvons connaître le rayon, la couleur de fond, les quartiers, et autres informations, nous pouvons nous lancer dans l'écriture plus avancée de la méthode **Paint**, censée dessiner complètement le composant.

VI-A - Un peu d'algorithmique

Comme l'algorithme utilisé pour dessiner les quartiers n'est pas trivial, nous allons d'abord faire un peu d'algorithmique pour le rédiger.

Commençons par définir les variables dont nous aurons besoin :

Tout d'abord, nous allons calculer les coordonnées du centre du cercle ainsi que le **TRect** du carré circonscrit :

```
soit Center de type TPoint <- centre du rectangle client du contrôle;  
soit CircRect de type TRect <- carré de côté 2*Spoke centré sur Center;
```

Ces deux données, avec les propriétés, permettront de dessiner relativement simplement le contrôle.

Ensuite, nous devons dessiner le disque de fond, sauf si **Color** vaut **clNone** :

```
si Color est différent de clNone :  
    dessiner une Ellipse pleine de couleur Color sans bord dans CircRect;  
finsi;
```


Il faut maintenant dessiner le disque de milieu de plan avec les informations des propriétés **Brush** et **Pen**.

```
dessiner une Ellipse dans CircRect avec les informations  
de pinceau et de crayon données par Brush et Pen;
```

Le fond est maintenant dessiné, nous allons donc dessiner les différents quartiers.

Pour cela, nous aurons besoin de quelques variables supplémentaires :

```
// quartier courant  
soit Quarter de type TChartQuarter;  
// informations de graphismes du quartier courant dans son état  
soit Graphics de type TChartQuarterGraphics;  
// angle de départ, de milieu et de fin du quartier  
soient MinAngle, MidAngle et MaxAngle de type Single;  
// points sur la circonférence correspondants aux angles  
soient MinPt, MidPt et MaxPt de type TPoint;  
// point sur lequel centrer le texte  
soit TextPos de type TPoint;  
// le texte à afficher dans le quartier  
soit Text de type string;
```

 Les trois angles sont des valeurs d'angles absolues en radians à partir de l'angle 0 dans le cercle trigonométrique, soit à droite. **MinAngle** est la valeur de l'angle au départ du quartier, **MaxAngle** à l'extrémité, et **MidAngle** au milieu des deux. Les trois points sont les points correspondants à ces angles dans le cercle dessiné par notre composant.

MinAngle sera à chaque fois le **MaxAngle** du quartier précédent, sauf la première fois. Pour simplifier les calculs et ne pas ajouter de tests inutiles, nous initialisons donc **MaxAngle** à la valeur de départ, qui est définie par **BaseAngle** (attention : il faut transformer la valeur en radians).

```
MaxAngle <- DegreeToRadian(BaseAngle);
```

Ensuite nous pouvons faire notre itération sur les quartiers :

```
pour chaque Quarter dans Quarters :
    ...
finpour;
```

D'abord nous testons si la valeur de ce quartier ne vaut pas 0, auquel cas il ne faut pas effectuer le traitement, parce que cela serait cause d'erreurs (le dessin du quartier en lui-même faisant alors les 360° au lieu de 0) :

```
pour chaque Quarter dans Quarters :
    si Quarter.Percent vaut 0 :
        passer à l'occurrence suivante;
    finsi;
finpour;
```

Ensuite, on récupère les informations de graphismes en fonction de l'état (normal, abaissé ou désactivé) du quartier :

```
pour chaque Quarter dans Quarters :
    ...
    si Quarter est désactivé :
        Graphics <- Quarter.DisabledGraphics;
    sinon si Quarter est abaissé :
        Graphics <- Quarter.DownGraphics;
    sinon :
        Graphics <- Quarter.Graphics;
    finsi;
finpour;
```

À présent, il faut calculer les nouvelles valeurs des trois angles. Comme nous l'avons dit plus haut, **MinAngle** est simplement le **MaxAngle** de l'angle précédent. **MaxAngle**, lui, est le nouveau **MinAngle** plus la valeur en radians du quartier (connus en pourcent par la propriété **Percent**). Finalement, **MidAngle** est la moyenne des deux.

```
pour chaque Quarter dans Quarters :
    ...
    MinAngle <- MaxAngle;
    MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);
    MidAngle <- Moyenne(MinAngle, MaxAngle);
finpour;
```

Voici maintenant la partie mathématique de l'histoire : calculer les coordonnées des points correspondants à ces angles sur la circonférence de notre disque. En imaginant qu'on veuille le savoir pour le cercle trigonométrique, on peut utiliser les valeurs trigonométriques (cosinus et sinus) de l'angle directement comme abscisse et ordonnée. Ensuite, il suffit de multiplier les valeurs obtenues par la longueur du rayon, puisque, finalement, notre cercle n'est qu'un agrandissement du cercle trigonométrique avec comme facteur la longueur du rayon. Il faut encore arrondir ces valeurs à l'entier le plus proche pour pouvoir les enregistrer dans un **TPoint**.



L'ordonnée que nous obtenons alors est une ordonnées mathématique, alors que nous avons besoin de l'ordonnée informatique pour pouvoir situer le point sur le canevas. C'est pourquoi nous retranchons cette valeur à la hauteur de notre contrôle.

```

pour chaque Quarter dans Quarters :
    ...
    MinPt <- Point(Round(Spoke * Cos(MinAngle)) + Center.X,
                  Height - Round(Spoke * Sin(MinAngle)) - Center.Y);
    MidPt <- Point(Round(Spoke * Cos(MidAngle)) + Center.X,
                  Height - Round(Spoke * Sin(MidAngle)) - Center.Y);
    MaxPt <- Point(Round(Spoke * Cos(MaxAngle)) + Center.X,
                  Height - Round(Spoke * Sin(MaxAngle)) - Center.Y);
finpour;
    
```

Nous pouvons maintenant dessiner le quartier, avec la partie **BackgroundBrush** de **Graphics**.

```

pour chaque Quarter dans Quarters :
    ...
    dessiner un Quartier d'Ellipse dans CircRect à partir de MinPt jusque
    MaxPt avec les informations de pinceau de Graphics.BackgroundBrush;
finpour;
    
```

Finalement, il faut dessiner le texte avec la fonte définie par **Graphics.Font** et un fond défini par **Graphics.FontBrush**, à moins que **ShowText** ne vaille **False**. Nous centrerons le texte sur le point au milieu de **Center** et **MidPoint**.

```

pour chaque Quarter dans Quarters :
    ...
    si Quarter.ShowText :
        TextPos <- point au milieu de Center et MidPt;
        si Quarter.Text est différent de '' :
            Text <- Format('%s\r\n(%f%%)', Quarter.Text, Quarter.Percent);
        sinon :
            Text <- Format('%f%%', Quarter.Percent);
        finsi;
        écrire le texte Text avec le fond Graphics.FontBrush et
        la police Graphics.Font centré sur TextPos;
    finsi;
finpour;
    
```

Voici donc l'algorithme complet. Il ne reste plus qu'à le transposer en code Delphi.

```

soit Center de type TPoint <- centre du rectangle client du contrôle;
soit CircRect de type TRect <- carré de côté 2*Spoke centré sur Center;
// quartier courant
soit Quarter de type TChartQuarter;
    
```

```
// informations de graphismes du quartier courant dans son état
soit Graphics de type TChartQuarterGraphics;
// angle de départ, de milieu et de fin du quartier
soient MinAngle, MidAngle et MaxAngle de type Single;
// points sur la circonférence correspondants aux angles
soient MinPt, MidPt et MaxPt de type TPoint;
// point sur lequel centrer le texte
soit TextPos de type TPoint;
// le texte à afficher dans le quartier
soit Text de type string;

si Color est différent de clNone :
  dessiner une Ellipse pleine de couleur Color sans bord dans CircRect;
finssi;

dessiner une Ellipse dans CircRect avec les informations
de pinceau et de crayon données par Brush et Pen;

MaxAngle <- DegreeToRadian(BaseAngle);
pour chaque Quarter dans Quarters :
  si Quarter.Percent vaut 0 :
    passer à l'occurrence suivante;
  finssi;

si Quarter est désactivé :
  Graphics <- Quarter.DisabledGraphics;
sinon si Quarter est abaissé :
  Graphics <- Quarter.DownGraphics;
sinon :
  Graphics <- Quarter.Graphics;
finssi;

MinAngle <- MaxAngle;
MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);
MidAngle <- Moyenne(MinAngle, MaxAngle);

MinPt <- Point(Round(Spoke * Cos(MinAngle)) + Center.X,
              Height - Round(Spoke * Sin(MinAngle)) - Center.Y);
MidPt <- Point(Round(Spoke * Cos(MidAngle)) + Center.X,
              Height - Round(Spoke * Sin(MidAngle)) - Center.Y);
MaxPt <- Point(Round(Spoke * Cos(MaxAngle)) + Center.X,
              Height - Round(Spoke * Sin(MaxAngle)) - Center.Y);

dessiner un Quartier d'Ellipse dans CircRect à partir de MinPt jusque MaxPt
avec les informations de pinceau de Graphics.BackgroundBrush;

si Quarter.ShowText :
  TextPos <- point au milieu de Center et MidPt;
  si Quarter.Text est différent de '' :
    Text <- Format('%s\r\n(%f%%)', Quarter.Text, Quarter.Percent);
  sinon :
    Text <- Format('%f%%', Quarter.Percent);
  finssi;
  écrire le texte Text avec le fond Graphics.FontBrush et
  la police Graphics.Font centré sur TextPos;
finssi;
finpour;
```

VI-B - Le code Delphi

Maintenant que notre algorithme est au point, nous pouvons le transposer en Delphi. Reprenons donc la méthode **Paint** et vidons-là de son contenu actuel (nous avons mis un petit code juste pour ne pas la laisser vide).

```
procedure TCircleChart.Paint;  
begin  
end;
```

Nous allons maintenant suivre l'algo pas à pas et le transposer. Commençons par les variables.

```
soit Center de type TPoint <- centre du rectangle client du contrôle;  
soit CircRect de type TRect <- carré de côté 2*Spoke centré sur Center;  
// quartier courant  
soit Quarter de type TChartQuarter;  
// informations de graphismes du quartier courant dans son état  
soit Graphics de type TChartQuarterGraphics;  
// angle de départ, de milieu et de fin du quartier  
soient MinAngle, MidAngle et MaxAngle de type Single;  
// points sur la circonférence correspondants aux angles  
soient MinPt, MidPt et MaxPt de type TPoint;  
// point sur lequel centrer le texte  
soit TextPos de type TPoint;  
// le texte à afficher dans le quartier  
soit Text de type string;
```

À cela nous ajouterons une variable **I** de type **integer**, dont nous aurons besoin pour implémenter la boucle, ainsi qu'une variable **DrawTextRect** de type **TRect**, qui servira lors du calcul de la position du texte, qui n'est pas triviale à coder en Delphi.

```
procedure TCircleChart.Paint;  
var Center : TPoint; // Centre du cercle  
    CircRect : TRect; // Carré circonscrit au cercle  
    I : integer;  
    Quarter : TChartQuarter;  
    Graphics : TChartQuarterGraphics;  
    MinAngle, MidAngle, MaxAngle : Single;  
    MinPt, MidPt, MaxPt, TextPos : TPoint;  
    Text : string;  
    DrawTextRect : TRect;  
begin  
    // Calcul des données concernant le disque  
    Center := Point(Width div 2, Height div 2);  
    CircRect := Rect(Center.X-Spoke, Center.Y-Spoke, Center.X+Spoke, Center.Y+Spoke);  
end;
```

Passons maintenant au dessin du fond.

```
si Color est différent de clNone :  
    dessiner une Ellipse pleine de couleur Color sans bord dans CircRect;  
fini;
```

dessiner une Ellipse dans CircRect avec les informations
de pinceau et de crayon données par Brush et Pen;



Pour faciliter les choses, nous utiliserons une instruction **with** sur **Canvas**.

```
with Canvas do
```

```
begin
  // Dessin de la couleur de fond
  if Color <> clNone then
    begin
      Brush.Color := Color;
      Brush.Style := bsSolid;
      Pen.Style := psClear;
      Ellipse(CircRect);
    end;

    // Dessin du disque
    Brush.Assign(Self.Brush);
    Pen.Assign(Self.Pen);
    Ellipse(CircRect);
  end;
```

 Notez que nous avons dû utiliser **Self.Brush** et **Self.Pen** car ces deux propriétés étaient cachées par leurs homographes de **Canvas**.

```
MaxAngle <- DegreeToRadian(BaseAngle);
pour chaque Quarter dans Quarters :
  ...
finpour;
```

L'appel à la fonction **DegreeToRadian** sera remplacé par le calcul correspondant en Delphi.

Pour implémenter la boucle, nous devons itérer sur la variable **I** plutôt que sur **Quarter** à cause de la façon de coder les tableaux en Delphi.

```
// Dessin des différents quartiers
MaxAngle := FBaseAngle * Pi / 180;
for I := 0 to Quarters.Count-1 do
  begin
    Quarter := Quarters[I];
  end;
```

```
si Quarter.Percent vaut 0 :
  passer à l'occurrence suivante;
finssi;

si Quarter est désactivé :
  Graphics <- Quarter.DisabledGraphics;
sinon si Quarter est abaissé :
  Graphics <- Quarter.DownGraphics;
sinon :
  Graphics <- Quarter.Graphics;
finssi;
```

Pour ce passage la traduction est littérale :

```
if Quarter.Percent = 0.0 then Continue;
if not Quarter.Enabled then
  Graphics := Quarter.DisabledGraphics
else if Quarter.Down then
```

```
Graphics := Quarter.DownGraphics  
else  
Graphics := Quarter.Graphics;
```

Nous arrivons au *passage mathématique* :

```
MinAngle <- MaxAngle;  
MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);  
MidAngle <- Moyenne(MinAngle, MaxAngle);  
  
MinPt <- Point(Round(Spoke * Cos(MinAngle)) + Center.X,  
              Height - Round(Spoke * Sin(MinAngle)) - Center.Y);  
MidPt <- Point(Round(Spoke * Cos(MidAngle)) + Center.X,  
              Height - Round(Spoke * Sin(MidAngle)) - Center.Y);  
MaxPt <- Point(Round(Spoke * Cos(MaxAngle)) + Center.X,  
              Height - Round(Spoke * Sin(MaxAngle)) - Center.Y);
```

Ici nous nous apercevons avec joie des extensions mathématiques de Delphi : la traduction est également littérale, mis à part le remplacement de **PercentToRadian** en le calcul correspondant. Notez qu'il faut rajouter l'unité **Math** à la clause **uses** pour pouvoir utiliser certaines fonctions présentées ci-dessus.

```
// Avancement des angles  
MinAngle := MaxAngle;  
MaxAngle := MinAngle + (Quarter.Percent * 2*Pi / 100);  
MidAngle := (MinAngle + MaxAngle) / 2;  
  
// Calcul des points  
MinPt := Point(Round(Spoke * Cos(MinAngle)) + Center.X, Height - Round(Spoke * Sin(MinAngle)) -  
              Center.Y);  
MidPt := Point(Round(Spoke * Cos(MidAngle)) + Center.X, Height - Round(Spoke * Sin(MidAngle)) -  
              Center.Y);  
MaxPt := Point(Round(Spoke * Cos(MaxAngle)) + Center.X, Height - Round(Spoke * Sin(MaxAngle)) -  
              Center.Y);
```

Voici le passage qui peut faire peur aux développeurs qui ne connaissent pas bien la classe **TCanvas**. En effet, il s'agit de dessiner un quartier de disque.

```
dessiner un Quartier d'Ellipse dans CircRect à partir de MinPt jusque MaxPt  
avec les informations de pinceau de Graphics.BackgroundBrush;
```

Heureusement, la méthode **Pie** de **TCanvas** nous permet de faire cela très aisément.


