Component Lists: TCollection And TCollectionItem

by Xavier Pacheco

You often need to maintain a collection of items, such as data types, objects, or components, which are manipulated in Delphi's design time environment. The Lines property of a TMemo is one example, it's a TStrings object type which encapsulates a list of strings. Various management tasks like adding, removing and streaming of the strings are handled by the TStrings class itself.

In many cases you might want to maintain a collection of items that aren't already encapsulated by an existing component which the user can manipulate at design time. There are a few approaches you can take to accomplish this. For example, you could create a component which encapsulates your collection and also performs the management functions, like the TStrings class. Another approach might be to override the default streaming mechanism of an owner component to make it aware of the collection of items. You'll have to code the management functions as part of the owner component. Although the two previously mentioned approaches are suitable, Delphi 2 offers a cleaner approach to maintaining collections of items with its TCollection and TCollectionItem classes.

The TCollection class is used to store a list of TCollectionItem objects. What's nice about TCollection is that it already knows how to perform the management functions on its collection of TCollectionItems. TCollection and TCollectionItem are not components, but rather descendants of TPersistent and therefore have built-in streaming mechanisms.

In order to use these classes to maintain a collection of items, you must create a descendant of each. The TCollectionItem descendant should encapsulate an element in the collection. The TCollection descendant will be made aware of the TCollectionItem descendant and can be made into a property of an existing component. An example of where you'll see these objects used is the TStatusBar component.

TStatusBar contains a property Panels of type TStatusPanels, which is a TCollection descendant. TStatusPanels is defined so that it stores a collection of the TCollectionItem descendant class TStatusPanel which represents a distinct panel on the status bar. Other places where descendant TCollection classes are used as properties are THeaderControl.Sections, TListView.Items and TDBGrid.Columns.

The example in this article will illustrate how you might use the TCollection and TCollectionItem classes to maintain a list of pie wedges for a pie graph component. The component, TPieGraphic, is not complex and probably not very useful as it stands. The intention is to focus on the technique of using the classes and not on the example itself.

The TPieGraphic component is a descendant of a TGraphicControl. TPieGraphic's Paint method draws the various pie wedges based on their value and color as specified by the user of the component. The component user can add, edit, remove and modify wedges both at design-time and run-time. The design-time interface is provided by the PiePieces property. PiePieces is a TCollection descendant which maintains a collection of TPieWedge objects. TPieWedge is a TCollectionItem descendant.

To create the TPieGraphic component, I took the following steps. Firstly, I defined TPieWedge, the TCollectionItem descendant which encapsulates a distinctive pie wedge. Then I defined TPiePieces, the TCollection descendant which maintains the collection of TPieWedges. Next I defined TPieGraphic, the TGraphicControl descendant which has a TPiePieces property, PiePieces, and paints the various pie wedges held by PiePieces. Lastly, I designed a property editor to allow the component user to add/remove items from the property TPieGraphic.PiePieces at design-time.

The remainder of this article will go through these steps. Listing 1 is the source code for the TPieWedge, TPiePieces and TPieGraphic classes.

Defining The TCollectionItem Descendant

Before creating my descendant of TCollectionItem, I had to determine what data was necessary to paint a pie wedge representing a certain value. More importantly, I had to determine what data needed to be stored at design time so that when a form containing my component was reloaded, the pie wedges created at design-time would still be intact. The data needing to be stored (or streamed) is a value which the pie wedge represents and a color with which to paint the pie wedge. Additionally, I need to keep a TBrush object for drawing purposes, which need not be stored.

