This article will cover how to write advanced properties, how to write custom streaming for those properties, and sub-properties.

This article originally appeared in Delphi Developer

This article is part two of a three part article on components. Part one covered the basic creating of components, part two will cover how to write advanced properties, how to write custom streaming for those properties, and sub-properties. The final part will cover property / component editors, how to write dedicated editors for your component / property, and how to write "hidden" components.

Quite often it is necessary to write components that perform more advanced functions. These components often need to either reference other components, have custom property data formats, or have a property that owns a list of values rather than a single value. In this part, we will explore various examples covering these very subjects, starting with the most simple.

Component references

Some components need to reference other components. TLabel for instance has a "FocusControl" property. When you include an ampersand in the "Caption" property it underlines the next letter (&Hello becomes Hello), pressing the shortcut key ALT-H on your keyboard will trigger an event in your label. If the "FocusControl" property has been set focus will be passed to the control specified.

To have such a property in your own component is quite simple. All you do is declare a new property, and set the property type to the lowest base class that it may accept (TWinControl will allow any descendent of TWinControl to be used), but, there are implications.

```
type
  TSimpleExample = class(TComponent)
  private
    FFocusControl: TWinControl;
  protected
    procedure SetFocusControl(const Value: TWinControl); virtual;
  public
  published
    property FocusControl: TWinControl read FFocusControl write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value: TWinControl);
begin
  FFocusControl := Value;
end;
```

Take the above example. This is quite a simple example (hence the component name) of how to write a component that references another component. If you have such a property in your component the Object Inspector will show a combobox with a list of components that match the criteria (all components descended from TWinControl).

Our component may do something like

```
procedure TSimpleExample.DoSomething;
begin
```

```
  if (Assigned(FocusControl)) and
     (FocusControl.Enabled) then
    FocusControl.Setfocus;
end;
```

First we check if the property has been assigned, if so we set focus to it, but there are situations when the property is not Nil yet the component it points to is no longer valid. This often happens when a property references a component that has been destroyed.

Luckily Delphi provides us with a solution. Whenever a component is destroyed it notifies its owner (our form) that it is being destroyed. At this point every component owned by the same form is notified of this event too. To trap this event we must override a standard method of TComponent called "Notification".

```
type
  TSimpleExample = class(TComponent)
  private
    FFocusControl: TWinControl;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure SetFocusControl(const Value: TWinControl); virtual;
  public
  published
    property FocusControl: TWinControl read FFocusControl write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value: TWinControl);
begin
  FFocusControl := Value;
end;

procedure TSimpleExample.Notification(AComponent :TComponent;
  Operation : TOperation);
begin
  inherited; //Never forget to call this
  if (Operation = opRemove) and
     (AComponent = FocusControl) then
    FFocusControl := nil;
end;
```

Now when our referenced component is destroyed we are notified, at which point we can set our reference to Nil. Note, however, that I said *"every component owned by the same form is notified of this event too"*

This introduces us with another problem. We are only notified that the component is being destroyed if it is owned by the same form. It is possible to have our property point to components on other forms (or even without an owner at all), and when these components are destroyed we are not notified. Yet again there is a solution.

TComponent introduces a method called "FreeNotification". The purpose of FreeNotification is to tell the component (FocusControl) to keep us in mind when it is destroyed.

An implementation would look like this

```
type
  TSimpleExample = class(TComponent)
  private
    FFocusControl: TWinControl;
  protected
```

```
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure SetFocusControl(const Value: TWinControl); virtual;
  public
  published
    property FocusControl: TWinControl read FFocusControl write SetFocusControl;
  end;

procedure TSimpleExample.SetFocusControl(const Value: TWinControl);
begin
  if Value <> FFocusControl then
  begin
    if Assigned(FFocusControl) then
      FFocusControl.RemoveFreeNotification(Self);

    FFocusControl := Value;

    if Assigned(FFocusControl) then
      FFocusControl.FreeNotification(Self);
  end;
end;

procedure TSimpleExample.Notification(AComponent: TComponent;
  Operation : TOperation);
begin
  if (Operation = opRemove) and
     (AComponent = FocusControl) then
    FFocusControl := nil;
end;
```

When setting our FocusControl property we first check if it is already set to a component. If it is already set we need to tell the original component that we no longer need to know when it is destroyed. Once our property has been set to the new value we inform the new component that we require a notification when it is freed. The rest of our code remains the same as the referenced component still calls our Notification method.