Voici le prototype de cette méthode :

```
procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4 : integer);
```

Extrait de l'aide de Delphi 2005 :

Utilisez **Pie** pour dessiner dans l'image une figure en forme de part de tarte. Cette forme est définie par l'ellipse inscrite dans le rectangle déterminé par les points (X1,Y1) et (X2,Y2). La section dessinée est déterminée par deux lignes, partant du centre de l'ellipse jusqu'aux points (X3,Y3) et (X4,Y4).

Le contour est dessiné en utilisant **Pen** et la forme est remplie en utilisant **Brush**.

 *À nouveau, afin de gagner de la place et donc de la lisibilité, nous utiliserons une instruction **with** cette fois sur **CircRect**. On remarque au passage qu'il existe donc des occasions où cette instruction peut se révéler très efficace sans **begin...end**.*

```
// Dessin du quartier
Brush.Assign(Graphics.BackgroundBrush);
with CircRect do
  Pie(Left, Top, Right, Bottom, MinPt.X, MinPt.Y, MaxPt.X, MaxPt.Y);
```

Il ne reste plus que l'affichage du texte :

```
si Quarter.ShowText :
  TextPos <- point au milieu de Center et MidPt;
  si Quarter.Text est différent de '' :
    Text <- Format('%s\r\n(%f%%)', Quarter.Text, Quarter.Percent);
  sinon :
    Text <- Format('%f%%', Quarter.Percent);
  finssi;
  écrire le texte Text avec le fond Graphics.FontBrush et
  la police Graphics.Font centré sur TextPos;
finssi;
```

Si l'implémentation des trois premières instructions est triviale, il n'en est pas de même pour l'écriture du texte. Commençons toujours par ces trois instructions :

```
// Calcul de la position du texte et affichage du texte
if Quarter.ShowText then
begin
  Brush.Assign(Graphics.TextBrush);
  Font.Assign(Graphics.Font);
  TextPos := Point((Center.X+MidPt.X) div 2, (Center.Y+MidPt.Y) div 2);
  if Quarter.Text <> '' then
    Text := Format('%s#13#10'(%f%%)', [Quarter.Text, Quarter.Percent])
  else
    Text := Format('%f%%', [Quarter.Percent]);
  // écrire le texte
end;
```

Le problème de l'écriture réside dans sa dernière spécification ("centré sur TextPos"). Il faut pour cela parvenir à mesurer la largeur et la hauteur qu'occupera le texte dans la police spécifiée. Il existe bien des méthodes **TextWidth** et **TextHeight** dans **TCanvas** mais celles-ci sont imprécises.

En réalité, il n'existe aucune solution avec les méthodes de la VCL. En revanche, une API permet de le faire : **DrawText**.

Pour utiliser cette méthode dans notre cas, il faut d'abord l'appeler avec le flag **DT_CALCRECT** afin de calculer les dimensions du rectangle dans lequel s'écrira le texte. Ensuite, il faut l'appeler une seconde fois sans ce flag, cette fois avec le bon rectangle, pour écrire réellement le texte.

Avant le premier appel, nous devons initialiser le rectangle. L'impératif est alors que sa largeur soit la largeur maximale que peut prendre le texte : nous utiliserons la largeur de notre contrôle.

```
DrawTextRect := Rect(0, 0, Width, 0);
```


Ensuite, nous appelons **DrawText** comme suit :

```
DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CALCRECT or DT_CENTER or DT_NOPREFIX);
```

Le premier paramètre est un handle de contexte de dessin (*device context*) : c'est le handle de notre canevas (nous sommes toujours dans le **with** correspondant). Le second est un pointeur vers la chaîne à écrire. Le troisième est la longueur de cette chaîne ; s'il vaut -1, la chaîne est considérée comme étant à zéro terminal, ce qui est notre cas puisque nous l'avons transtypée en **PChar**. Le quatrième est le rectangle dans lequel écrire le texte. Et le dernier est un ensemble de drapeaux d'options d'écriture.

Cet appel n'écrit rien, à cause du flag **DT_CALCRECT**. En revanche, après cet appel, les propriétés **Right** et **Bottom** de **DrawTextRect** auront été ajustées à la largeur et la hauteur que prendra le texte pour s'afficher.

Nous devons maintenant modifier légèrement le rectangle obtenu pour le centrer sur **TextPos** :

 *Jamais deux sans trois, voici encore une utilisation bizarre de l'instruction **with**, puisque nous spécifions deux objets à utiliser.*

```
with TextPos, DrawTextRect do  
  DrawTextRect := Rect(X - Right div 2, Y - Bottom div 2, X + Right div 2, Y + Bottom div 2);
```

Finalement, il ne reste plus qu'à appeler **DrawText** pour la deuxième fois, sans le paramètre **DT_CALCRECT** pour écrire effectivement le texte :

```
DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CENTER or DT_NOPREFIX);
```

Nous avons finalement terminé cette méthode de dessin. Nous avons pu à cette occasion découvrir plusieurs astuces de dessin, notamment l'utilisation des fonctions trigonométrique et de l'API **DrawText**.

Voici le code complet de cette méthode :

```
procedure TCircleChart.Paint;  
var Center : TPoint; // Centre du cercle  
    CircRect : TRect; // Carré circonscrit au cercle  
    I : integer;  
    Quarter : TChartQuarter;  
    Graphics : TChartQuarterGraphics;
```

```

MinAngle, MidAngle, MaxAngle : Single;
MinPt, MidPt, MaxPt, TextPos : TPoint;
Text : string;
DrawTextRect : TRect;
begin
    // Calcul des données concernant le disque
    Center := Point(Width div 2, Height div 2);
    CircRect := Rect(Center.X-Spoke, Center.Y-Spoke, Center.X+Spoke, Center.Y+Spoke);

    with Canvas do
    begin
        // Dessin de la couleur de fond
        if Color <> clNone then
        begin
            Brush.Color := Color;
            Brush.Style := bsSolid;
            Pen.Style := psClear;
            Ellipse(CircRect);
        end;

        // Dessin du disque
        Brush.Assign(Self.Brush);
        Pen.Assign(Self.Pen);
        Ellipse(CircRect);

        // Dessin des différents quartiers
        MaxAngle := FBaseAngle * Pi / 180;
        for I := 0 to Quarters.Count-1 do
        begin
            Quarter := Quarters[I];
            if Quarter.Percent = 0.0 then Continue;
            if not Quarter.Enabled then
                Graphics := Quarter.DisabledGraphics
            else if Quarter.Down then
                Graphics := Quarter.DownGraphics
            else
                Graphics := Quarter.Graphics;

            // Avancement des angles
            MinAngle := MaxAngle;
            MaxAngle := MinAngle + (Quarter.Percent * 2*Pi / 100);
            MidAngle := (MinAngle + MaxAngle) / 2;

            // Calcul des points
            MinPt := Point(Round(Spoke * Cos(MinAngle)) + Center.X, Height - Round(Spoke * Sin(MinAngle))
            - Center.Y);
            MidPt := Point(Round(Spoke * Cos(MidAngle)) + Center.X, Height - Round(Spoke * Sin(MidAngle))
            - Center.Y);
            MaxPt := Point(Round(Spoke * Cos(MaxAngle)) + Center.X, Height - Round(Spoke * Sin(MaxAngle))
            - Center.Y);

            // Dessin du quartier
            Brush.Assign(Graphics.BackgroundBrush);
            with CircRect do
                Pie(Left, Top, Right, Bottom, MinPt.X, MinPt.Y, MaxPt.X, MaxPt.Y);

            // Calcul de la position du texte et affichage du texte
            if Quarter.ShowText then
            begin
                Brush.Assign(Graphics.TextBrush);
                Font.Assign(Graphics.Font);
                TextPos := Point((Center.X+MidPt.X) div 2, (Center.Y+MidPt.Y) div 2);
                if Quarter.Text <> '' then
                    Text := Format('%s'#13#10'('%f%%)', [Quarter.Text, Quarter.Percent])
                else
                    Text := Format('%f%%', [Quarter.Percent]);
                DrawTextRect := Rect(0, 0, Width, 0);
                DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CALCRECT or DT_CENTER or DT_NOPREFIX);
            end;
        end;
    end;
end;

```

```
with TextPos, DrawTextRect do
  DrawTextRect := Rect(X - Right div 2, Y - Bottom div 2, X + Right div 2, Y + Bottom div
2);
  DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CENTER or DT_NOPREFIX);
end;
end;
end;
```

VII - Réagir aux clics sur les quartiers

Puisque nous avons décidé d'ajouter un événement **OnClick** aux **TChartQuarter**, il faut bien pouvoir intercepter les clics de souris, et détecter sur quel quartier on a cliqué.

Pour rendre la chose compatible à la *norme* Windows, nous ne déclencherons l'événement que si on a relâché la souris au-dessus du même quartier que celui sur lequel on l'a enfoncée. C'est le comportement du bouton par exemple.

VII-A - Un événement supplémentaire au niveau de TCircleChart

Puisqu'il est fort possible que l'on veuille intercepter les clics venant de tous les quartiers de la même façon, il serait utile d'avoir un événement générique au niveau de **TCircleChart**, qui indiquerait en paramètre le quartier sur lequel on a cliqué.

En fait, en plus du traditionnel paramètre **Sender**, nous utiliserons deux paramètres : un de type **integer** indiquant l'index du quartier et un de type **TChartQuarter** indiquant le quartier lui-même. Nous nommerons ce type d'événement **TQuarterClickEvent**.

```
type
  TQuarterClickEvent = procedure(Sender : TObject; Index : integer; Quarter : TChartQuarter) of
  object;
```

Nous ajouterons donc un événement **OnClickQuarter** dans **TCircleChart** :

```
private
  FOnClickQuarter : TQuarterClickEvent;
published
  property OnClickQuarter : TQuarterClickEvent read FOnClickQuarter write FOnClickQuarter;
```

VII-B - Méthodes ClickQuarter et DoClickQuarter

Si vous vous souvenez bien, nous avons, dans la méthode **Click** de **TChartQuarter**, appelé une méthode **ClickQuarter** de **TCircleChart**. Nous ne l'avons pas encore définie.

Cette méthode sera protégée et acceptera en paramètre l'index du quartier correspondant.

En outre, nous déclarerons une méthode **DoClickQuarter**, elle aussi protégée, mais virtuelle, qui acceptera en paramètre l'index du quartier et le quartier lui-même. C'est cette méthode qui se chargera de déclencher l'événement, la méthode **ClickQuarter** ne faisant que l'appeler elle.

Ce système est une façon de bien séparer les appels lors du déclenchement d'événements. Vous retrouverez souvent ce type d'arrangement dans les sources de la VCL, ou même de la JVCL par exemple.

```
protected
  procedure DoClickQuarter(Index : integer; Quarter : TChartQuarter); virtual;
  procedure ClickQuarter(Index : integer);
```

Comme dit plus haut, la méthode **DoClickQuarter** doit se charger de déclencher l'événement :

```
procedure TCircleChart.DoClickQuarter(Index : integer; Quarter : TChartQuarter);
begin
  if Assigned(FOnClickQuarter) then
    FOnClickQuarter(Self, Index, Quarter);
end;
```

La méthode **ClickQuarter**, elle, se charge simplement d'appeler **DoClickQuarter**, avec un paramètre en plus :

```
procedure TCircleChart.ClickQuarter(Index : integer);
begin
  if Index <> -1 then
    DoClickQuarter(Index, Quarters[Index]);
end;
```

VII-C - Interception des événements souris

Pour intercepter les événements souris, nous surchargerons les méthodes **MouseDown** et **MouseUp** que nous avons déjà rencontrée dans la construction du **TDropImage**.

```
protected
procedure MouseDown(Button : TMouseButton; Shift : TShiftState;
  X, Y : integer); override;
procedure MouseUp(Button : TMouseButton; Shift : TShiftState;
  X, Y : integer); override;
```

Pour retenir sur quel quartier on a enfoncé la souris, nous aurons besoin d'une variable privée **FClickedQuarter** de type **integer**. Nous l'initialiserons à -1 dans le constructeur.

```
private
  FClickedQuarter : integer;
```

L'implémentation de ces deux méthodes utilise encore une autre méthode **PointToQuarterIndex**. Cette méthode accepte un paramètre de type **TPoint** indiquant les coordonnées relatives dans le **TCircleChart** ; et elle renvoie l'index du quartier qui se trouve à cette position. Nous implémenterons cette méthode un peu plus tard.

Avec cela, l'implémentation de **MouseDown** et de **MouseUp** devient triviale :

```
procedure TCircleChart.MouseDown(Button : TMouseButton; Shift : TShiftState;
  X, Y : integer);
begin
  if Button = mbLeft then
    begin
      FClickedQuarter := PointToQuarterIndex(Point(X, Y));
      if (FClickedQuarter <> -1) and (not Quarters[FClickedQuarter].Enabled) then
        FClickedQuarter := -1;
```

```
end;
end;

procedure TCircleChart.MouseUp(Button : TMouseButton; Shift : TShiftState;
X, Y : integer);
begin
  if (Button = mbLeft) and (FClickedQuarter <> -1) then
  begin
    if PointToQuarterIndex(Point(X, Y)) = FClickedQuarter then
      Quarters[FClickedQuarter].Click;
    FClickedQuarter := -1;
  end;
end;
```

Remarquez que l'on invoque la méthode **Click** de **TChartQuarter** qui déclenchera son événement **OnClick** avant d'appeler la méthode **ClickQuarter** de **TCircleChart**, qui, à travers la méthode **DoClickQuarter**, déclenchera l'événement **OnClickQuarter**.

VII-D - Les méthodes PointToQuarterIndex et PointToQuarter

Les méthodes **PointToQuarterIndex** et **PointToQuarter** permettront de trouver un quartier avec ses coordonnées.

Nous déclarerons ces deux méthodes publiques, puisqu'elles peuvent avoir un intérêt depuis l'application.

```
public
  function PointToQuarterIndex(Point : TPoint) : integer;
  function PointToQuarter(Point : TPoint) : TChartQuarter;
```

VII-D-1 - Méthode PointToQuarter

L'implémentation de **PointToQuarter** est triviale : elle appelle juste **PointToQuarterIndex** et renvoie le quartier correspondant. Comme **PointToQuarterIndex** peut renvoyer -1 (si les coordonnées ne se trouvent sur aucun quartier, en dehors du cercle par exemple), il faut tester la valeur de retour de **PointToQuarterIndex** et renvoyer **nil** le cas échéant.

```
function TCircleChart.PointToQuarter(Point : TPoint) : TChartQuarter;
var Index : integer;
begin
  Index := PointToQuarterIndex(Point);
  if Index = -1 then Result := nil else
    Result := FQuarters[Index];
end;
```

VII-D-2 - Algorithme de la méthode PointToQuarterIndex

À nouveau, nous tombons sur une méthode dont l'implémentation n'est pas simple du tout. L'algorithme lui-même n'est pas trivial.

Nous allons donc, tout comme pour **Paint**, décrire l'algorithme avant de l'implémenter.

Contrairement à **Paint**, **PointToQuarterIndex** accepte un paramètre en entrée et renvoie une valeur en sortie. Signalons-le dans l'algorithme :

```
entree :  
  Point de type TPoint; // coordonnées informatiques  
finentree;  
sortie :  
  Result de type integer; // index du quartier  
finsortie;
```

Comme nous utiliserons souvent la valeur $2 \cdot \text{Pi}$ dans cet algo, nous déclarerons une constante **TwoPi** pour cette valeur.

```
soit TwoPi une constante <- 2*Pi;
```

Outre cela, nous aurons besoin de quelques variables :

```
soit CenterToPoint de type Single; // distance du centre au point  
soit PtCos de type Single; // cosinus de l'angle au point  
soit PtAngle de type Single; // valeur de l'angle au point  
soient MinAngle et MaxAngle de type Single;  
  // valeurs des angles de départ et d'arrivée de chaque quartier  
soit Quarter de type TChartQuarter; // quartier courant
```

Nous pouvons maintenant réellement commencer l'algorithme.