I gave TPieWedge three private fields: FWedgeValue to hold an integer value for the given wedge, FColor to hold the pie wedge's color and FBrush to be used for drawing. I also published the properties WedgeValue and Color which refer to the private fields FWedgeValue and FColor respectively. This is an important step,

➤ *Facing page: Listing 1*
For additional commenting see file PIEGRAF.PAS on the disk

```
unit piegraf;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TPieGraphic = class;
  TPieWedge = class(TCollectionItem)
  private
    FWedgeValue: Integer;    // Value this wedge represents
    FColor: TColor;          // Color to paint the wedge
    FBrush: TBrush;          // Brush object to use
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    procedure SetWedgeValue(Value: Integer);
    procedure SetColor(Value: TColor);
  published
    { published properties will be streamed automatically }
    property WedgeValue: Integer
      read FWedgeValue write SetWedgeValue;
    property Color: TColor read FColor write SetColor;
  end;
  TPiePieces = class(TCollection)
  private
    FPieGraphic: TPieGraphic; // Owner component of property
    FTotal: Integer;          // Total of all WedgeValues
    function GetItem(Index: Integer): TPieWedge;
    procedure SetItem(Index: Integer; Value: TPieWedge);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(PieGraphic: TPieGraphic);
    function Add: TPieWedge;
    procedure UpdatePiePieces;
    function AddPiece(Value: Integer; wColor: TColor):
      TPieWedge;
    property Items[Index: Integer]: TPieWedge
      read GetItem write SetItem; default;
    property Total: Integer read FTotal;
  end;
  TPieGraphic = class(TGraphicControl)
  private
    FPiePieces: TPiePieces;
  protected
    procedure SetPiePieces(Value: TPiePieces);
  public
    procedure Paint; override;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure AddPiece(Value: Integer; wColor: TColor);
  published
    property PiePieces: TPiePieces
      read FPiePieces write SetPiePieces;
  end;
procedure Register;

implementation
uses DsgnIntf, Piegrpe;

constructor TPieWedge.Create(Collection: TCollection);
begin
  inherited Create(Collection);
  FBrush := TBrush.Create;
end;

destructor TPieWedge.Destroy;
begin
  FBrush.Free;
  inherited Destroy;
end;

procedure TPieWedge.Assign(Source: TPersistent);
begin
  if Source is TPieWedge then begin
    WedgeValue := TPieWedge(Source).WedgeValue;
    Color := TPieWedge(Source).Color;
    Exit;
  end;
  inherited Assign(Source);
end;

procedure TPieWedge.SetWedgeValue(Value: Integer);
begin
  FWedgeValue := Value;
  Changed(False);
end;

procedure TPieWedge.SetColor(Value: TColor);
begin
  FColor := Value;
  FBrush.Color := Value;
  Changed(False);
end;

{ TPiePieces }
constructor TPiePieces.Create(PieGraphic: TPieGraphic);
begin
  inherited Create(TPieWedge);
  FPieGraphic := PieGraphic;
end;

function TPiePieces.GetItem(Index: Integer): TPieWedge;
begin
  Result := TPieWedge(inherited GetItem(Index));
end;
```

```
procedure TPiePieces.SetItem(Index: Integer;
  Value: TPieWedge);
begin
  inherited SetItem(Index, Value);
end;

function TPiePieces.Add: TPieWedge;
begin
  Result := TPieWedge(inherited Add);
end;

function TPiePieces.AddPiece(Value: Integer;
  wColor: TColor): TPieWedge;
begin
  Result := Add;
  Result.WedgeValue := Value;
  Result.Color := wColor;
end;

procedure TPiePieces.UpdatePiePieces;
begin
  FPieGraphic.Refresh;
end;

procedure TPiePieces.Update(Item: TCollectionItem);
var
  i: integer;
begin
  FTotal := 0;
  for i := 0 to Count - 1 do
    FTotal := FTotal + Items[i].WedgeValue;
  if Item <> nil then
    UpdatePiePieces;
end;

{ TPieGraphic }
constructor TPieGraphic.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPiePieces := TPiePieces.Create(self);
  Width := 200;
  Height := 200;
end;

destructor TPieGraphic.Destroy;
begin
  FPiePieces.Free;
  inherited Destroy;
end;

procedure TPieGraphic.AddPiece(Value: Integer;
  wColor: TColor);
begin
  FPiePieces.AddPiece(Value, wColor);
  Refresh;
end;

procedure TPieGraphic.SetPiePieces(Value: TPiePieces);
begin
  FPiePieces.Assign(Value);
end;

procedure TPieGraphic.Paint;
var
  StartA, EndA: Integer;
  midX, midY, stX, stY, endX, endY: Integer;
  sX, sY, eX, eY: double;
  i: integer;
begin
  if FPiePieces.FTotal <> 0 then begin
    StartA := 0;
    for i := 0 to FPiePieces.Count - 1 do begin
      if i = FPiePieces.Count - 1 then
        EndA := 0
      else begin
        EndA := StartA +
          Trunc((Integer(FPiePieces.Items[i].FWedgeValue) /
          FPiePieces.FTotal) * 360);
        if EndA = StartA then EndA := StartA+1;
      end;
      midX := Width div 2;
      midY := Height div 2;
      sX := Cos((StartA / 180.0) * pi);
      sY := Sin((StartA / 180.0) * pi);
      eX := Cos((EndA / 180.0) * pi);
      eY := Sin((EndA / 180.0) * pi);
      stX := Round(sX * 100);
      stY := Round(sY * 100);
      endX := Round(eX * 100);
      endY := Round(eY * 100);
      with Canvas do begin
        { Copy the brush from the TPieWedge to this Canvas }
        Brush := FPiePieces.Items[i].FBrush;
        Pie(0,0, Width,Height, midX + stX, midY - stY,
          midX + endX, midY - endY);
      end;
      StartA := EndA;
    end;
  end;
end;

procedure Register;
begin
  RegisterComponents(’Test’, [TPieGraphic]);
  RegisterPropertyEditor(TypeInfo(TPiePieces), TPieGraphic,
    ’PiePieces’, TPiePiecesProperty);
end;

end.
```