Sets

This section is really quite simple and will not take long to cover. I do not doubt that you are already familiar with creating your own ordinal types.

```
type
  TComponentOption = (coDrawLines,
                      coDrawSolid,
                      coDrawBackground);
```

Properties of this type will show a combobox with a list of all possible values, but sometimes you will need to set a combination of many (or all) of these values. This is were sets come in to play

```
Type
  TComponentOption = (coDrawLines,
                      coDrawSolid,
                      coDrawBackground);
  TComponentOptions = set of TComponentOption;
```

Publishing a property of type TComponentOptions would result in a [+] appearing next to

our property name. When you click to expand the property you will see a list of options. For each element in TComponentOption you will see a Boolean property, you can include / exclude elements from your set by setting its value to True / False.

It is simple to check / alter elements in a set from within our component like so.

```
if coDrawLines in OurComponentOptions
  then DrawTheLines;
```

or

```
procedure TSomeComponent.SetOurComponentOptions(const Value:
TComponentOptions);
begin
  if (coDrawSolid in Value) and
  (coDrawBackground in value) then
    {raise an exception}

  FOurComponentOptions := Value;
  Invalidate;
end;
```

Binary properties

Sometimes it is necessary to write your own streaming routines to read and write custom property types (This is how Delphi reads / writes the Top and Left properties for non visible components without actually publishing those properties in the object inspector).

For example, I once wrote a component to shape a form based on a bitmap image. My code at the time to convert a bitmap to a window-region was extremely slow and would not possibly be of any use at runtime. My solution was to convert the data at design time, and stream the binary data that resulted from the conversion. To create binary properties is a three step process.

1. Write a method to write the data.
2. Write a method to read the data.
3. Tell Delphi that we have a binary property, and pass our read / write methods.

```
type
  TBinaryComponent = class(TComponent)
  private
    FBinaryData : Pointer;
    FBinaryDataSize : DWord;
    procedure WriteData(S : TStream);
    procedure ReadData(S : TStream);
  protected
    procedure DefineProperties(Filer : TFiler); override;
  public
    constructor Create(AOwner : TComponent); override;
  end;
```

DefineProperties is called by Delphi when it needs to stream our component. All we need to do is to override this method, and add a property using either TFiler.DefineProperty or TFiler.DefineBinaryProperty.

```
procedure TFiler.DefineBinaryProperty(const Name: string;
  ReadData, WriteData: TStreamProc; HasData: Boolean);

constructor TBinaryComponent.Create(AOwner: TComponent);
begin
  inherited;
  FBinaryDataSize := 0;
end;

procedure TBinaryComponent.DefineProperties(Filer: TFiler);
var
  HasData : Boolean;
begin
  inherited;
  HasData := FBinaryDataSize <> 0;
  Filer.DefineBinaryProperty('BinaryData',ReadData,
    WriteData, HasData );
end;

procedure TBinaryComponent.ReadData(S: TStream);
begin
  S.Read(FBinaryDataSize, SizeOf(DWord));
  if FBinaryDataSize > 0 then begin
    GetMem(FBinaryData, FBinaryDataSize);
    S.Read(FBinaryData^, FBinaryDataSize);
  end;
end;

procedure TBinaryComponent.WriteData(S: TStream);
begin
  //This will not be called if FBinaryDataSize = 0

  S.Write(FBinaryDataSize, Sizeof(DWord));
  S.Write(FBinaryData^, FBinaryDataSize);
end;
```

Firstly we override DefineProperties. Once we have done this we define a binary property with the values -

-BinaryData : The invisible property name to be used.
-ReadData : The procedure responsible for reading the data.
-WriteData : The procedure responsible for writing the data.
-HasData : If this is false, the WriteData procedure is not even called.