Tout d'abord, il faut transformer les coordonnées informatiques du point en coordonnées mathématiques, afin de pouvoir utiliser les fonctions trigonométriques. Pour des raisons de commodités, nous utiliserons comme origine de repère le centre du contrôle. Il faut donc soustraire la moitié des largeur et hauteur de ces coordonnées. Ensuite, il faut prendre l'opposé de l'ordonnée.

```
Point.X <- Point.X - (largeur du contrôle / 2);  
Point.Y <- Point.Y - (hauteur du contrôle / 2);  
Point.Y <- -Point.Y;
```

Ensuite, nous calculons la distance du centre au point, donc la distance de l'origine au point, soit la racine carrée de la somme des carrés de ses coordonnées.

```
PointToCenter <- racine carrée(Point.X2 + Point.Y2);
```

Si cette valeur est plus grande que **Spoke**, alors le point est situé en dehors du disque, et il n'est donc forcément sur aucun quartier.

```
si PointToCenter est strictement plus grand que Spoke :  
  Result <- -1;  
  fin du sous-programme;  
finsi;
```

Nous savons maintenant que le point est dans le disque.

Nous allons à présent calculer l'angle au point. Pour cela, commençons par déterminer son cosinus. Celui-ci peut être trouvé en divisant l'abscisse du point par sa distance au centre. En effet, considérant le cercle dont le centre est l'origine et qui contient ce point, son rayon est **CenterToPoint** et c'est donc un agrandissement du cercle trigonométrique avec un facteur **CenterToPoint**.

```
PtCos <- Point.X / CenterToPoint;
```

Nous pouvons maintenant déterminer la valeur de l'angle grâce à un arc cosinus. Cependant, si l'ordonnée est négative, alors on n'obtient pas le bon angle, mais son opposé. On règle donc ce petit défaut.

```
PtAngle <- ArcCos(PtCos);  
si Point.Y est négatif :  
  PtAngle <- -PtAngle;  
fini;
```

Ici, nous allons soustraire la valeur de **FBaseAngle** (convertie en radian) à la valeur d'angle ainsi obtenue. Il est en effet plus facile de la soustraire ici plutôt que de l'ajouter au premier **MaxAngle**, contrairement à la méthode **Paint**, et contrairement à la logique.

```
PtAngle <- PtAngle - DegreeToRadian(FBaseAngle);
```

Il est cependant plus facile de traiter un angle dont on est certain qu'il est compris entre 0 radian (inclus) et TwoPi (exclus).

Pour s'assurer que **PtAngle** est compris dans cet intervalle, nous allons effectuer une sorte de *modulo* pour nombres décimaux. Le principe est de diviser **PtAngle** par TwoPi, d'en prendre l'arrondi vers le bas (pour les puristes, vers l'infini négatif), de multiplier à nouveau cette valeur par TwoPi, et de soustraire le résultat à **PtAngle**.

```
PtAngle <- PtAngle - TwoPi * Floor(PtAngle/TwoPi);
```

Il ne reste plus qu'à itérer sur les quartiers jusqu'à en trouver un dont **PtAngle** soit compris entre **MinAngle** et **MaxAngle**. Dès qu'on l'a trouvé, on peut arrêter la méthode. Si au bout de l'itération, on n'a trouvé aucune correspondance (cela pourrait arriver si la somme des valeurs des quartiers était inférieure à 100%), on renvoie -1.

```
MaxAngle <- 0;  
pour chaque Quarter dans Quarters :  
  MinAngle <- MaxAngle;  
  MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);  
  
  si PtAngle est compris entre MinAngle et MaxAngle :  
    Result <- Quarter.Index;  
    fin du sous-programme;  
  fin;  
finpour;
```

```
Result <- -1;
```

L'algorithme est terminé. Le voici en entier :

```
entree :
  Point de type TPoint; // coordonnées informatiques
finentree;
sortie :
  Result de type integer; // index du quartier
finsortie;

soit TwoPi une constante <- 2*Pi;

soit CenterToPoint de type Single; // disance du centre au point
soit PtCos de type Single; // cosinus de l'angle au point
soit PtAngle de type Single; // valeur de l'angle au point
soient MinAngle et MaxAngle de type Single;
  // valeurs des angles de départ et d'arrivée de chaque quartier
soit Quarter de type TChartQuarter; // quartier courant

Point.X <- Point.X - (largeur du contrôle / 2);
Point.Y <- Point.Y - (hauteur du contrôle / 2);
Point.Y <- -Point.Y;

PointToCenter <- racine carrée(Point.X2 + Point.Y2);
si PointToCenter est strictement plus grand que Spoke :
  Result <- -1;
  fin du sous-programme;
finsi;

PtCos <- Point.X / CenterToPoint;
PtAngle <- ArcCos(PtCos);
si Point.Y est négatif :
  PtAngle <- -PtAngle;
finsi;
PtAngle <- PtAngle - DegreeToRadian(FBaseAngle);
PtAngle <- PtAngle - TwoPi * Floor(PtAngle/TwoPi);

MaxAngle <- 0;
pour chaque Quarter dans Quarters :
  MinAngle <- MaxAngle;
  MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);

  si PtAngle est compris entre MinAngle et MaxAngle :
    Result <- Quarter.Index;
    fin du sous-programme;
  finpour;
Result <- -1;
```

VII-D-3 - Implémentation de la méthode PointToQuarterIndex

Nous allons maintenant implémenter cet algorithme en Delphi. Cela ne sera pas difficile du tout, la traduction est ici presque littérale.

Commençons les constante et variables.

```
soit TwoPi une constante <- 2*Pi;
```

```

soit CenterToPoint de type Single; // disance du centre au point
soit PtCos de type Single; // cosinus de l'angle au point
soit PtAngle de type Single; // valeur de l'angle au point
soient MinAngle et MaxAngle de type Single;
// valeurs des angles de départ et d'arrivée de chaque quartier
soit Quarter de type TChartQuarter; // quartier courant
    
```

Voilà ce que ça donne :

```

function TCircleChart.PointToQuarterIndex(Point : TPoint) : integer;
const
    TwoPi = 2*Pi;
var CenterToPoint, PtCos : Single;
    PtAngle, MinAngle, MaxAngle : Single;
    Quarter : TChartQuarter;
begin
end;
    
```

Le début est on ne peut plus mathématique, et est donc très simple à traduire :

```

Point.X <- Point.X - (largeur du contrôle / 2);
Point.Y <- Point.Y - (hauteur du contrôle / 2);
Point.Y <- -Point.Y;

PointToCenter <- racine carrée(Point.X2 + Point.Y2);
si PointToCenter est strictement plus grand que Spoke :
    Result <- -1;
    fin du sous-programme;
fin;

PtCos <- Point.X / CenterToPoint;
PtAngle <- ArcCos(PtCos);
si Point.Y est négatif :
    PtAngle <- -PtAngle;
fin;
PtAngle <- PtAngle - DegreeToRadian(FBaseAngle);
PtAngle <- PtAngle - TwoPi * Floor(PtAngle/TwoPi);
    
```

En voici la traduction :

```

dec(Point.X, Width div 2);
dec(Point.Y, Height div 2);
Point.Y := -Point.Y;

CenterToPoint := Sqrt(Point.X*Point.X + Point.Y*Point.Y);
if CenterToPoint > Spoke then
begin
    Result := -1;
    exit;
end;

PtCos := Point.X / CenterToPoint;
PtAngle := ArcCos(PtCos);
if Point.Y < 0 then
    PtAngle := -PtAngle;
PtAngle := PtAngle - FBaseAngle * Pi / 180;
PtAngle := PtAngle - TwoPi * Floor(PtAngle/TwoPi);
    
```

Pour l'implémentation de la boucle, on utilisera **Result** comme variable de contrôle de boucle. Cependant, cela nous empêche d'utiliser une boucle **for**, puisque la variable de contrôle de ce type de boucle est souvent remplacée par le registre **ax**, la valeur de **Result** peut être indéfinie. Nous utiliserons donc une boucle **while**.

```
MaxAngle <- 0;
pour chaque Quarter dans Quarters :
  MinAngle <- MaxAngle;
  MaxAngle <- MinAngle + PercentToRadian(Quarter.Percent);

  si PtAngle est compris entre MinAngle et MaxAngle :
    Result <- Quarter.Index;
  fin du sous-programme;
finssi;
finpour;
Result <- -1;
```

L'intérieur de la boucle étant très simple, je ne m'étendrai pas dessus.

```
MaxAngle := 0;
Result := 0;
while Result < FQuarters.Count do
begin
  Quarter := Quarters[Result];

  MinAngle := MaxAngle;
  MaxAngle := MinAngle + (Quarter.Percent * TwoPi / 100);

  if (MinAngle < PtAngle) and (PtAngle < MaxAngle) then exit;

  inc(Result);
end;
Result := -1;
```

Cette traduction n'était pas aussi difficile que celle de **Paint**. Voici le code complet de la méthode **PointToQuarterIndex** :

```
function TCircleChart.PointToQuarterIndex(Point : TPoint) : integer;
const
  TwoPi = 2*Pi;
var CenterToPoint, PtCos : Single;
    PtAngle, MinAngle, MaxAngle : Single;
    Quarter : TChartQuarter;
begin
  dec(Point.X, Width div 2);
  dec(Point.Y, Height div 2);
  Point.Y := -Point.Y;

  CenterToPoint := Sqrt(Point.X*Point.X + Point.Y*Point.Y);
  if CenterToPoint > Spoke then
  begin
    Result := -1;
    exit;
  end;

  PtCos := Point.X / CenterToPoint;
  PtAngle := ArcCos(PtCos);
  if Point.Y < 0 then
    PtAngle := -PtAngle;
  PtAngle := PtAngle - FBaseAngle * Pi / 180;
```

```
PtAngle := PtAngle - TwoPi * Floor(PtAngle/TwoPi);

MaxAngle := 0;
Result := 0;
while Result < FQuarters.Count do
begin
  Quarter := Quarters[Result];

  MinAngle := MaxAngle;
  MaxAngle := MinAngle + (Quarter.Percent * TwoPi / 100);

  if (MinAngle < PtAngle) and (PtAngle < MaxAngle) then exit;

  inc(Result);
end;
Result := -1;
end;
```

VIII - Propriétés héritées

Le fait que nous ayons hérité d'une classe abstraite fait que toutes les propriétés classiques des contrôles ne sont pas présentes.

Heureusement, nous n'avons pas besoin de les implémenter : elles existent, mais cachée (protégées ou publiques selon les cas).

La classe **TControl** propose des dizaines de propriétés protégées et publiques, qu'il ne tient qu'à vous de rendre publiées.

Il est facile de rendre les propriétés protégées et publiques publiées. Il suffit de les redéclarer, avec juste le mot-clef **property** et le nom de la propriété, sans son type ni ses accès.

Vous pouvez choisir les propriétés que vous voulez rendre publiées. Dans notre exemple, nous utiliserons les propriétés suivantes. Vous remarquerez qu'il est possible de modifier les propriétés par défaut ou les spécifications de stockage. Nous utiliserons cette technique pour les propriétés **Color** (rappelez-vous que nous avons mis **clNone** dans le constructeur), **ParentColor** (puisque le fait de modifier la propriété **Color** entraîne la mise à **False** de **ParentColor**) et **AutoSize** (dont nous n'avons pas encore parlé).

```
published
  property AutoSize default True;
  property Color default clNone;
  property DragKind;
  property DragCursor;
  property DragMode;
  property ParentBiDiMode;
  property ParentColor default False;
  property ParentShowHint;
  property PopupMenu;
  property Align;
  property Anchors;
  property BiDiMode;
  property Constraints;
  property DockOrientation;
  property ShowHint;
  property Visible;

  property OnClick;
  property OnConstrainedResize;
  property OnContextPopup;
  property OnDbClick;
  property OnDragDrop;
  property OnDragOver;
  property OnEndDock;
  property OnEndDrag;
  property OnMouseActivate;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
  property OnMouseWheel;
  property OnMouseWheelDown;
  property OnMouseWheelUp;
  property OnResize;
  property OnStartDock;
  property OnStartDrag;
```

Nous allons tout de suite modifier le constructeur pour positionner **AutoSize** à **True**.

```
constructor TCircleChart.Create(AOwner : TComponent);  
begin  
  inherited;  
  FClickedQuarter := -1;  
  AutoSize := True;  
  Color := clNone;  
  FSpoke := 100;  
  FBrush := TBrush.Create;  
  FBrush.OnChange := GraphicsChange;  
  FPen := TPen.Create;  
  FPen.OnChange := GraphicsChange;  
  FQuarters := TChartQuarters.Create(Self);  
  FBaseAngle := 90;  
end;
```

IX - Effet de la propriété AutoSize

Il est temps de nous intéresser à la façon dont **AutoSize** fait effet. Comment savoir en effet comment redimensionner le contrôle, ni même quand le faire ?

Dans **TControl**, il existe une méthode **AdjustSize** virtuelle qui doit redimensionner le contrôle en fonction des diverses propriétés. Voilà la réponse à notre première question.

Pour ce qui est de la deuxième, c'est simple : lorsque **AutoSize** est positionnée à **True**, la méthode **AdjustSize** est appelée (cela se fait automatiquement). De plus, dans les *setter* des propriétés influant sur la taille, il vous suffit d'appeler vous aussi **AdjustSize**.

Notre taille sera contrôlée par la propriété **Spoke** : la largeur et la hauteur n'ont aucune raison d'être différentes du double de cette propriété.

Nous allons donc modifier la méthode **SetSpoke** pour ajouter un appel à **AdjustSize** :

```
procedure TCircleChart.SetSpoke(New : integer);
begin
  if New <= 0 then exit;
  FSpoke := New;
  if AutoSize then AdjustSize;
  Invalidate;
end;
```

Nous allons également surcharger la méthode **AdjustSize** pour faire correspondre la taille avec le rayon du disque.

```
protected
  procedure AdjustSize; override;
```

Son implémentation est plus que simple. Mais il ne faut pas oublier à veiller que l'on est pas en train de charger le composant à partir d'un flux, au risque d'interférer avec les valeurs enregistrées.

```
procedure TCircleChart.AdjustSize;
begin
  if not (csLoading in ComponentState) then
  begin
    Width := Spoke*2;
    Height := Spoke*2;
  end;
end;
```

X - Améliorer le système de menu pop-up

Le simple fait d'avoir rendu publiée la propriété **PopupMenu** a fait que l'on peut lui assigner un menu pop-up qui surgira automatiquement en cas de clic droit.

Toutefois, il est impossible, pour l'instant, à l'application de savoir sur quel quartier on a enfoncé le bouton droit, ce qui pourrait toutefois être très utile.

Nous allons donc surcharger la méthode **DoContextPopup** déclarée elle aussi dans **TControl**. Cette méthode appelle le gestionnaire d'événement **OnContextPopup**, mais permet aussi d'effectuer des actions avant que le menu soit affiché.

```
protected
  procedure DoContextPopup(MousePos : TPoint; var Handled : boolean); override;
```

Dans cette méthode, nous allons renseigner une variable privée **FPopupQuarter** de type **TChartQuarter** avec le quartier se trouvant à la position **MousePos**. Nous ajouterons une propriété publique en lecture seule pour que l'application puisse accéder à cette information.

```
private
  FPopupQuarter : TChartQuarter;
public
  property PopupQuarter : TChartQuarter read FPopupQuarter;
```

Nous initialiserons cette variable à **nil** dans le constructeur :

```
constructor TCircleChart.Create(AOwner : TComponent);
begin
  inherited;
  FClickedQuarter := -1;
  FPopupQuarter := nil;
  Color := clNone;
  FSpoke := 100;
  FBrush := TBrush.Create;
  FBrush.OnChange := GraphicsChange;
  FPen := TPen.Create;
  FPen.OnChange := GraphicsChange;
  FQuarters := TChartQuarters.Create(Self);
  FBaseAngle := 90;
  AutoSize := True;
end;
```

Voici l'implémentation de **DoContextPopup** :

```
procedure TCircleChart.DoContextPopup(MousePos : TPoint; var Handled : boolean);
begin
  FPopupQuarter := PointToQuarter(MousePos);
  inherited;
end;
```

XI - Afficher un hint selon le quartier pointé

Il reste une dernière chose à faire : nous n'avons pas encore utilisé la propriété **Hint** de **TChartQuarter**.