because whatever gets published is automatically streamed when the form using the TPieGraphic component is saved. This is what is so nice about using the TCollectionItem to wrap elements of the collection. You just tell the TCollectionItem what to save by making it a published property.

The TPieWedge.Create constructor takes a TCollection as a parameter. This parameter gets assigned to the TPieWedge.Collection property which is inherited from the TCollectionItem class. This allows TCollectionItem descendants to refer to the TCollection with which they are associated. In addition to calling the inherited constructor, TPieWedge's Create constructor also instantiates the FBrush object, which is freed in TPieWedge.Destroy.

It is necessary to override the Assign method which is actually inherited from TPersistent. TPieWedge.Assign is responsible for copying the TPieWedge passed in as a parameter. You will notice in Listing 1 that TPieWedge.Assign first ensures that a TPieWedge is passed in and then copies its fields. Notice that this method also assigns these values to its properties rather then to its private field members. The reason for this is to invoke any side-effects that occur in the property write access methods for those properties.

The TPieWedge.SetWedgeValue and TPieWedge.SetColor methods are the access methods for the WedgeValue and WedgeColor properties. These methods do as expected in assigning the specified values to the appropriate private fields. They also both call the TCollectionItem.Changed method. This method causes the associated TCollection object to call its Update method, which is an abstract method that must be overriden to perform any necessary logic whenever a change is made to a TCollectionItem. You will see later how I overrode this method to maintain a total of the TPieWedge values and to re-draw the TPieGraphic to reflect any changes.

You can see that creating the TCollectionItem descendant is actually very simple. Its main purpose is to specify which data of a collection's element to store by making that data a published property. The rest is just setting up the various access methods for those particular properties and overriding a few necessary methods.

One general point I should make about encapsulating a component or object with a TCollectionItem is that you shouldn't try to stream the component itself. Rather, you simply publish the necessary data required to create the component in the state that it was saved. The TCollectionItem should create an instance of this component and use the streamed data to restore the component's state.