Persistency

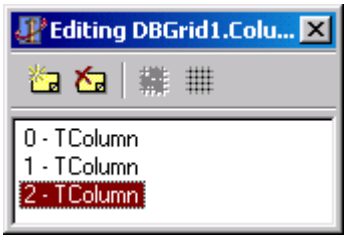
A quick explanation of persistency is in order as we shall refer to it in the following sections. Persistency is what makes it possible for Delphi to read and write the properties of all of its components. TComponent derives from a class called TPersistent. TPersistent is simply a Delphi class capable of having its properties read and written by Delphi, which means that any descendents of TPersistent also have this same capability.

Collections

As we progress through this article we cover component properties of more complexity. Collections are one of the most complex "standard" Delphi property types. If you drop a TDBGrid onto a form and look at its properties in the Object Inspector, you will see a property named "Columns".

Columns is a collection property, when you click on the [..] button you will see a small window pop up. This window is the standard property editor for TCollection properties (and descendents of TCollection).

descendents of TCollection).



Whenever you click the "New" button you see a new item added (a TColumn item), clicking on that item will select it into the Object Inspector so that you can alter its properties / events. How is this done ?

The Columns property descends from TCollection. TCollection is similar to an array, which contains a list of TCollectionItem's. Because TCollection is descended from TPersistent it is able to stream this list of items, similarly, TCollectionItem is also descended from TPersistent and can also stream its properties. So what we have is an array-like item capable of streaming all of its items and their properties.

The first thing to do when creating our own structure based on TCollection / TCollectionItem is to define our CollectionItem.

(See OurCollection.pas)

```
type
  TOurCollectionItem = class(TCollectionItem)
  private
    FSomeValue : String;
  protected
    function GetDisplayName : String; override;
  public
    procedure Assign(Source: TPersistent); override;
  published
    property SomeValue : String
      read FSomeValue
      write FSomeValue;
  end;
```

What we have done here is to create a descendent of TCollectionItem. We have added a token property called "SomeValue", overridden the GetDisplayName function (to alter the text that is shown in the default editor), and finally overridden the Assign method in order to allow TOurCollectionItem to be assigned to another TOurCollectionItem. If we omit the final step then the Assign method of our Collection class will not work !

```
procedure TOurCollectionItem.Assign(Source: TPersistent);
begin
  if Source is TOurCollectionItem then
    SomeValue := TOurCollectionItem(Source).SomeValue
  else
    inherited; //raises an exception
end;

function TOurCollectionItem.GetDisplayName: String;
begin
  Result := Format('Item %d',[Index]);
end;
```

The implementation of TOurCollection is much more complex, and requires us to do quite a

The implementation of TOurCollection is much more complex, and requires us to do quite a bit of work.

```
TOurCollection = class(TCollection)
  private
    FOwner : TComponent;
  protected
    function GetOwner : TPersistent; override;
    function GetItem(Index: Integer): TOurCollectionItem;
    procedure SetItem(Index: Integer; Value:
      TOurCollectionItem);
    procedure Update(Item: TOurCollectionItem);
  public
    constructor Create(AOwner : TComponent);

    function Add : TOurCollectionItem;
    function Insert(Index: Integer): TOurCollectionItem;

    property Items[Index: Integer]: TOurCollectionItem
      read GetItem
      write SetItem;
  end;
```

There are a number of items to cover based on the above class declaration, so we shall start from the top and cover each in turn.