Le but que nous poursuivons est que dans le cas où la souris stationne au-dessus d'un quartier, ce soit le hint de ce quartier qui soit affiché en lieu et place du hint du **TCircleChart**.

Tous les cas que nous avons rencontré jusqu'à présent ont pu être réglés grâce à la VCL. Malheureusement, ce n'est pas le cas de ce nouveau problème !

Nous devons donc nous débrouiller autrement. Dans ce genre de cas, une étude des sources de la VCL pourra souvent se révéler très instructive sur la manière de procéder.

Après une étude intensive de ces sources, j'ai remarqué qu'il était possible d'implémenter cette fonctionnalité en interceptant le message **CM_HINTSHOW**.

Pour intercepter un message Windows (envoyé via **SendMessage** ou **PostMessage**), il faut écrire une méthode qui possède la directive **message**. Ce type de méthode doit accepter un unique paramètre variable (**var**) mais son type peut être quelconque ; en pratique, il se nomme **Message** et de type **TMessage** ou **TWMXXX** ou **TCMXXX**. Ces **TWMXXX** et **TCMXXX** sont des **record** de la même taille que **TMessage** mais avec des valeurs de type différents, permettant de récupérer plus facilement la signification des paramètres L et R du message. Ces méthodes sont très souvent privées.

Le message qu'intercepte la méthode est déterminé par une constante d'identificateur de message placée après la directive **message**. Par exemple : **message WM_LBUTTONDOWN**;

Nous déclarerons donc une méthode **CMHintShow** privée comme suit :

```
private
  procedure CMHintShow(var Message : TCMHintShow); message CM_HINTSHOW;
```

Cette méthode sera appelée automatiquement lorsque notre contrôle interceptera un message de type **CM_HINTSHOW**.

Dans le code d'implémentation de telles méthodes, on peut utiliser l'instruction **inherited**, bien qu'elles ne soient pas des méthodes surchargées, pour appeler la méthode correspondante (qui ne porte pas forcément le même nom ni n'accepte le même paramètre, mais qui intercepte le même message) dans la plus proche classe parente qui en possède une.

Notre méthode **CMHintShow** devra, si la position de la souris (récupérée grâce à **Message.HintInfo.CursorPos**) se trouve sur un quartier qui possède un hint, modifier le champ **Message.HintInfo.HintStr** pour refléter cette valeur.

```
procedure TCircleChart.CMHintShow(var Message : TCMHintShow);
var Quarter : TChartQuarter;
begin
  inherited;
  if Message.Result <> 0 then exit;
  Quarter := PointToQuarter(Message.HintInfo.CursorPos);
  if Assigned(Quarter) and (Quarter.Hint <> '') then
    Message.HintInfo.HintStr := Quarter.Hint;
end;
```


XII - Ajout d'actions personnalisées

Nous allons de nouveau aborder un sujet relativement complexe de la création de composants : la création d'actions personnalisées et leur utilisation dans vos composants.

Pour illustrer cela, nous créerons un type d'actions associé à chaque quartier du **TCircleChart**. Vous aurez d'ailleurs peut-être remarqué que nombre des propriétés de la classe **TChartQuarter** ressemblaient aux propriétés standards de **TAction** : c'était bien entendu dans ce but.

XII-A - Système des actions

Nous allons tout d'abord voir comment fonctionne le système des actions, comment les composants savent qu'ils doivent actualiser leurs propriétés par rapport à l'action qui leur est associée.

XII-A-1 - Les classes en présence

Le système des actions est basé sur trois types de classes : les classes d'actions, les classes des composants les utilisant, et les classes de liens d'actions.

Vous connaissez certainement les deux premiers types. C'est le troisième qui est intéressant.

Les classes de liens d'actions (devant hériter de **TBasicActionLink**, mais en pratique, héritant souvent de **TActionLink**) se trouvent, si l'on peut dire, *entre* les deux premiers types : elles effectuent la liaison des propriétés du composant et de son action. C'est une sorte d'interface entre les deux.

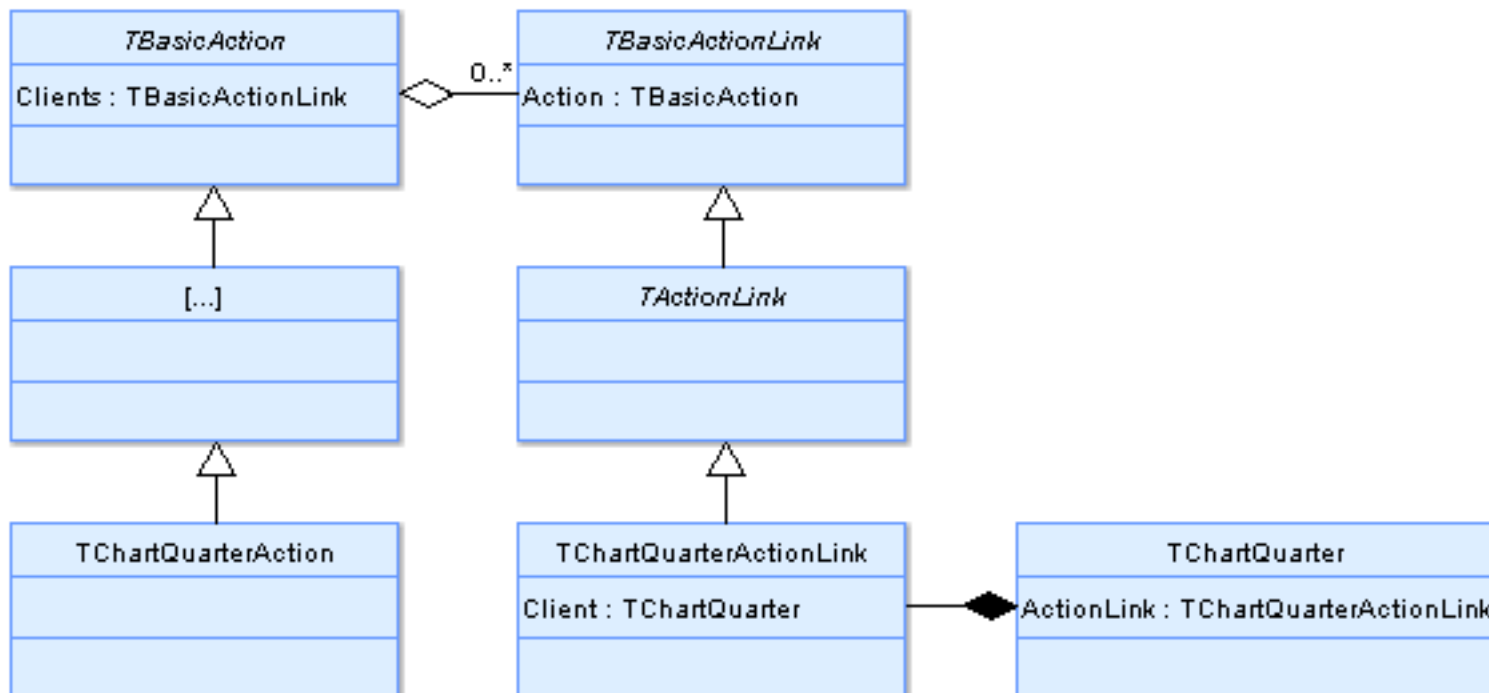
En général, chaque composant désirant mettre en oeuvre les actions possède sa classe propre de lien d'actions.

Le but des liens d'actions est que les composants n'aient pas besoin de savoir quel type d'action leur est associé, et que les actions n'aient pas besoin de savoir le type des différents composants auxquels elles sont associées.

C'est la classe de lien d'actions, connaissant le type de composant (fixe pour chaque classe de lien en général) et le type d'action, qui effectue tout le travail.

Il est également important de savoir que les types d'actions doivent hériter de **TBasicAction**, et qu'en pratique ils héritent de **TCustomAction**.

Pour éclairer les lanternes, rien ne vaut un petit diagramme UML :



XII-A-2 - La classe d'action

Premièrement, nous allons étudier la classe d'action. Comme dit plus haut, elle doit hériter de **TBasicAction**. Cette classe, qui hérite elle-même de **TComponent**, déclare des méthodes et propriétés indispensables à la gestion des actions.

Une propriété intéressante est la propriété **ActionComponent** de type **TComponent** qui indique le dernier composant qui a déclenché l'événement **OnExecute** de l'action.

Une méthode utile est la méthode **Execute**, qui déclenche l'événement **OnExecute**.

Cependant, la classe **TBasicAction** n'est guère utilisable. On hérite pour ainsi dire toujours de la classe **TContainedAction**, qui est la classe de base pour les actions devant apparaître dans une liste d'action ou un gestionnaire d'actions.

Cette classe apporte notamment une propriété **Category** qui permet de classer l'action dans la liste ou le gestionnaire.

En outre, on utilise la plupart du temps la classe **TCustomAction** qui ajoute des propriétés classiques (toutes celles se trouvent dans **TAction** en fait).

Ainsi, nous hériterons de **TCustomAction** une classe **TCustomChartQuarterAction** ; nous spécialiserons encore celle-ci avec la classe **TChartQuarterAction**, qui en publiera les propriétés.

La classe **TCustomChartQuarterAction** ajoute deux propriétés à **TCustomAction** : **Percent** et **ShowText**. On y redéfinit aussi la méthode **AssignTo** de **TPersistent** pour pouvoir l'assigner à une autre instance de **TCustomChartQuarterAction**.

La propriété **Percent** spécifie le pourcentage associé au quartier concerné. La propriété **ShowText**, quant à elle, indique s'il faut dessiner le texte (comprenant la propriété **Caption** et une indication de pourcentage) sur le quartier.

```
type
  TCustomChartQuarterAction = class(TCustomAction)
  private
    FPercent : Single;
    FShowText : boolean;
    procedure SetPercent(New : Single);
    procedure SetShowText(New : boolean);
  protected
    procedure AssignTo(Dest : TPersistent); override;
  public
    constructor Create(AOwner : TComponent); override;

    property Percent : Single read FPercent write SetPercent;
    property ShowText : boolean read FShowText write SetShowText default True;
  end;

  TChartQuarterAction = class(TCustomChartQuarterAction)
  published
    property AutoCheck;
    property Caption;
    property Percent;
    property Checked;
    property Enabled;
    property GroupIndex;
    property HelpContext;
    property HelpKeyword;
    property HelpType;
    property Hint;
    property ImageIndex;
    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property ShowText;
    property OnExecute;
    property OnHint;
    property OnUpdate;
  end;
```

L'implémentation du constructeur et de la méthode **AssignTo** est assez simple. Aussi je me contenterai de vous en donner le code :

```
constructor TCustomChartQuarterAction.Create(AOwner : TComponent);
begin
  inherited;
  FPercent := 0.0;
  FShowText := True;
end;

procedure TCustomChartQuarterAction.AssignTo(Dest : TPersistent);
begin
  if Dest is TCustomChartQuarterAction then
    with TCustomChartQuarterAction(Dest) do
      begin
        Percent := Self.Percent;
        ShowText := Self.ShowText;
      end;
  inherited;
end;
```

En revanche, les méthodes d'accès aux deux propriétés méritent leur mot d'explication. En effet, ces deux méthodes doivent avertir tous les liens d'actions (héritant de **TBasicActionLink**) qui référencent cette action. Il faut leur signaler que des changements sont survenus, et qu'il faut actualiser le composant correspondant.

Pour cela, nous bouclons sur la propriété **FClients**. Nous vérifions d'abord que la valeur retenue est bien un descendant de **TActionLink**. Ensuite, au cas où c'est aussi un descendant de **TChartQuarterActionLink** (que nous définirons plus tard), nous appelons respectivement sa méthode **SetPercent** ou **SetShowText**.

Finalement, nous appelons la méthode **Change** pour déclencher l'événement **OnChange** de l'action au cas où celui-ci serait renseigné.

```

procedure TCustomChartQuarterAction.SetPercent(New : Single);
var I : integer;
    Link : TActionLink;
begin
    if (New <> FPercent) and (New >= 0.0) then
    begin
        for I := 0 to FClients.Count-1 do
        begin
            Link := TObject(FClients.List[I]) as TActionLink;
            if Assigned(Link) and (Link is TChartQuarterActionLink) then
                TChartQuarterActionLink(Link).SetPercent(New);
            end;
            FPercent := New;
            Change;
        end;
    end;
end;

procedure TCustomChartQuarterAction.SetShowText(New : boolean);
var I : integer;
    Link : TActionLink;
begin
    if New <> FShowText then
    begin
        for I := 0 to FClients.Count-1 do
        begin
            Link := TObject(FClients.List[I]) as TActionLink;
            if Assigned(Link) and (Link is TChartQuarterActionLink) then
                TChartQuarterActionLink(Link).SetShowText(New);
            end;
            FShowText := New;
            Change;
        end;
    end;
end;
    
```

XII-A-3 - La classe de lien d'action

La clef du succès des actions, c'est la façon dont est liée un composant à son action.

Comment, en effet, un composant peut-il savoir qu'une propriété de l'action qui lui est associée a changé ? Il doit en effet en être informé afin de mettre à jour sa propriété correspondante ? Autrement dit, par exemple, comment un **TMenuItem** peut-il savoir que la propriété **Checked** de son action a changé, et qu'il doit donc modifier la sienne pour refléter les changements ?

Cette réponse est apportée par les classes de liens d'actions, qui viennent *s'intercaler* si l'on peut dire entre la classe du composant et celle de l'action.

Comme nous l'avons dit plus haut, les classes de liens d'actions sont des classes descendantes de **TBasicActionLink**, et en pratique de **TActionLink**. Notre classe de lien d'action **TChartQuarterActionLink** sera donc déclarée comme suit :

```
TChartQuarterActionLink = class(TActionLink)
```

Ces classes doivent satisfaire trois capacités :

- Référencer le composant contrôlé
- Déterminer si une propriété donnée est concernée par le lien
- Mettre à jour une propriété du composant contrôlé

XII-A-3.1 - Posséder une référence vers le composant contrôlé

Cette première capacité est implémentée au simple moyen d'une variable protégée :

```
protected  
  FClient : TChartQuarter;
```

Pour permettre aux objets extérieurs de modifier cette propriété, on surcharge la méthode **AssignClient**, déclarée initialement dans **TBasicActionLink**.

```
protected  
  procedure AssignClient(AClient : TObject); override;
```

Son implémentation est triviale :

```
procedure TChartQuarterActionLink.AssignClient(AClient : TObject);  
begin  
  FClient := AClient as TChartQuarter;  
end;
```

XII-A-3.2 - Déterminer si une propriété donnée est concernée par le lien

Afin de savoir si une propriété modifiée dans l'action doit être également modifiée dans le composant, et afin de savoir si la propriété doit être stockée en flux côté composant, il faut, pour chaque propriété de l'action, savoir si elle est liée.

Cela se fait au moyen de méthodes **IsXXXLinked**, de ce gabarit :

```
function IsAutoCheckLinked : boolean; virtual;
```

Pour les méthodes existant déjà dans **TActionLink**, on les surchargera bien entendu.

Ainsi, **TChartQuarterActionLink** possède les fonctions suivantes :

```
function IsAutoCheckLinked : boolean; virtual;  
function IsCaptionLinked : boolean; override;  
function IsPercentLinked : boolean; virtual;  
function IsCheckedLinked : boolean; override;  
function IsEnabledLinked : boolean; override;  
function IsHelpContextLinked : boolean; override;  
function IsHintLinked : boolean; override;  
function IsGroupIndexLinked : boolean; override;  
function IsImageIndexLinked : boolean; override;  
function IsShortCutLinked : boolean; override;  
function IsVisibleLinked : boolean; override;  
function IsShowTextLinked : boolean; virtual;  
function IsOnExecuteLinked : boolean; override;
```

Leurs implémentations varient selon que la propriété existe ou pas côté composant et côté action, et que leurs valeurs sont identiques de ces deux côtés.