Defining The TCollection Descendant

There's a bit more to do to the TCollection descendant type, TPiePieces, to make it aware of the TPieWedge. First, I need to maintain a link to the component of which TPiePieces will be a property. I used the FPieGraphic field for this purpose, which is of type TPieGraphic. Although I haven't yet defined TPieGraphic, I placed a forward declaration so that I could use it in the TPiePieces definition. Also, I needed to maintain a total of the TPieWedge values and I use FTotal for this purpose.

The TPiePieces.Create constructor takes a TPieGraphic parameter and assigns it to FPieGraphic. It does this after calling the inherited TCollection.Create constructor. Notice that the TPiePieces.Create constructor does not override the TCollection.Create constructor but rather creates its own constructor and just calls the inherited one. TCollection.Create takes the type of the TCollectionItem descendant with which it is associated as a parameter. TCollection uses this information internally in creating and adding new TPieWedge items to its collection list.

Earlier, I said that it is necessary to override the abstract TCollection.Update method to tell TPiePieces what to do if a change is made to an item in its collection. TPiePieces.Update recalculates the total of the TPieWedge values in case the user changed a value or added/removed a TPieWedge. It then forces the TPieGraphic to repaint itself by calling TPiePieces.UpdatePiePieces which in turn calls FPieGraphic.Refresh. The method UpdatePiePieces was created to give the component user a public method to force a repaint of the pie wedges.

The TCollection.Add method creates and adds to its collection list a TCollectionItem descendant. Instead of returning a reference to a TPieWedge, however, it returns a reference to the TCollectionItem base class. Internally, TCollection knows to create a TPieWedge because the type with which it is associated is passed to its constructor and it uses this information to create the correct TCollectionItem descendant type. However, its Add function cannot know which type to return. Therefore, I created an Add function specific to TPiePieces which calls the inherited TCollection.Add method and typecasts its return value to the appropriate type. This just means the user doesn't have to perform this intermediate step.

I also created a function called TCollection.AddPiece which takes a value and a color and adds a new pie piece to the collection with the specified property settings. This gives the user a run-time method with which to add new pieces.

Finally, I declared two properties: Items and Total. Total is a read only property which returns the value stored in FTotal – the total of the pie wedge values. Items is the default array property which allows the user to access the collection items sequentially. The GetItem and SetItem access methods call TCollection's GetItem and SetItem methods to retrieve and set the specified TCollectionItem instance.

The important thing to remember about your TCollection descendant is that you're making it aware of your TCollectionItem descendant. Since one of the primary purposes of creating a collection is to give the user a design-time interface with which s/he can manipulate a collection of items, it makes sense to create a component of which the TCollection descendant will become a property. For the TPiePieces, this component would be the TPieGraphic.

Defining The Owner Component

TPieGraphic isn't too complex, it's just a TGraphicControl descendant which contains a TPiePieces property. Its various methods make it so that it knows how to assign pie wedges to the PiePieces property and also allow it to paint the pie chart based on the pie wedge values and colors.