GetOwner is a virtual method introduced in TPersistent. This needs to be overridden as the default code for this method returns Nil. In our implementation we alter the constructor to receive only one parameter (AOwner : TComponent). We store this parameter in FOwner, which is then passed as the result of GetOwner (TComponent descends from TPersistent, so is therefore a valid result type).

```
constructor TOurCollection.Create(AOwner: TComponent);
begin
  inherited Create(TOurCollectionItem);
  FOwner := AOwner;
end;

function TOurCollection.GetOwner: TPersistent;
begin
  Result := FOwner;
end;
```

Not only does Create store the owner (which is required for the Object Inspector to work correctly), it also tells Delphi what class our CollectionItem is by calling "inherited Create (TOurCollectionItem)".

GetItem / SetItem are declared the same way as they are in TCollection, but instead of working on TCollectionItem they work on our new class TOurCollectionItem. These are used in our "Items" property later on.

Update as above is a straight forward replacement of the original, working on our new CollectionItem class instead.

Add / Insert are both responsible for adding items to the list, these have both been replaced to return objects of the appropriate class.

Finally, an "Items" property is introduced to replace the original Items property, again so that we are returned a result of TOurCollectionItem rather than TCollectionItem saving us the unnecessary problem of typecasting the result each time.

Finally an example of implanting this property type in a component of our own.

```
TCollectionComponent = class(TComponent)
  private
    FOurCollection : TOurCollection;
    procedure SetOurCollection(const Value:
      TOurCollection);
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property OurCollection : TOurCollection
      read FOurCollection
      write SetOurCollection;
  end;
```

It is as simple as that. Once our TCollection class is written all of the hard work is done. Our constructor creates the collection class, the destructor destroys it, and SetOurCollection does this.

```
constructor TCollectionComponent.Create(AOwner: TComponent);
begin
  inherited;
  FOurCollection := TOurCollection.Create(Self);
end;

destructor TCollectionComponent.Destroy;
begin
  FOurCollection.Free;
  inherited;
end;

procedure TCollectionComponent.SetOurCollection(
  const Value: TOurCollection);
begin
  FOurCollection.Assign(Value);
end;
```

As mentioned before, the (Self) passed to the TOurCollectionItem.Create is stored in TOurCollection's FOwner variable, which is passed as the result of GetOwner. A point to note here is that in SetOurCollection we do not set FOurCollection := value as you are replacing the object (objects are simply pointers), we Assign our property to the value.

Later versions of Delphi make this simpler still. Rather than having to override GetOwner in our Collection class, we can now derive our Collection from TOwnedCollection instead. TOwnedCollection is a wrapper for TCollection with this work done for us.

Sub-properties

Earlier on in this article we saw how it was possible to create an expandable property. The limitation of the earlier technique was that each sub-item appeared as a Boolean property. This next section will demonstrate how to create expandable properties that can contain any property type.

If a component requires a property that is a record type, this could quite easily be implemented by exposing each of the properties seperately. If however our component needs to introduce two or more properties of the same complex type our Object Inspector view suddenly becomes very complicated.

The answer is to create a complex structure (alike to a record or object) and to publish this structure as a property whenever needed. The obvious problem is that Delphi does not know how to display this property unless we tell it. Creating a fully blown property editor (with dialogs etc) would be overkill, so luckily Delphi has provided a solution. As mentioned earlier, Delphi's internal streaming is based around the TPersistent class. The first step therefore is to derive our complex structure from this class.

```
type
  TExpandingRecord = class(TPersistent)
  private
    FIntegerProp : Integer;
    FStringProp : String;
    FCollectionProp : TOurCollection;
    procedure SetCollectionProp(const Value:
      TOurCollection);
  public
    constructor Create(AOwner : TComponent);
    destructor Destroy; override;

    procedure Assign(Source : TPersistent); override;
  published
    property IntegerProp : Integer
      read FIntegerProp
      write FIntegerProp;
    property StringProp : String
      read FStringProp
      write FStringProp;
    property CollectionProp : TOurCollection
      read FCollectionProp
      write SetCollectionProp;
  end;
```

In the above structure we have created a descendent of TPersistent and given it three example properties, an integer, a string, and the collection that we created earlier in this article TOurCollection.