Voici les implémentations de ces fonctions pour **TChartQuarterActionLink** :

```
function TChartQuarterActionLink.IsAutoCheckLinked : boolean;  
begin  
  Result := (Action is TCustomAction) and  
    (FClient.AutoCheck = (Action as TCustomAction).AutoCheck);  
end;  
  
function TChartQuarterActionLink.IsCaptionLinked : boolean;  
begin  
  Result := inherited IsCaptionLinked and  
    (FClient.Text = (Action as TCustomAction).Caption);  
end;  
  
function TChartQuarterActionLink.IsPercentLinked : boolean;  
begin  
  Result := (Action is TCustomChartQuarterAction) and  
    (FClient.Percent = (Action as TCustomChartQuarterAction).Percent);  
end;  
  
function TChartQuarterActionLink.IsCheckedLinked : boolean;  
begin  
  Result := inherited IsCheckedLinked and  
    (FClient.Down = (Action as TCustomAction).Checked);  
end;  
  
function TChartQuarterActionLink.IsEnabledLinked : boolean;  
begin  
  Result := inherited IsEnabledLinked and  
    (FClient.Enabled = (Action as TCustomAction).Enabled);  
end;  
  
function TChartQuarterActionLink.IsHelpContextLinked : boolean;  
begin  
  Result := False;  
end;  
  
function TChartQuarterActionLink.IsHintLinked : boolean;  
begin  
  Result := inherited IsHintLinked and  
    (FClient.Hint = (Action as TCustomAction).Hint);  
end;
```

```

function TChartQuarterActionLink.IsGroupIndexLinked : boolean;
begin
    Result := inherited IsGroupIndexLinked and
        (FClient.GroupIndex = (Action as TCustomAction).GroupIndex);
end;


function TChartQuarterActionLink.IsImageIndexLinked : boolean;
begin
    Result := False;
end;

function TChartQuarterActionLink.IsShortCutLinked : boolean;
begin
    Result := False;
end;

function TChartQuarterActionLink.IsVisibleLinked : boolean;
begin
    Result := False;
end;

function TChartQuarterActionLink.IsShowTextLinked : boolean;
begin
    Result := (Action is TCustomChartQuarterAction) and
        (FClient.ShowText = (Action as TCustomChartQuarterAction).ShowText);
end;

function TChartQuarterActionLink.IsOnExecuteLinked : boolean;
begin
    Result := inherited IsOnExecuteLinked and
        (@FClient.OnClick = @Action.OnExecute);
end;
    
```

 *Remarquez l'utilisation de l'opérateur @ parfois méconnu des programmeurs Delphi. Celui-ci permet de récupérer l'adresse d'une variable. Son utilisation est rarement nécessaire. On l'utilise ici pour pouvoir comparer deux variables de type procédural.*

XII-A-3.3 - Mettre à jour une propriété du composant contrôlé

Il ne reste plus qu'à implémenter les méthodes qui mettent à jour les propriétés côté composant en fonction des modifications effectuées côté action.

Ces fonctions doivent bien entendu d'abord vérifier que la propriété en question est liée, au moyen des méthodes **IsXXXLinked** vue précédemment.

Certaines de ces fonctions existent déjà au niveau de **TActionLink**, il faut le cas échéant les surcharger, sinon les déclarer **virtual**.

```

procedure SetAutoCheck(Value : boolean); override;
procedure SetCaption(const value : string); override;
procedure SetPercent(Value : Single); virtual;
procedure SetChecked(Value : boolean); override;
procedure SetEnabled(Value : boolean); override;
procedure SetHint(const Value : string); override;
procedure SetShowText(Value : boolean); virtual;
procedure SetOnExecute(Value : TNotifyEvent); override;
    
```

Leur implémentation se résume à la modification d'une propriété de **FClient** si l'appel correspondant à **IsXXXLinked** a renvoyé **True** :

```

procedure TChartQuarterActionLink.SetAutoCheck(Value : boolean);
begin
    if IsAutoCheckLinked then FClient.AutoCheck := Value;
end;

procedure TChartQuarterActionLink.SetCaption(const Value : string);
begin
    if IsCaptionLinked then FClient.Text := Value;
end;

procedure TChartQuarterActionLink.SetPercent(Value : Single);
begin
    if IsPercentLinked then FClient.Percent := Value;
end;

procedure TChartQuarterActionLink.SetChecked(Value : boolean);
begin
    if IsCheckedLinked then FClient.Down := Value;
end;

procedure TChartQuarterActionLink.SetEnabled(Value : boolean);
begin
    if IsEnabledLinked then FClient.Enabled := Value;
end;

procedure TChartQuarterActionLink.SetHint(const Value : string);
begin
    if IsHintLinked then FClient.Hint := Value;
end;

procedure TChartQuarterActionLink.SetShowText(Value : boolean);
begin
    if IsShowTextLinked then FClient.ShowText := Value;
end;

procedure TChartQuarterActionLink.SetOnExecute(Value : TNotifyEvent);
begin
    if IsOnExecuteLinked then FClient.OnClick := Value;
end;
    
```

XII-B - Support des actions par le composant

À présent que les classes du système d'action sont implémentées, il ne nous reste plus qu'à faire en sorte que notre composant **TChartQuarter** supporte les actions.

Contrairement à ce que l'on pourrait penser, la propriété **Action** que l'on voit en surface dans la VCL ne correspond pas à une variable privée de type **TBasicAction**. Sa valeur passe par un lien d'action.

XII-B-1 - Propriétés ActionLink et Action

Nous allons donc déclarer une variable privée de type **TChartQuarterActionLink**, avec une propriété pour y accéder de manière protégée. Nous déclarerons également une propriété publiée de type **TBasicAction**, avec un *getter* et un *setter*.

```
TChartQuarter = class(TCollectionItem)
private
    {...}
    FActionLink : TChartQuarterActionLink;
    {...}

    function GetAction : TBasicAction;
    procedure SetAction(New : TBasicAction);
    {...}
protected
    {...}
    property ActionLink : TChartQuarterActionLink read FActionLink write FActionLink;
public
    {...}
published
    property Action : TBasicAction read GetAction write SetAction;
    {...}
end;
```

Le constructeur de **TChartQuarter** initialisera **FActionLink** à **nil**, et le destructeur détruira l'instance si elle est assignée.

```
constructor TChartQuarter.Create(Collection : TCollection);
begin
    inherited;
    {...}
    FActionLink := nil;
    {...}
end;

destructor TChartQuarter.Destroy;
begin
    {...}
    if Assigned(FActionLink) then
        FActionLink.Free;
    inherited;
end;
```

La propriété **Action** de notre **TChartQuarter** se fera le reflet de la propriété **Action** de l'instance du lien d'action **FActionLink**. Le *getter* en est donc trivial :


```
function TChartQuarter.GetAction : TBasicAction;
begin
    if Assigned(FActionLink) then
        Result := FActionLink.Action
    else
        Result := nil;
end;
```

Le *setter* est légèrement plus compliqué : il doit allouer et libérer l'objet **FActionLink** en fonction de la nullité de la nouvelle valeur.

```
procedure TChartQuarter.SetAction(New : TBasicAction);
begin
    if New = nil then FreeAndNil(FActionLink) else
    begin
        if not Assigned(FActionLink) then
```

```

    FActionLink := GetActionLinkClass.Create(Self);
    FActionLink.Action := New;
end;
end;
    
```

 *Le code de **SetAction**, tel que figuré dans la partie de code ci-dessus, n'est pas définitif. Il sera modifié plusieurs fois d'ici la fin de ce tutoriel, afin de vous présenter pas à pas le pourquoi du comment de ce que nous y plaçons.*

Le lecteur attentif aura remarqué que nous créons une instance de **GetActionLinkClass**. Qu'est-ce que c'est que ça pour un type classe ? Rassurez-vous, ce n'en est pas un. Il s'agit d'une fonction qui renvoie la classe de lien d'action.

Cette fonction renvoie une valeur de type **TChartQuarterActionLinkClass**, déclarée comme suit :

```


type
    TChartQuarterActionLinkClass = class of TChartQuarterActionLink;
    
```

L'appel à **Create** se résoud donc - grâce à la magie des constructeurs virtuels - en un appel au constructeur de la classe renvoyée par **GetActionLinkClass**. Cela permet aux éventuels descendants de **TChartQuarter** d'utiliser une classe de lien d'action plus personnalisée.

La méthode **GetActionLinkClass** est déclarée protégée dans **TChartQuarter** :

```

protected
    function GetActionLinkClass : TChartQuarterActionLinkClass; dynamic;
    
```

 *La directive **dynamic** n'est pas toujours bien connue. Elle est globalement semblable à **virtual**, mais privilégie l'espace mémoire utilisé plutôt que la rapidité d'appel. Cette directive doit en principe être choisie en lieu et place de **virtual** lorsque la méthode concernée a peu de chance d'être surchargée dans les classes enfants. Si vous ne la comprenez pas bien - et c'est le cas de beaucoup -, utilisez toujours **virtual**, qui est préférable dans 95% des cas.*

Vous trouverez des explications supplémentaires dans la FAQ : [FAQ Qu'est-ce que la DMT](#) ?

Son implémentation est particulièrement simple :

```

function TChartQuarter.GetActionLinkClass : TChartQuarterActionLinkClass;
begin
    Result := TChartQuarterActionLink;
end;
    
```

XII-B-2 - Mettre à jour les propriétés lors du déclenchement d'action

Nous allons maintenant définir les méthodes qui permettront de mettre à jour toutes les propriétés d'un coup lorsque l'action elle-même sera modifiée. Rappelons que si c'est une propriété de l'action qui est modifiée, celle-ci le fait savoir à tous ses liens d'action clients, qui se chargent de modifier leurs composants clients.

Pour ce faire, déclarons une méthode **ActionChange**. Celle-ci accepte en paramètre la nouvelle action sous forme de **Sender**, ainsi qu'un paramètre booléen **CheckDefaults**, qui indique si l'on doit vérifier que les valeurs actuelles sont celles par défaut avant de les mettre à jour.


Le seul cas où une telle vérification sera nécessaire est lors du chargement des composants depuis un flux. En effet, il faut veiller à ne pas faire interférer les changements dus à l'action d'avec ceux dus à des propriétés non liées.

Cette méthode est ainsi déclarée :

```
protected
  procedure ActionChange(Sender : TObject; CheckDefaults : boolean); dynamic;
```

Et voici son implémentation :

```
procedure TChartQuarter.ActionChange(Sender : TObject; CheckDefaults : boolean);
begin
  if Sender is TCustomAction then
    with TCustomAction(Sender) do
      begin
        if not CheckDefaults or (Self.AutoCheck = False) then
          Self.AutoCheck := AutoCheck;
        if not CheckDefaults or (Self.Text = '') then
          Self.Text := Caption;
        if not CheckDefaults or (Self.Down = False) then
          Self.Down := Checked;
        if not CheckDefaults or (Self.Enabled = True) then
          Self.Enabled := Enabled;
        if not CheckDefaults or (Self.Hint = '') then
          Self.Hint := Hint;
        if not CheckDefaults or (Self.GroupIndex = 0) then
          Self.GroupIndex := GroupIndex;
        if not CheckDefaults or not Assigned(Self.OnClick) then
          Self.OnClick := OnExecute;
      end;
  if Sender is TCustomChartQuarterAction then
    with TCustomChartQuarterAction(Sender) do
      begin
        if not CheckDefaults or (Self.Percent = 0.0) then
          Self.Percent := Percent;
        if not CheckDefaults or (Self.ShowText = True) then
          Self.ShowText := ShowText;
      end;
end;
```


 *À nouveau, nous avons eu recours à une directive **dynamic**. En effet, il est peu probable que cette méthode soit effectivement surchargée dans une classe enfant de **TChartQuarter**.*

Nous allons donc modifier le *setter* de la propriété **Action** pour appeler **ActionChange** lors de l'affectation d'une nouvelle action.

```

procedure TChartQuarter.SetAction(New : TBasicAction);
begin
    if New = nil then FreeAndNil(FActionLink) else
        begin
            if not Assigned(FActionLink) then
                FActionLink := GetActionLinkClass.Create(Self);
            FActionLink.Action := New;
            ActionChange(New, csLoading in New.ComponentState);
        end;
    end;

```

 Notez que nous avons dû utiliser la propriété **ComponentState** de la nouvelle action. Les éléments de collection tels que **TChartQuarter** ne dérivant pas de **TComponent**, ils ne possèdent pas cette source d'informations cruciale.

Nous ne sommes pas encore tirés d'affaire. Rappelez-vous que nous avons déclaré une propriété protégée **ActionLink**, ce qui permet aux classes enfants de modifier directement sa propriété **Action**, sans passer par notre **setter**. Nous devons donc intercepter l'événement **OnChange** de notre **FActionLink** pour appeler également **ActionChange**.

Définissons donc une méthode privée dont la signature est celle d'un **TNotifyEvent**, qui est le type de l'événement **OnChange**.

```

private
procedure DoActionChange(Sender : TObject);

```

Son implémentation est un simple appel à **ActionChange**, avec **False** pour valeur de **CheckDefaults**.

```

procedure TChartQuarter.DoActionChange(Sender : TObject);
begin
    if Sender = FActionLink then
        ActionChange(FActionLink.Action, False);
    end;

```

Nous n'avons plus qu'à assigner ce gestionnaire à l'événement **OnChange** de tout nouveau lien d'action que nous créons :

```

procedure TChartQuarter.SetAction(New : TBasicAction);
begin
    if New = nil then FreeAndNil(FActionLink) else
        begin
            if not Assigned(FActionLink) then
                FActionLink := GetActionLinkClass.Create(Self);
            FActionLink.Action := New;
            FActionLink.OnChange := DoActionChange;
            ActionChange(New, csLoading in New.ComponentState);
        end;
    end;

```

XII-B-3 - Down et OnClick : des propriétés à forte interaction

Si vous arrivez encore à suivre, vous vous rappellerez peut-être des méthodes **SetDown** et **Click** de notre **TChartQuarter**. Celles-ci vont devoir être réétudiées ici car elles opposent une forte interaction avec les autres éléments de la collection.

En effet, la première devait positionner à **False** la propriété **Down** de tous les autres quartiers ayant la même valeur de **GroupIndex** lorsqu'elle était positionnée à **True**.

La seconde modifiant la propriété **Down**, ce qui entraîne un appel à **SetDown**, se voit répercuter l'interaction de cette méthode.

Afin de nous rafraîchir la mémoire, voici l'implémentation actuelle des deux méthodes :

```

procedure TChartQuarter.SetDown(New : boolean);
var I : integer;
    Quarter : TChartQuarter;
begin
    if New = FDown then exit;
    FDown := New;
    if (FGroupIndex > 0) and FDown then
        for I := 0 to Collection.Count-1 do
            begin
                Quarter := TChartQuarters(Collection)[I];
                if (Quarter <> Self) and
                    (Quarter.FGroupIndex = FGroupIndex) then
                    Quarter.Down := False;
            end;
        Changed(False);
    end;

procedure TChartQuarter.Click;
begin
    if Enabled then
        begin
            if AutoCheck and ((not Down) or (GroupIndex = 0)) then
                Down := not Down;
            if Assigned(FOnClick) then
                FOnClick(Self);
            FCircleChart.ClickQuarter(Index);
        end;
    end;

```

En quoi les actions viennent-elles poser problème ?

Elles posent problème car les actions, de leur côté, effectuent également cette interaction, au niveau des actions. Il faut empêcher que cette interaction soit faite des deux côtés, sous peine de risquer des interférences difficiles à gérer.