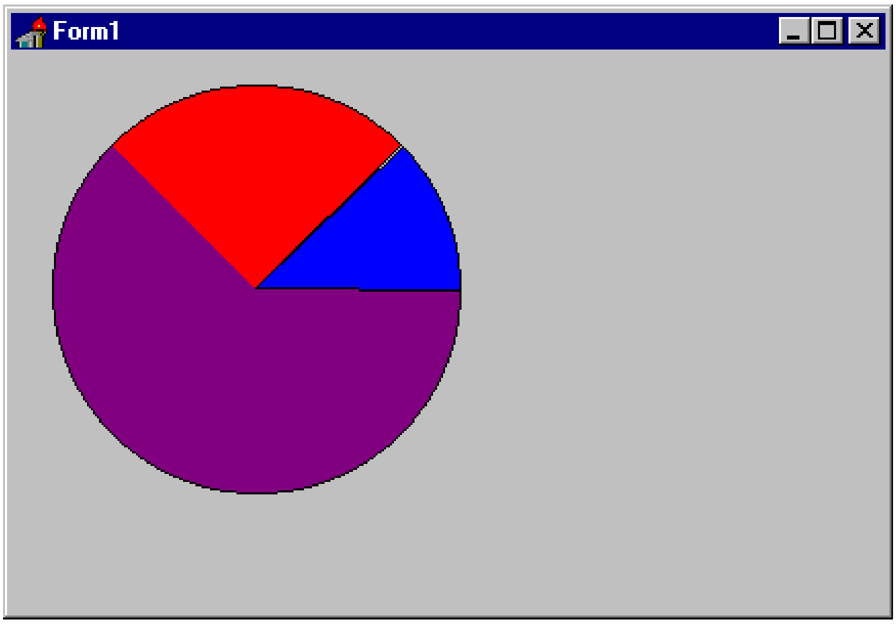
The TPieGraphic.Create constructor instantiates the TPiePieces collection instance, FPiePieces, and sets its default width and height. FPiePieces is accessed through the PiePieces property. SetPiePieces is the write access method for PiePieces. This method replaces the TPiePieces parameter passed in to its TPiePieces instance. TPieGraphic.Destroy simply frees the TPiePieces instance. The TPieGraphic.AddPiece method is just another interface function to add another pie wedge to the TPiePieces collection.

The main method of TPieGraphic is its Paint method, which iterates through the pie wedges and paints them to the TPieGraphic.Canvas. This is just a modified copy of the TPie.Paint method which ships with Borland's examples, but making use of the TPieWedge properties to paint the wedges.

At this point, you can successfully use this component to draw a pie graph. To add pie wedges to the PiePieces property just execute the TPieGraphic.AddPiece method as shown below, resulting in the output shown in Figure 1.

```
PieGraphic1.AddPiece(
  10, clBlue);
PieGraphic1.AddPiece(
  20, clRed);
PieGraphic1.AddPiece(
  50, clPurple);
```

The real benefit to using the TCollection and TCollectionItem is that they can allow the user to modify your collection of items at design-time. These modifications will then be stored along with the form on which the component sits and later restored when the user reloads the form. Therefore, you should give the user the ability to edit the list of items at design-time with a property editor.



➤ *Figure 1*

Designing A Property Editor

I won't go into the details of how to design a property editor since that is not the focus of this article. Instead, I'll discuss the specifics of the TPiePiecesProperty property editor and how it works with the TPiePieces collection class. For an excellent discussion on designing property editors see the article *Under Construction: Property Editors* by Bob Swart in Issue 6, February 1996.

Notice in Listing 1 that we included the units PieGrpe and DsgnIntf in the uses statement. DsgnIntf is where the base property editor classes are defined. PieGrpe is where the TPiePiecesProperty editor is defined – the property editor for the TPiePieces class. Listing 2 (over the page) shows PIEGRPE.PAS.

The property editor class, TPiePiecesProperty, overrides three methods. GetAttributes is overriden to tell the Object Inspector that this property will invoke a dialog when edited. This places the ellipsis button in the Object Inspector for the PiePieces property. the Edit method calls the EditPiePieces function to which it passes the TPiePieces property being edited. A reference to the actual property can be obtained by using the GetOrdValue function as shown in the Edit procedure. The GetValue function writes the class type of the PiePieces value in the Object Inspector.

It is the TPieGraphEditor dialog where the editing of the TPiePieces collection actually occurs. Figure 2 shows TPieGraphEditor.