The constructor and destructor simply take care of creating and destroying the CollectionProp object, and the SetCollectionProp is implanted to stop this object reference from being lost (we Assign(value), rather than FCollectionProp := value). Whereas Assign is implemented in order to allow us to assign the properties of TExpandableRecord to another TExpandableRecord (again, this is necessary because we will need to Assign it when it is finally implemented as a property of a component).

```
procedure TExpandingRecord.Assign(Source: TPersistent);
begin
  if Source is TExpandingRecord then
    with TExpandingRecord(Source) do begin
      Self.IntegerProp := IntegerProp;
      Self.StringProp := StringProp;

      //This actually assigns
      Self.CollectionProp := CollectionProp;
    end else
      inherited; //raises an exception
end;

constructor TExpandingRecord.Create(AOwner : TComponent);
begin
  inherited Create;
  FCollectionProp := TOurCollection.Create(AOwner);
end;
```

```
end;

destructor TExpandingRecord.Destroy;
begin
  FCollectionProp.Free;
  inherited;
end;

procedure TExpandingRecord.SetCollectionProp(const Value: TOurCollection);
begin
  FCollectionProp.Assign(Value);
end;
```

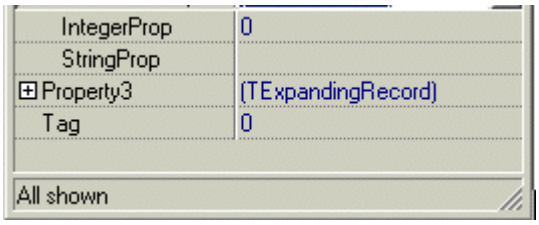
Once this code has been implemented all of the hard work is done. To implement this class as an object is now quite straight forward.

(See ExpandingComponent.pas)

```
TExpandingComponent = class(TComponent)
  private
    FProperty1,
    FProperty2,
    FProperty3 : TExpandingRecord;
  protected
    procedure SetProperty1(const Value :
      TExpandingRecord);
    procedure SetProperty2(const Value :
      TExpandingRecord);
    procedure SetProperty3(const Value :
      TExpandingRecord);
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property Property1 : TExpandingRecord
      read FProperty1
      write SetProperty1;
    property Property2 : TExpandingRecord
      read FProperty2
      write SetProperty2;
    property Property3 : TExpandingRecord
      read FProperty3
      write SetProperty3;
  end;
```

The constructor will need to create the three objects used as properties, the destructor will obviously need to destroy them. The SetPropertyX procedures will Assign(Value) to the correct object.

Compile this component into a package and drop a TExpandingComponent onto your form. Looking in the object inspector you will notice that our TExpandingRecord propertys all have a [+] next to them, clicking this button will expand our property to reveal all of its sub-properties.



At first glance all looks well, our properties are shown with an expandable sub-set of properties, but everything is not as perfect as it may first seem. Clicking on the [..] button of our "CollectionProp" does not invoke the standard TCollection editor, in fact, it does nothing.

You may ask yourself "What have we done wrong ?", the answer is "Nothing !". The error here is not on our part at all, the fault lies with the developers of Delphi, Borland. Although I like to think of the developers of Delphi as infallible, they are not, and they do sometimes make mistakes, and this is a perfect example of one.

When you register a property editor you can limit the component scope that it applies to. You can specify that it should only work on certain property names, or only on certain components, and this is what they have done. Although Delphi's architecture defines that the lowest object-form capable of streaming is TPersistent, someone at Borland registered the property editor to work only on objects that descend from TComponent. An oversight I am sure, but that doesn't really help us at all.

The answer to this problem is to descend our TExpandableRecord from TComponent instead of TPersistent, then the property editor will be invoked. The problem is that Delphi's default property editor for properties of type TComponent (and descendents) shows a combobox rather than showing an expandable view of sub-properties.

The whole solution to this dilemma lies in property editors, and will be covered in the final part of this series.