XII-B-3.1 - SetDown

Commençons par la méthode **SetDown**. Son cas est simple, il s'agit juste d'éviter de s'occuper des autres quartiers si la propriété **Checked** est réglé par le mécanisme des actions. Seul le test d'entrée dans la boucle est donc modifié :

```

if ((FActionLink = nil) or (not FActionLink.IsCheckedLinked)) and
    (FGroupIndex > 0) and FDown then

```

Le code complet et définitif est donc celui-ci :

```

procedure TChartQuarter.SetDown(New : boolean);
var I : integer;
    Quarter : TChartQuarter;
begin
    if New = FDown then exit;
    FDown := New;
    if ((FActionLink = nil) or (not FActionLink.IsCheckedLinked)) and
        (FGroupIndex > 0) and FDown then
        for I := 0 to Collection.Count-1 do
        begin
            Quarter := TChartQuarters(Collection)[I];
            if (Quarter <> Self) and
                (Quarter.FGroupIndex = FGroupIndex) then
                Quarter.Down := False;
            end;
        changed(False);
    end;

```

XII-B-3.2 - Click

Nous allons décomposer cette seconde méthode en ces trois instructions, et les faire évoluer une par une.

La première instruction a pour responsabilité d'inverser la propriété **Down** si la propriété **AutoCheck** est positionnée à **True** :

```

if AutoCheck and ((not Down) or (GroupIndex = 0)) then
    Down := not Down;

```

Ici, tout comme pour **SetDown**, il s'agit juste de vérifier que la propriété **AutoCheck** n'est pas liée au système des actions. Il faut donc juste ajouter une condition au test :

```

if AutoCheck and (not Down) or (GroupIndex = 0) and
    (not Assigned(ActionLink) or not ActionLink.IsAutoCheckLinked) then
    Down := not Down;

```

La seconde instruction invoque le gestionnaire affecté à l'événement **OnClick**.

```

if Assigned(FOnClick) then
    FOnClick(Self);

```

Si la propriété **OnClick** est liée (à **OnExecute**), il faut appeler la méthode **Execute** de l'action associée. Celle-ci se chargera d'effectuer les traitements côté action dus à la propriété **AutoCheck**.

À force de fouilles dans les sources de la VCL, j'ai fini par trouver les bons tests à effectuer. Cependant, leur logique ne m'est pas parvenue, et je ne suis donc pas en mesure de vous expliquer le pourquoi du comment de chacun des tests.

Tout ce que je peux vous donner, c'est le bon code à utiliser. À vous de l'adapter, plus tard, si vous n'avez pas exactement les mêmes besoins :

```
{ Appeler OnClick s'il est assigné et différent du OnExecute de l'action
associée. Si une action est associée, alors invoquer sa méthode Execute,
sinon, appeler OnClick }
if Assigned(FOnClick) and (Action <> nil) and (@FOnClick <> @Action.OnExecute) then
  FOnClick(Self)
else if not (csDesigning in FCircleChart.ComponentState) and (FActionLink <> nil) then
  FActionLink.Action.Execute
else if Assigned(FOnClick) then
  FOnClick(Self);
```

La troisième instruction se chargeait d'appeler le gestionnaire de click au niveau de **TCircleChart** :

```
FCircleChart.ClickQuarter(Index);
```

Au sujet de cette instruction, nous allons devoir - *horresco referens* - ne rien faire.

Le code complet et définitif de la méthode **Click** est donc le suivant :

```
procedure TChartQuarter.Click;
begin
  if Enabled then
  begin
    if AutoCheck and ((not Down) or (GroupIndex = 0)) and
      (not Assigned(ActionLink) or not ActionLink.IsAutoCheckLinked) then
      Down := not Down;
    { Appeler OnClick s'il est assigné et différent du OnExecute de l'action
associée. Si une action est associée, alors invoquer sa méthode Execute,
sinon, appeler OnClick }
    if Assigned(FOnClick) and (Action <> nil) and (@FOnClick <> @Action.OnExecute) then
      FOnClick(Self)
    else if not (csDesigning in FCircleChart.ComponentState) and (FActionLink <> nil) then
      FActionLink.Action.Execute
    else if Assigned(FOnClick) then
      FOnClick(Self);
    FCircleChart.ClickQuarter(Index);
  end;
end;
```

XII-B-4 - Ne pas enregistrer les propriétés liées dans le flux

Nous devons maintenant, non seulement pour ne pas stocker des informations inutiles dans le flux, mais aussi, dans ce cas, pour éviter des interférences gênantes, ajouter des clauses **stored** à nos propriétés.

Celles-ci devront, pour chacune des propriétés que peut contrôler une action, spécifier de stocker la propriété si et seulement si cette propriété n'est pas liée par le système d'action et si sa valeur n'est pas celle par défaut.

Nous n'apprenons ici rien de nouveau, puisque nous avons déjà étudié les **spécificateurs de stockage** dans la partie I de ce tutoriel.

Ce n'est guère compliqué. Voici les déclarations des méthodes ainsi que les déclarations modifiées des propriétés le nécessitant :

```
private
function IsAutoCheckStored : boolean;
function IsTextStored : boolean;
function IsPercentStored : boolean;
function IsDownStored : boolean;
function IsEnabledStored : boolean;
function IsGroupIndexStored : boolean;
function IsHintStored : boolean;
function IsShowTextStored : boolean;
published
property AutoCheck : boolean read FAutoCheck write FAutoCheck stored IsAutoCheckStored;
property Text : string read FText write SetText stored IsTextStored;
property Percent : Single read FPercent write SetPercent stored IsPercentStored;
property Down : boolean read FDown write SetDown stored IsDownStored;
property Enabled : boolean read FEnabled write SetEnabled stored IsEnabledStored;
property GroupIndex : integer read FGroupIndex write SetGroupIndex stored IsGroupIndexStored;
property Hint : string read FHint write FHint stored IsHintStored;
property ShowText : boolean read FShowText write SetShowText stored IsShowTextStored;
```

Voici maintenant l'implémentation de ces méthodes. Elles n'ont rien d'extraordinaire, comparé à ce que nous avons déjà vu :

```
function TChartQuarter.IsAutoCheckStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsAutoCheckLinked) and
        FAutoCheck <> False;
end;

function TChartQuarter.IsTextStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsCaptionLinked) and
        (FText <> '');
end;

function TChartQuarter.IsPercentStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsPercentLinked) and
        (FPercent <> 0.0);
end;

function TChartQuarter.IsDownStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsCheckedLinked) and
        (FDown <> False);
end;

function TChartQuarter.IsEnabledStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsEnabledLinked) and
        (FEnabled <> True);
end;

function TChartQuarter.IsGroupIndexStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsGroupIndexLinked) and
        (FGroupIndex <> 0);
end;

function TChartQuarter.IsHintStored : boolean;
```

```

begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsHintLinked) and
        (FHint <> '');
end;

function TChartQuarter.IsShowTextStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsShowTextLinked) and
        (FShowText <> True);
end;
    
```

XII-B-5 - Sécuriser la destruction des actions associées

Dans la deuxième partie de ce tutoriel, nous avons vu que lorsqu'une propriété d'un composant indiquaient un autre composant, il était nécessaire de se prémunir contre la destruction de ce second composant. La section qui en parlait s'intitulait **Sécuriser la destruction du contrôle DropControl**.

Or, nous nous trouvons ici dans la même situation : un composant (**TChartQuarter**) en référence un autre (**TBasicAction**) par l'intermédiaire d'une propriété (**Action**).

Il est donc nécessaire de sécuriser la destruction de toute action reliée à un **TChartQuarter**.

Le hic, c'est que **TChartQuarter**, comme tout autre élément de collection, n'est pas un descendant de **TComponent**, et ne possède donc pas de méthode **Notification**. On ne peut d'ailleurs pas non plus passer une instance de cette classe à la méthode **FreeNotification** de l'action concernée.

Pour remédier à ce problème, nous allons *squatter* la méthode **Notification** du **TCircleChart** associé à notre **TChartQuarter**. C'est celle-ci qui se chargera de boucler sur ses quartiers et de nullifier les propriétés **Action** qui référence l'action en cours de destruction :

```

TCircleChart = class(TGraphicControl)
protected
    procedure Notification(AComponent : TComponent; Operation : TOperation); override;
end;
    
```

Son implémentation est on ne peut plus simple :

```


procedure TCircleChart.Notification(AComponent : TComponent; Operation : TOperation);
var I : integer;
begin
    inherited;
    if Operation = opRemove then for I := 0 to Quarters.Count-1 do
        if AComponent = Quarters[I].Action then
            Quarters[I].Action := nil;
    end;
end;
    
```

Il ne reste plus qu'à ajouter l'objet graphique à la liste de notification de toute nouvelle action référencée. Nous modifions donc pour la troisième et dernière fois la méthode **SetAction** de **TChartQuarter**.

```

procedure TChartQuarter.SetAction(New : TBasicAction);
begin
    
```

```
if New = nil then FreeAndNil(FActionLink) else
begin
  if not Assigned(FActionLink) then
    FActionLink := GetActionLinkClass.Create(Self);
  FActionLink.Action := New;
  FActionLink.OnChange := DoActionChange;
  ActionChange(New, csLoading in New.ComponentState);
  New.FreeNotification(FCircleChart);
end;
end;
```

 Nous n'utilisons pas ici la méthode **RemoveFreeNotification** en cas de déréférencement d'une action. En effet, il se pourrait (même si c'est illogique) que plusieurs quartiers du même graphique soient attachés à la même action.

Nous avons donc, enfin, terminé notre composant graphique.

XIII - Code complet du composant TCircleChart

CircChart.pas

```

unit CircChart;

interface

uses
    Windows, Messages, Forms, Classes, SysUtils, Graphics, Controls, Math,
    ActnList;

type
    TChartQuarter = class;
    TCircleChart = class;

    TChartQuarterActionLink = class(TActionLink)
    protected
        FClient : TChartQuarter;
        procedure AssignClient(AClient : TObject); override;

        function IsAutoCheckLinked : boolean; virtual;
        function IsCaptionLinked : boolean; override;
        function IsPercentLinked : boolean; virtual;
        function IsCheckedLinked : boolean; override;
        function IsEnabledLinked : boolean; override;
        function IsHelpContextLinked : boolean; override;
        function IsHintLinked : boolean; override;
        function IsGroupIndexLinked : boolean; override;
        function IsImageIndexLinked : boolean; override;
        function IsShortCutLinked : boolean; override;
        function IsVisibleLinked : boolean; override;
        function IsShowTextLinked : boolean; virtual;
        function IsOnExecuteLinked : boolean; override;

        procedure SetAutoCheck(Value : boolean); override;
        procedure SetCaption(const value : string); override;
        procedure SetPercent(Value : Single); virtual;
        procedure SetChecked(Value : boolean); override;
        procedure SetEnabled(Value : boolean); override;
        procedure SetHint(const Value : string); override;
        procedure SetShowText(Value : boolean); virtual;
        procedure SetOnExecute(Value : TNotifyEvent); override;
    end;

    TChartQuarterActionLinkClass = class of TChartQuarterActionLink;

    TCustomChartQuarterAction = class(TCustomAction)
    private
        FPercent : Single;
        FShowText : boolean;
        procedure SetPercent(New : Single);
        procedure SetShowText(New : boolean);
    protected
        procedure AssignTo(Dest : TPersistent); override;
    public
        constructor Create(AOwner : TComponent); override;

        property Percent : Single read FPercent write SetPercent;
        property ShowText : boolean read FShowText write SetShowText default True;
    end;

    TChartQuarterAction = class(TCustomChartQuarterAction)
    published
        property AutoCheck;
        property Caption;
    
```

CircChart.pas

```

property Percent;
property Checked;
property Enabled;
property GroupIndex;
property HelpContext;
property HelpKeyword;
property HelpType;
property Hint;
property ImageIndex;
property ShortCut;
property SecondaryShortCuts;
property Visible;
property ShowText;
property OnExecute;
property OnHint;
property OnUpdate;
end;

TChartQuarterGraphics = class(TPersistent)
private
    FBackgroundBrush : TBrush;
    FTextBrush : TBrush;
    FFont : TFont;
    procedure SetBackgroundBrush(New : TBrush);
    procedure SetTextBrush(New : TBrush);
    procedure SetFont(New : TFont);
public
    constructor Create(AOnChange : TNotifyEvent = nil);
    destructor Destroy; override;
    procedure Assign(Source : TPersistent); override;
published
    property BackgroundBrush : TBrush read FBackgroundBrush write SetBackgroundBrush;
    property TextBrush : TBrush read FTextBrush write SetTextBrush;
    property Font : TFont read FFont write SetFont;
end;

TChartQuarter = class(TCollectionItem)
private
    FCircleChart : TCircleChart;
    FActionLink : TChartQuarterActionLink;

    FAutoCheck : boolean;
    FText : string;
    FPercent : Single;
    FDown : boolean;
    FEnabled : boolean;
    FGroupIndex : integer;
    FHint : string;
    FShowText : boolean;
    FGraphics : TChartQuarterGraphics;
    FDownGraphics : TChartQuarterGraphics;
    FDisabledGraphics : TChartQuarterGraphics;

    FOnClick : TNotifyEvent;

    function GetAction : TBasicAction;
    procedure SetAction(New : TBasicAction);
    procedure SetText(const New : string);
    procedure SetPercent(New : Single);
    procedure SetDown(New : boolean);
    procedure SetEnabled(New : boolean);
    procedure SetGroupIndex(New : integer);
    procedure SetGraphics(New : TChartQuarterGraphics);
    procedure SetDownGraphics(New : TChartQuarterGraphics);
    procedure SetDisabledGraphics(New : TChartQuarterGraphics);
    procedure SetShowText(New : boolean);
    procedure GraphicsChange(Sender : TObject);

```

CircChart.pas

```

procedure DoActionChange(Sender : TObject);

function IsAutoCheckStored : boolean;
function IsTextStored : boolean;
function IsPercentStored : boolean;
function IsDownStored : boolean;
function IsEnabledStored : boolean;
function IsGroupIndexStored : boolean;
function IsHintStored : boolean;
function IsShowTextStored : boolean;
protected
function GetDisplayName : string; override;
procedure ActionChange(Sender : TObject; CheckDefaults : boolean); dynamic;
function GetActionLinkClass : TChartQuarterActionLinkClass; dynamic;

property ActionLink : TChartQuarterActionLink read FActionLink write FActionLink;
public
constructor Create(Collection : TCollection); override;
destructor Destroy; override;

procedure Assign(Source : TPersistent); override;

procedure Click;
published
property Action : TBasicAction read GetAction write SetAction;
property AutoCheck : boolean read FAutoCheck write FAutoCheck stored IsAutoCheckStored;
property Text : string read FText write SetText stored IsTextStored;
property Percent : Single read FPercent write SetPercent stored IsPercentStored;
property Down : boolean read FDown write SetDown stored IsDownStored;
property Enabled : boolean read FEnabled write SetEnabled stored IsEnabledStored;
property GroupIndex : integer read FGroupIndex write SetGroupIndex stored IsGroupIndexStored;
property Hint : string read FHint write FHint stored IsHintStored;
property ShowText : boolean read FShowText write SetShowText stored IsShowTextStored;
property Graphics : TChartQuarterGraphics read FGraphics write SetGraphics;
property DownGraphics : TChartQuarterGraphics read FDownGraphics write SetDownGraphics;
property DisabledGraphics : TChartQuarterGraphics read FDisabledGraphics write
SetDisabledGraphics;

property OnClick : TNotifyEvent read FOnClick write FOnClick;
end;

TChartQuarters = class(TCollection)
private
    FCircleChart : TCircleChart;
function GetItem(Index : integer) : TChartQuarter;
procedure SetItem(Index : integer; Value : TChartQuarter);
protected
function GetOwner : TPersistent; override;
procedure Update(Item : TCollectionItem); override;
public
constructor Create(ACircleChart : TCircleChart);

function Add : TChartQuarter;
function AddItem(Item : TChartQuarter; Index : integer) : TChartQuarter;
function Insert(Index : integer) : TChartQuarter;

property Items[Index : integer] : TChartQuarter read GetItem write SetItem; default;
end;

TQuarterClickEvent = procedure(Sender : TObject; Index : integer; Quarter : TChartQuarter) of
object;

TCircleChart = class(TGraphicControl)
private
    FClickedQuarter : integer;
    FPopupQuarter : TChartQuarter;