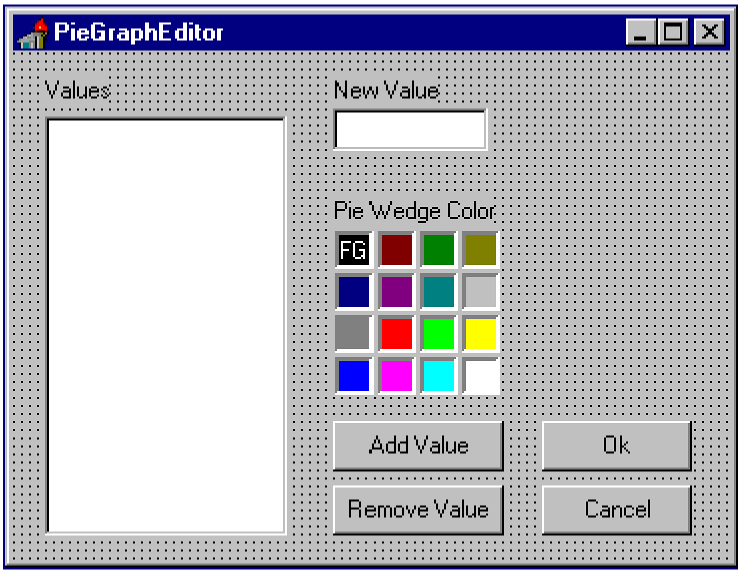
The TPieGraphEditor dialog uses an owner-draw TListBox to display the TPieWedge values in their respective colors. The user can add and remove pie wedges and uses the TEdit to specify the wedge value and the TColorGrid to specify the wedge color.

The TPieGraphEditor is invoked from the EditPiePieces function. This function instantiates the dialog and assigns the TPiePieces class passed to its private TPieFields member, FPieFields, which is used by TPieGraphEditor's methods to allow the user to modify the pie wedges. EditPiePieces also instantiates an internal TPieGraphic instance which it uses as a backup to restore the original pie wedge values in case the user