```

CircChart.pas

```

FSpoke : integer;
FBrush : TBrush;
FPen : TPen;
FQuarters : TChartQuarters;
FBaseAngle : integer;

FOnClickQuarter : TQuarterClickEvent;

procedure SetSpoke(New : integer);
procedure SetBrush(New : TBrush);
procedure SetPen(New : TPen);
procedure SetQuarters(New : TChartQuarters);
procedure SetBaseAngle(New : integer);
procedure GraphicsChange(Sender : TObject);

procedure CMHintShow(var Message : TCMHintShow); message CM_HINTSHOW;
protected
procedure Notification(AComponent : TComponent; Operation : TOperation); override;
procedure AdjustSize; override;
procedure Paint; override;

procedure DoContextPopup(MousePos : TPoint; var Handled : boolean); override;
procedure MouseDown(Button : TMouseButton; Shift : TShiftState;
    X, Y : integer); override;
procedure MouseUp(Button : TMouseButton; Shift : TShiftState;
    X, Y : integer); override;
procedure DoClickQuarter(Index : integer; Quarter : TChartQuarter); virtual;

procedure ClickQuarter(Index : integer);
public
constructor Create(AOwner : TComponent); override;
destructor Destroy; override;

function PointToQuarterIndex(Point : TPoint) : integer;
function PointToQuarter(Point : TPoint) : TChartQuarter;

property PopupQuarter : TChartQuarter read FPopupQuarter;
published
property AutoSize default True;
property Color default clNone;
property DragKind;
property DragCursor;
property DragMode;
property ParentBiDiMode;
property ParentColor default False;
property ParentShowHint;
property PopupMenu;
property Align;
property Anchors;
property BiDiMode;
property Constraints;
property DockOrientation;
property ShowHint;
property Visible;

property Spoke : integer read FSpoke write SetSpoke default 100;
property Brush : TBrush read FBrush write SetBrush;
property Pen : TPen read FPen write SetPen;
property Quarters : TChartQuarters read FQuarters write SetQuarters;
property BaseAngle : integer read FBaseAngle write SetBaseAngle default 90;

property OnClick;
property OnConstrainedResize;
property OnContextPopup;
property OnDblClick;
property OnDragDrop;
    
```

CircChart.pas

```

property OnDragOver;
property OnEndDock;
property OnEndDrag;
property OnMouseActivate;
property OnMouseDown;
property OnMouseMove;
property OnMouseUp;
property OnMouseWheel;
property OnMouseWheelDown;
property OnMouseWheelUp;
property OnResize;
property OnStartDock;
property OnStartDrag;
    
```

```

property OnClickQuarter : TQuarterClickEvent read FOnClickQuarter write FOnClickQuarter;
end;
    
```

implementation

```

////////////////////////////////////
/// Classe TChartQuarterActionLink ///
////////////////////////////////////
    
```

```

procedure TChartQuarterActionLink.AssignClient(AClient : TObject);
begin
    FClient := AClient as TChartQuarter;
end;
    
```

```

function TChartQuarterActionLink.IsAutoCheckLinked : boolean;
begin
    Result := (Action is TCustomAction) and
        (FClient.AutoCheck = (Action as TCustomAction).AutoCheck);
end;
    
```

```

function TChartQuarterActionLink.IsCaptionLinked : boolean;
begin
    Result := inherited IsCaptionLinked and
        (FClient.Text = (Action as TCustomAction).Caption);
end;
    
```

```

function TChartQuarterActionLink.IsPercentLinked : boolean;
begin
    Result := (Action is TCustomChartQuarterAction) and
        (FClient.Percent = (Action as TCustomChartQuarterAction).Percent);
end;
    
```

```

function TChartQuarterActionLink.IsCheckedLinked : boolean;
begin
    Result := inherited IsCheckedLinked and
        (FClient.Down = (Action as TCustomAction).Checked);
end;
    
```

```

function TChartQuarterActionLink.IsEnabledLinked : boolean;
begin
    Result := inherited IsEnabledLinked and
        (FClient.Enabled = (Action as TCustomAction).Enabled);
end;
    
```

```

function TChartQuarterActionLink.IsHelpContextLinked : boolean;
begin
    Result := False;
end;
    
```

```

function TChartQuarterActionLink.IsHintLinked : boolean;
begin
    Result := inherited IsHintLinked and
        (FClient.Hint = (Action as TCustomAction).Hint);
    
```

CircChart.pas

```
end;

function TChartQuarterActionLink.IsGroupIndexLinked : boolean;
begin
  Result := inherited IsGroupIndexLinked and
    (FClient.GroupIndex = (Action as TCustomAction).GroupIndex);
end;

function TChartQuarterActionLink.IsImageIndexLinked : boolean;
begin
  Result := False;
end;

function TChartQuarterActionLink.IsShortCutLinked : boolean;
begin
  Result := False;
end;

function TChartQuarterActionLink.IsVisibleLinked : boolean;
begin
  Result := False;
end;

function TChartQuarterActionLink.IsShowTextLinked : boolean;
begin
  Result := (Action is TCustomChartQuarterAction) and
    (FClient.ShowText = (Action as TCustomChartQuarterAction).ShowText);
end;

function TChartQuarterActionLink.IsOnExecuteLinked : boolean;
begin
  Result := inherited IsOnExecuteLinked and
    (@FClient.OnClick = @Action.OnExecute);
end;

procedure TChartQuarterActionLink.SetAutoCheck(Value : boolean);
begin
  if IsAutoCheckLinked then FClient.AutoCheck := Value;
end;

procedure TChartQuarterActionLink.SetCaption(const Value : string);
begin
  if IsCaptionLinked then FClient.Text := Value;
end;

procedure TChartQuarterActionLink.SetPercent(Value : Single);
begin
  if IsPercentLinked then FClient.Percent := Value;
end;

procedure TChartQuarterActionLink.SetChecked(Value : boolean);
begin
  if IsCheckedLinked then FClient.Down := Value;
end;

procedure TChartQuarterActionLink.SetEnabled(Value : boolean);
begin
  if IsEnabledLinked then FClient.Enabled := Value;
end;

procedure TChartQuarterActionLink.SetHint(const Value : string);
begin
  if IsHintLinked then FClient.Hint := Value;
end;

procedure TChartQuarterActionLink.SetShowText(Value : boolean);
begin
```

CircChart.pas

```

    if IsShowTextLinked then FClient.ShowText := Value;
end;

procedure TChartQuarterActionLink.SetOnExecute(Value : TNotifyEvent);
begin
    if IsOnExecuteLinked then FClient.OnClick := Value;
end;

////////////////////////////////////////////////////////////////////////////////
/// Classe TCustomChartQuarterAction ///
////////////////////////////////////////////////////////////////////////////////

constructor TCustomChartQuarterAction.Create(AOwner : TComponent);
begin
    inherited;
    FPercent := 0.0;
    FShowText := True;
end;

procedure TCustomChartQuarterAction.SetPercent(New : Single);
var I : integer;
    Link : TActionLink;
begin
    if (New <> FPercent) and (New >= 0.0) then
    begin
        for I := 0 to FClients.Count-1 do
        begin
            Link := TObject(FClients.List[I]) as TActionLink;
            if Assigned(Link) and (Link is TChartQuarterActionLink) then
                TChartQuarterActionLink(Link).SetPercent(New);
        end;
        FPercent := New;
        Change;
    end;
end;

procedure TCustomChartQuarterAction.SetShowText(New : boolean);
var I : integer;
    Link : TActionLink;
begin
    if New <> FShowText then
    begin
        for I := 0 to FClients.Count-1 do
        begin
            Link := TObject(FClients.List[I]) as TActionLink;
            if Assigned(Link) and (Link is TChartQuarterActionLink) then
                TChartQuarterActionLink(Link).SetShowText(New);
        end;
        FShowText := New;
        Change;
    end;
end;

procedure TCustomChartQuarterAction.AssignTo(Dest : TPersistent);
begin
    if Dest is TCustomChartQuarterAction then
        with TCustomChartQuarterAction(Dest) do
            begin
                Percent := Self.Percent;
                ShowText := Self.ShowText;
            end;
    inherited;
end;

////////////////////////////////////////////////////////////////////////////////
/// Classe TChartQuarterGraphics ///
////////////////////////////////////////////////////////////////////////////////

```

CircChart.pas

```

constructor TChartQuarterGraphics.Create(AOnChange : TNotifyEvent = nil);
begin
    inherited Create;
    FBackgroundBrush := TBrush.Create;
    FBackgroundBrush.OnChange := AOnChange;
    FTextBrush := TBrush.Create;
    FTextBrush.OnChange := AOnChange;
    FFont := TFont.Create;
    FFont.OnChange := AOnChange;
end;

destructor TChartQuarterGraphics.Destroy;
begin
    FFont.Free;
    FTextBrush.Free;
    FBackgroundBrush.Free;
    inherited Destroy;
end;

procedure TChartQuarterGraphics.SetBackgroundBrush(New : TBrush);
begin
    FBackgroundBrush.Assign(New);
end;

procedure TChartQuarterGraphics.SetTextBrush(New : TBrush);
begin
    FTextBrush.Assign(New);
end;

procedure TChartQuarterGraphics.SetFont(New : TFont);
begin
    FFont.Assign(New);
end;

procedure TChartQuarterGraphics.Assign(Source : TPersistent);
begin
    if Source is TChartQuarterGraphics then
    begin
        with TChartQuarterGraphics(Source) do
        begin
            Self.FBackgroundBrush.Assign(FBackgroundBrush);
            Self.FTextBrush.Assign(FTextBrush);
            Self.FFont.Assign(FFont);
        end;
    end else inherited;
end;

////////////////////////////////////
/// Classe TChartQuarter ///
////////////////////////////////////

constructor TChartQuarter.Create(Collection : TCollection);
begin
    inherited;
    FCircleChart := TChartQuarters(Collection).FCircleChart;
    FActionLink := nil;
    FText := '';
    FPercent := 0.0;
    FDown := False;
    FEnabled := True;
    FGroupIndex := 0;
    FHint := '';
    FShowText := True;
    FGraphics := TChartQuarterGraphics.Create(GraphicsChange);
    FDownGraphics := TChartQuarterGraphics.Create(GraphicsChange);
    FDisabledGraphics := TChartQuarterGraphics.Create(GraphicsChange);

```

CircChart.pas

```

end;

destructor TChartQuarter.Destroy;
begin
    FDisabledGraphics.Free;
    FDownGraphics.Free;
    FGraphics.Free;
    if Assigned(FActionLink) then
        FActionLink.Free;
    inherited;
end;

function TChartQuarter.GetAction : TBasicAction;
begin
    if Assigned(FActionLink) then
        Result := FActionLink.Action
    else
        Result := nil;
end;

procedure TChartQuarter.SetAction(New : TBasicAction);
begin
    if New = nil then FreeAndNil(FActionLink) else
    begin
        if not Assigned(FActionLink) then
            FActionLink := GetActionLinkClass.Create(Self);
        FActionLink.Action := New;
        FActionLink.OnChange := DoActionChange;
        ActionChange(New, csLoading in New.ComponentState);
        New.FreeNotification(FCircleChart);
    end;
end;

procedure TChartQuarter.SetText(const New : string);
begin
    FText := New;
    Changed(False);
end;

procedure TChartQuarter.SetPercent(New : Single);
begin
    if New < 0.0 then exit;
    FPercent := New;
    Changed(True);
end;

procedure TChartQuarter.SetDown(New : boolean);
var I : integer;
    Quarter : TChartQuarter;
begin
    if New = FDown then exit;
    FDown := New;
    if ((FActionLink = nil) or (not FActionLink.IsCheckedLinked)) and
        (FGroupIndex > 0) and FDown then
        for I := 0 to Collection.Count-1 do
            begin
                Quarter := TChartQuarters(Collection)[I];
                if (Quarter <> Self) and
                    (Quarter.FGroupIndex = FGroupIndex) then
                    Quarter.Down := False;
            end;
        Changed(False);
    end;
end;

procedure TChartQuarter.SetEnabled(New : boolean);
begin
    FEnabled := New;

```

CircChart.pas

```

    Changed(False);
end;

procedure TChartQuarter.SetGroupIndex(New : integer);
begin
    FGroupIndex := New;
end;

procedure TChartQuarter.SetGraphics(New : TChartQuarterGraphics);
begin
    FGraphics.Assign(New);
end;

procedure TChartQuarter.SetDownGraphics(New : TChartQuarterGraphics);
begin
    FDownGraphics.Assign(New);
end;

procedure TChartQuarter.SetDisabledGraphics(New : TChartQuarterGraphics);
begin
    FDisabledGraphics.Assign(New);
end;

procedure TChartQuarter.SetShowText(New : boolean);
begin
    FShowText := New;
    Changed(False);
end;

procedure TChartQuarter.GraphicsChange(Sender : TObject);
begin
    Changed(False);
end;

procedure TChartQuarter.DoActionChange(Sender : TObject);
begin
    if Sender = FActionLink then
        ActionChange(FActionLink.Action, False);
end;

function TChartQuarter.IsAutoCheckStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsAutoCheckLinked) and
        FAutoCheck <> False;
end;

function TChartQuarter.IsTextStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsCaptionLinked) and
        (FText <> '');
end;

function TChartQuarter.IsPercentStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsPercentLinked) and
        (FPercent <> 0.0);
end;

function TChartQuarter.IsDownStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsCheckedLinked) and
        (FDown <> False);
end;

function TChartQuarter.IsEnabledStored : boolean;
begin
    Result := (not Assigned(FActionLink) or not FActionLink.IsEnabledLinked) and

```

CircChart.pas

```

        (FEnabled <> True);
    end;

    function TChartQuarter.IsGroupIndexStored : boolean;
    begin
        Result := (not Assigned(FActionLink) or not FActionLink.IsGroupIndexLinked) and
            (FGroupIndex <> 0);
    end;

    function TChartQuarter.IsHintStored : boolean;
    begin
        Result := (not Assigned(FActionLink) or not FActionLink.IsHintLinked) and
            (FHint <> '');
    end;

    function TChartQuarter.IsShowTextStored : boolean;
    begin
        Result := (not Assigned(FActionLink) or not FActionLink.IsShowTextLinked) and
            (FShowText <> True);
    end;

    function TChartQuarter.GetDisplayName : string;
    begin
        if Text <> '' then
            Result := Format('%s (%f%%)', [FText, FPercent])
        else
            Result := Format('%f%%', [FPercent]);
    end;

    procedure TChartQuarter.ActionChange(Sender : TObject; CheckDefaults : boolean);
    begin
        if Sender is TCustomAction then
            with TCustomAction(Sender) do
                begin
                    if not CheckDefaults or (Self.AutoCheck = False) then
                        Self.AutoCheck := AutoCheck;
                    if not CheckDefaults or (Self.Text = '') then
                        Self.Text := Caption;
                    if not CheckDefaults or (Self.Down = False) then
                        Self.Down := Checked;
                    if not CheckDefaults or (Self.Enabled = True) then
                        Self.Enabled := Enabled;
                    if not CheckDefaults or (Self.Hint = '') then
                        Self.Hint := Hint;
                    if not CheckDefaults or (Self.GroupIndex = 0) then
                        Self.GroupIndex := GroupIndex;
                    if not CheckDefaults or not Assigned(Self.OnClick) then
                        Self.OnClick := OnExecute;
                    end;
                end;
        if Sender is TCustomChartQuarterAction then
            with TCustomChartQuarterAction(Sender) do
                begin
                    if not CheckDefaults or (Self.Percent = 0.0) then
                        Self.Percent := Percent;
                    if not CheckDefaults or (Self.ShowText = True) then
                        Self.ShowText := ShowText;
                    end;
                end;
        end;

    function TChartQuarter.GetActionLinkClass : TChartQuarterActionLinkClass;
    begin
        Result := TChartQuarterActionLink;
    end;

    procedure TChartQuarter.Assign(Source : TPersistent);
    var ChartQuarterSource : TChartQuarter;
    begin

```

CircChart.pas

```

if Source is TChartQuarter then
begin
    ChartQuarterSource := TChartQuarter(Source);
    Action := ChartQuarterSource.Action;
    FText := ChartQuarterSource.FText;
    FPercent := ChartQuarterSource.FPercent;
    FDown := ChartQuarterSource.FDown;
    FEnabled := ChartQuarterSource.FEnabled;
    FGroupIndex := ChartQuarterSource.FGroupIndex;
    FHint := ChartQuarterSource.FHint;
    FShowText := ChartQuarterSource.FShowText;
    FGraphics.Assign(ChartQuarterSource.FGraphics);
    FDownGraphics.Assign(ChartQuarterSource.FDownGraphics);
    FDisabledGraphics.Assign(ChartQuarterSource.FDisabledGraphics);
    Changed(True);
end else inherited;
end;

procedure TChartQuarter.Click;
begin
    if Enabled then
    begin
        if AutoCheck and ((not Down) or (GroupIndex = 0)) and
            (not Assigned(ActionLink) or not ActionLink.IsAutoCheckLinked) then
            Down := not Down;
        { Appeler OnClick s'il est assigné et différent du OnExecute de l'action
          associée. Si une action est associée, alors invoquer sa méthode Execute,
          sinon, appeler OnClick }
        if Assigned(FOnClick) and (Action <> nil) and (@FOnClick <> @Action.OnExecute) then
            FOnClick(Self)
        else if not (csDesigning in FCircleChart.ComponentState) and (FActionLink <> nil) then
            FActionLink.Action.Execute
        else if Assigned(FOnClick) then
            FOnClick(Self);
        FCircleChart.ClickQuarter(Index);
    end;
end;

////////////////////////////////////
/// Classe TChartQuarters ///
////////////////////////////////////

constructor TChartQuarters.Create(ACircleChart : TCircleChart);
begin
    inherited Create(TChartQuarter);
    FCircleChart := ACircleChart;
end;

function TChartQuarters.GetItem(Index : integer) : TChartQuarter;
begin
    Result := TChartQuarter(inherited GetItem(Index));
end;

procedure TChartQuarters.SetItem(Index : integer; Value : TChartQuarter);
begin
    inherited SetItem(Index, Value);
end;

function TChartQuarters.GetOwner : TPersistent;
begin
    Result := FCircleChart;
end;

procedure TChartQuarters.Update(Item : TCollectionItem);
begin
    FCircleChart.Invalidate;
end;
    
```

CircChart.pas

```

function TChartQuarters.Add : TChartQuarter;
begin
    Result := TChartQuarter(inherited Add);
end;

function TChartQuarters.AddItem(Item : TChartQuarter; Index : integer) : TChartQuarter;
begin
    if Item = nil then
        Result := Add
    else
        Result := Item;
    if Assigned(Result) then
        begin
            Result.Collection := Self;
            if Index < 0 then
                Index := Count - 1;
            Result.Index := Index;
        end;
    end;

function TChartQuarters.Insert(Index : integer) : TChartQuarter;
begin
    Result := AddItem(nil, Index);
end;

////////////////////////////////////
/// Classe TCircleChart ///
////////////////////////////////////

constructor TCircleChart.Create(AOwner : TComponent);
begin
    inherited;
    FClickedQuarter := -1;
    FPopupQuarter := nil;
    Color := clNone;
    FSpoke := 100;
    FBrush := TBrush.Create;
    FBrush.OnChange := GraphicsChange;
    FPen := TPen.Create;
    FPen.OnChange := GraphicsChange;
    FQuarters := TChartQuarters.Create(Self);
    FBaseAngle := 90;
    AutoSize := True;
end;

destructor TCircleChart.Destroy;
begin
    FQuarters.Free;
    FPen.Free;
    FBrush.Free;
    inherited;
end;

procedure TCircleChart.SetSpoke(New : integer);
begin
    if New <= 0 then exit;
    FSpoke := New;
    if AutoSize then AdjustSize;
    Invalidate;
end;

procedure TCircleChart.SetBrush(New : TBrush);
begin
    FBrush.Assign(New);
end;
    
```

CircChart.pas

```

procedure TCircleChart.SetPen(New : TPen);
begin
    FPen.Assign(New);
end;

procedure TCircleChart.SetQuarters(New : TChartQuarters);
begin
    FQuarters.Assign(New);
end;

procedure TCircleChart.SetBaseAngle(New : integer);
begin
    FBaseAngle := New mod 360;
    if FBaseAngle < 0 then inc(FBaseAngle, 360);
    Invalidate;
end;

procedure TCircleChart.GraphicsChange(Sender : TObject);
begin
    Invalidate;
end;

procedure TCircleChart.CMHintShow(var Message : TCMHintShow);
var Quarter : TChartQuarter;
begin
    inherited;
    if Message.Result <> 0 then exit;
    Quarter := PointToQuarter(Message.HintInfo.CursorPos);
    if Assigned(Quarter) and (Quarter.Hint <> '') then
        Message.HintInfo.HintStr := Quarter.Hint;
end;

procedure TCircleChart.Notification(AComponent : TComponent; Operation : TOperation);
var I : integer;
begin
    inherited;
    if Operation = opRemove then for I := 0 to Quarters.Count-1 do
        if AComponent = Quarters[I].Action then
            Quarters[I].Action := nil;
end;

procedure TCircleChart.AdjustSize;
begin
    if not (csLoading in ComponentState) then
        begin
            Width := Spoke*2;
            Height := Spoke*2;
        end;
end;

procedure TCircleChart.Paint;
var Center : TPoint; // Centre du cercle
    CircRect : TRect; // Carré circonscrit au cercle
    I : integer;
    Quarter : TChartQuarter;
    Graphics : TChartQuarterGraphics;
    MinAngle, MidAngle, MaxAngle : Single;
    MinPt, MidPt, MaxPt, TextPos : TPoint;
    Text : string;
    DrawTextRect : TRect;
begin
    // Calcul des données concernant le disque
    Center := Point(Width div 2, Height div 2);
    CircRect := Rect(Center.X-Spoke, Center.Y-Spoke, Center.X+Spoke, Center.Y+Spoke);

    with Canvas do
        begin

```

CircChart.pas

```

// Dessin de la couleur de fond
if Color <> clNone then
begin
    Brush.Color := Color;
    Brush.Style := bsSolid;
    Pen.Style := psClear;
    Ellipse(CircRect);
end;

// Dessin du disque
Brush.Assign(Self.Brush);
Pen.Assign(Self.Pen);
Ellipse(CircRect);

// Dessin des différents quartiers
MaxAngle := FBaseAngle * Pi / 180;
for I := 0 to Quarters.Count-1 do
begin
    Quarter := Quarters[I];
    if Quarter.Percent = 0.0 then Continue;
    if not Quarter.Enabled then
        Graphics := Quarter.DisabledGraphics
    else if Quarter.Down then
        Graphics := Quarter.DownGraphics
    else
        Graphics := Quarter.Graphics;

    // Avancement des angles
    MinAngle := MaxAngle;
    MaxAngle := MinAngle + (Quarter.Percent * 2*Pi / 100);
    MidAngle := (MinAngle + MaxAngle) / 2;

    // Calcul des points
    MinPt := Point(Round(Spoke * Cos(MinAngle)) + Center.X, Height - Round(Spoke * Sin(MinAngle))
- Center.Y);
    MidPt := Point(Round(Spoke * Cos(MidAngle)) + Center.X, Height - Round(Spoke * Sin(MidAngle))
- Center.Y);
    MaxPt := Point(Round(Spoke * Cos(MaxAngle)) + Center.X, Height - Round(Spoke * Sin(MaxAngle))
- Center.Y);

    // Dessin du quartier
    Brush.Assign(Graphics.BackgroundBrush);
    with CircRect do
        Pie(Left, Top, Right, Bottom, MinPt.X, MinPt.Y, MaxPt.X, MaxPt.Y);

    // Calcul de la position du texte et affichage du texte
    if Quarter.ShowText then
    begin
        Brush.Assign(Graphics.TextBrush);
        Font.Assign(Graphics.Font);
        TextPos := Point((Center.X+MidPt.X) div 2, (Center.Y+MidPt.Y) div 2);
        if Quarter.Text <> '' then
            Text := Format('%s'#13#10'('%f%%)', [Quarter.Text, Quarter.Percent])
        else
            Text := Format('%f%%', [Quarter.Percent]);
        DrawTextRect := Rect(0, 0, Width, 0);
        DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CALCRECT or DT_CENTER or DT_NOPREFIX);
        with TextPos, DrawTextRect do
            DrawTextRect := Rect(X - Right div 2, Y - Bottom div 2, X + Right div 2, Y + Bottom div
2);
        DrawText(Handle, PChar(Text), -1, DrawTextRect, DT_CENTER or DT_NOPREFIX);
    end;
end;
end;
end;

procedure TCircleChart.DoContextPopup(MousePos : TPoint; var Handled : boolean);

```

CircChart.pas

```

begin
    FPopupQuarter := PointToQuarter(MousePos);
    inherited;
end;

procedure TCircleChart.MouseDown(Button : TMouseButton; Shift : TShiftState;
    X, Y : integer);
begin
    if Button = mbLeft then
    begin
        FClickedQuarter := PointToQuarterIndex(Point(X, Y));
        if (FClickedQuarter <> -1) and (not Quarters[FClickedQuarter].Enabled) then
            FClickedQuarter := -1;
        end;
    end;
end;

procedure TCircleChart.MouseUp(Button : TMouseButton; Shift : TShiftState;
    X, Y : integer);
begin
    if (Button = mbLeft) and (FClickedQuarter <> -1) then
    begin
        if PointToQuarterIndex(Point(X, Y)) = FClickedQuarter then
            Quarters[FClickedQuarter].Click;
        FClickedQuarter := -1;
        end;
    end;
end;

procedure TCircleChart.DoClickQuarter(Index : integer; Quarter : TChartQuarter);
begin
    if Assigned(FOnClickQuarter) then
        FOnClickQuarter(Self, Index, Quarter);
end;

procedure TCircleChart.ClickQuarter(Index : integer);
begin
    if Index <> -1 then
        DoClickQuarter(Index, Quarters[Index]);
end;

function TCircleChart.PointToQuarterIndex(Point : TPoint) : integer;
const
    TwoPi = 2*Pi;
var
    CenterToPoint, PtCos : Single;
    PtAngle, MinAngle, MaxAngle : Single;
    Quarter : TChartQuarter;
begin
    dec(Point.X, Width div 2);
    dec(Point.Y, Height div 2);
    Point.Y := -Point.Y;

    CenterToPoint := Sqrt(Point.X*Point.X + Point.Y*Point.Y);
    if CenterToPoint > Spoke then
    begin
        Result := -1;
        exit;
    end;

    PtCos := Point.X / CenterToPoint;
    PtAngle := ArcCos(PtCos);
    if Point.Y < 0 then
        PtAngle := -PtAngle;
    PtAngle := PtAngle - FBaseAngle * Pi / 180;
    PtAngle := PtAngle - TwoPi * Floor(PtAngle/TwoPi);

    MaxAngle := 0;
    Result := 0;
    while Result < FQuarters.Count do

```

CircChart.pas

```
begin
  Quarter := Quarters[Result];

  MinAngle := MaxAngle;
  MaxAngle := MinAngle + (Quarter.Percent * TwoPi / 100);

  if (MinAngle < PtAngle) and (PtAngle < MaxAngle) then exit;

  inc(Result);
end;
Result := -1;
end;

function TCircleChart.PointToQuarter(Point : TPoint) : TChartQuarter;
var Index : integer;
begin
  Index := PointToQuarterIndex(Point);
  if Index = -1 then Result := nil else
    Result := FQuarters[Index];
end;

end.
```

XIV - Test du composant

Après avoir passé avec succès l'étape de compilation du paquet, il faut tester le composant hors-EDI, ainsi que nous l'avons déjà dit lors de la construction des deux précédents composants.

Nous n'allons pas ici revoir une troisième fois comment tester un composant. La technique a été vue en **partie I, section III**, et revue en **partie II, section IX**.

Je vous laisse donc en exercice de révision le test de **TCircleChart**.

XV - Intégration dans la palette des composants

Afin d'insérer le **TCircleChart** et toutes ses classes acolytes, rouvrez l'unité **RegComposTuto.pas**. Commencez par ajouter l'unité **CircChart** à sa clause **uses**.

XV-A - Recenser le composant TCircleChart

Pour l'instant, la procédure **Register** se présente de cette façon :

```
procedure Register;
begin
  RegisterComponents('Tutoriel', [TSelectDirDialog, TDropImage]);
end;
```

Pour recenser le composant **TCircleChart**, ajoutez-le dans le tableau ouvert passé en paramètre.

```
RegisterComponents('Tutoriel', [TSelectDirDialog, TDropImage, TCircleChart]);
```

XV-B - Recenser l'action TChartQuarterAction

Afin de parachever le professionnalisme que vous êtes en droit d'exiger de vous-même, il faut encore recenser notre action personnalisée, **TChartQuarterAction**, afin que celle-ci apparaisse dans la boîte de dialogue **Classes d'actions standard**.

Ce tour de force est réalisé par la routine **RegisterActions**, déclarée dans l'unité **ActnList**. Voici sa déclaration :

```
procedure RegisterActions(const CategoryName: string;
  const AClasses: array of TBasicActionClass; Resource: TComponentClass);
```


Le paramètre **CategoryName** représente la catégorie dans laquelle seront affichées les actions à recenser, dans la boîte de dialogue **Classes d'actions standard**.

Le paramètre **AClasses** est un tableau ouvert de classes d'actions indiquant les actions à recenser.

Le troisième paramètre, **Resource**, doit être positionné à **nil** dans 99 % des cas. Si vous voulez de plus amples informations sur ce paramètre, reportez-vous à l'aide en ligne de Delphi.

Nous allons donc ajouter l'instruction suivante à la procédure **Register** :

```
RegisterActions('Graphiques', [TChartQuarterAction], nil);
```

 **La routine *RegisterActions* n'est pas disponible avec les éditions personnelles de Delphi. Vous ne pourrez donc pas recenser vos propres actions si vous utiliser une telle édition.**

XVI - Conclusion

Ainsi se termine notre étude des composants graphiques. Après compilation du paquet de conception, vous aurez l'immense joie de pouvoir jouer avec les collections, les actions, et les événements de **TCircleChart**.

À travers la réalisation de ce composants, nous avons pu explorer des concepts avancés de la construction de composants tels que les collections, l'interception de messages VCL et Windows, les actions personnalisées, etc.

Vous pouvez **télécharger tous les sources présentés dans ce tutoriel**, tels qu'ils sont à ce stade.

Si ce lien ne fonctionne pas chez vous, utilisez **celui-ci**.

Vous êtes invités à **réagir à cet article** sur les blogs. Merci.

La partie suivante montrera la création de composants acceptant le focus, les composants Windows.

XVII - Liens utiles

Voici quelques liens utiles en rapport avec ce dont avons parlé dans cette partie :

-  **Delphi User Interface Design : Using Actions, Menus, and ToolBars**

XVIII - Remerciements

Je voudrais adresser un très grand MERCI à **Laurent Dardenne** pour sa grande aide dans la réalisation de ce tutoriel, autant aux niveaux documentation, fond et forme.

Merci aussi à **gege2061** et **Bestiol** et encore une fois à **Laurent Dardenne** pour leurs relectures sur la forme et l'orthographe.