➤ *Listing 2 For additional commenting see file* PIEGRPE.PAS *on the disk*

```
unit piegrpe;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, PieGraf, DsgnIntF, TypInfo,
  StdCtrls, Mask, ColorGrd;
type
  TPieGraphEditor = class(TForm)
    Label1: TLabel;
    ValuesListBox: TListBox;
    Label2: TLabel;
    AddBtn: TButton;
    ColorGrid1: TColorGrid;
    Label3: TLabel;
    RemoveBtn: TButton;
    OkBtn: TButton;
    CancelBtn: TButton;
    NewValue: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ValuesListBoxDrawItem(Control:
      TWinControl; Index: Integer;
    Rect: TRect; State: TOwnerDrawState);
    procedure AddBtnClick(Sender: TObject);
    procedure RemoveBtnClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure NewValueKeyPress(Sender: TObject;
      var Key: Char);
  private
    FPiePieces: TPiePieces;
    FPieGraphic: TPieGraphic; // Used as a Backup
    Modified: Boolean;
    procedure UpdateValuesListBox;
  end;
  { Now declare TPropertyEditor descendant and override
    the required methods }
  TPiePiecesProperty = class(TPropertyEditor)
    function GetAttributes: TPropertyAttributes;
      override;
    function GetValue: String ; override;
    procedure Edit; override;
  end;
{ This function will be called by the property editor’s
  Edit method }
function EditPiePieces(PiePieces: TPiePieces): Boolean;
var
  PieGraphEditor: TPieGraphEditor;

implementation
{$R *.DFM}

function IsCharNumeric(C: Char): Boolean;
var Code, V: Integer;
begin
  Val(C, V, Code);
  Result := Code = 0;
end;

function EditPiePieces(PiePieces: TPiePieces): Boolean;
begin
  with TPieGraphEditor.Create(Application) do begin
    try
      { Point to the actual TPiePieces collection }
      FPiePieces := PiePieces;
      { Copy TPiePieces to the backup FPieGraphic which
        will be used as a backup in case user cancels }
      FPieGraphic.PiePieces.Assign(PiePieces);
      { Draw the listbox with list of TPiePieces Values }
      UpdateValuesListBox;
      ShowModal; // Display the form
      Result := Modified;
    finally
      Free;
    end;
  end;
end;

{ TPieGraphEditor }
procedure TPieGraphEditor.UpdateValuesListBox;
var i: Integer;
begin
  ValuesListBox.Clear; // First clear the list box
  for i := 0 to FPiePieces.Count - 1 do
    with FPiePieces[i] do
      ValuesListBox.Items.AddObject(IntToStr(WedgeValue),
        Pointer(Color));
end;

procedure TPieGraphEditor.FormCreate(Sender: TObject);
begin
  FPieGraphic := TPieGraphic.Create(self);
end;

procedure TPieGraphEditor.FormDestroy(Sender: TObject);
begin
  FPieGraphic.Free;
end;

procedure TPieGraphEditor.ValuesListBoxDrawItem(
  Control: TWinControl; Index: Integer; Rect: TRect;
  State: TOwnerDrawState);
{ Uses an owner-draw list box to draw the TPieWedge
  values in their specified color }
begin
  with ValuesListBox do begin
    Canvas.FillRect(Rect);
    Canvas.Font.Color := TColor(Items.Objects[Index]);
    DrawText(Canvas.Handle, PChar(Items[Index]),
      Length(Items[Index]), Rect, dt_Left or dt_VCenter);
   end;
end;

procedure TPieGraphEditor.AddBtnClick(Sender: TObject);
var
  PieWedge: TPieWedge;
begin
  if StrToInt(NewValue.Text) > 0 then begin
    ValuesListBox.Items.Add(NewValue.Text);
    ValuesListBox.Refresh;
    PieWedge :=
      FPiePieces.AddPiece(StrToInt(NewValue.Text),
      ColorGrid1.ForegroundColor);
    Modified := True;
  end;
end;

procedure TPieGraphEditor.RemoveBtnClick(Sender:
  TObject);
var i: integer;
begin
  i := ValuesListBox.ItemIndex;
  if i >= 0 then begin
    { Remove the item from the listbox }
    ValuesListBox.Items.Delete(i);
    { Remove the item from the collection }
    FPiePieces[i].Free;
    Modified := True;
  end;
end;

procedure TPieGraphEditor.CancelBtnClick(Sender:
  TObject);
begin
  FPiePieces.Assign(FPieGraphic.PiePieces);
  Modified := False;
  ModalResult := mrCancel;
end;

procedure TPieGraphEditor.NewValueKeyPress(Sender:
  TObject; var Key: Char);
begin
  if not IsCharNumeric(Key) then Key := #0;
end;

{ TPiePiecesProperty }
function TPiePiecesProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paDialog];
end;

procedure TPiePiecesProperty.Edit;
begin
  if EditPiePieces(TPiePieces(GetOrdValue)) then begin
    Modified;
  end;
  TPiePieces(GetOrdValue).UpdatePiePieces;
end;

function TPiePiecesProperty.GetValue: String;
begin
  Result := Format(’(%s)’, [GetPropType^.Name]);
end;
end.
```

➤ *Figure 2*



cancels the edit operation. By the way, another approach you may consider is to have the user edit the internal pie wedge values and then copy them to the actual property only when the user clicks the Ok button, or you can also place an Apply button on the form.

After creating the internal TPieGraphic instance, the method UpdateValuesListBox is called, which adds the pie wedge values and colors to the ValuesListBox. Finally, the form is shown modally and the user can then edit the TPiePieces. The AddBtnClick adds a new TPieWedge instance to the FPiePieces field. Remember, FPiePieces refers to the actual property being edited. RemoveBtnClick removes the selected TPieWedge instance. Both AddBtnClick and RemoveBtnClick ensure that ValuesListBox reflects the changes made.

Conclusion

That's all there is to using the TCollection and TCollectionItem classes to create and manage a collection of items that can be manipulated and saved at design-time in Delphi. Although it seems that there are several steps to take, these are the same steps you would take when designing just about any collection of items.

Xavier Pacheco is a Field Consulting Engineer with Borland International and co-author of the upcoming book *Delphi 2.0 Developer's Guide* from Sams publishing. Xavier can be reached at xpacheco@wpo.borland.com or on Compuserve at 76711,666