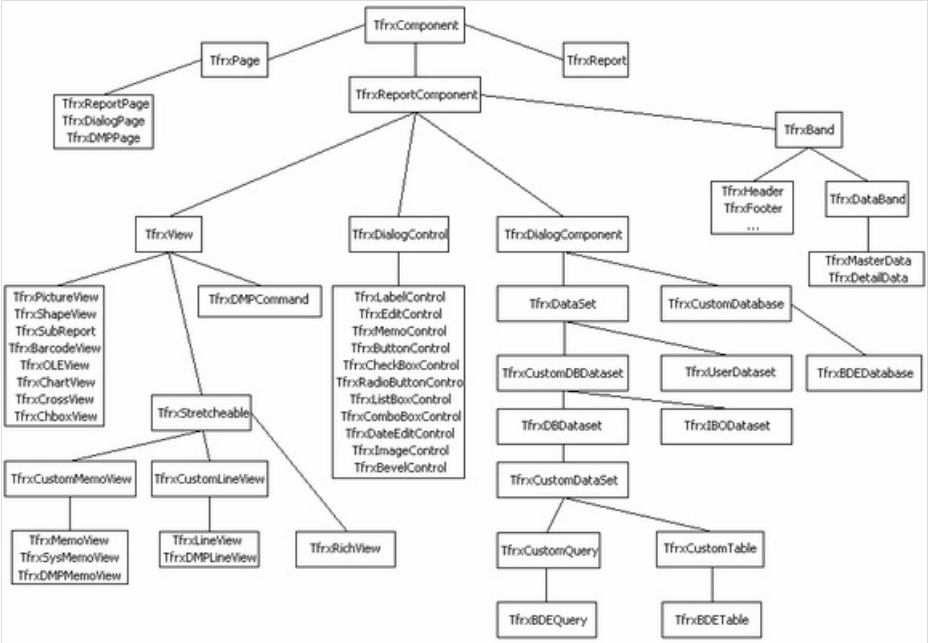
FastReport VCL Developer Manual

Version 2024.2

FastReport Classes Hierarchy



`TfrxComponent` is the base class for all FastReport components. Objects of this type have parameters, such as coordinates, size, font, visibility, and lists of subordinate objects. This class also contains methods which allow saving/restoring of object state to/from stream.

```
TfrxComponent = class(TComponent)
protected
  procedure SetParent(AParent: TfrxComponent); virtual;
  procedure SetLeft(Value: Extended); virtual;
  procedure SetTop(Value: Extended); virtual;
  procedure SetWidth(Value: Extended); virtual;
  procedure SetHeight(Value: Extended); virtual;
  procedure SetFont(Value: TFont); virtual;
  procedure SetParentFont(Value: Boolean); virtual;
  procedure SetVisible(Value: Boolean); virtual;
  procedure FontChanged(Sender: TObject); virtual;
public
  constructor Create(AOwner: TComponent); override;
  procedure Assign(Source: TPersistent); override;
  procedure Clear; virtual;
  procedure CreateUniqueName;
  procedure LoadFromStream(Stream: TStream); virtual;
  procedure SaveToStream(Stream: TStream); virtual;
  procedure SetBounds(ALeft, ATop, AWidth, AHeight: Extended);
  function FindObject(const AName: String): TfrxComponent;
  class function GetDescription: String; virtual;
  property Objects: TList readonly;
  property AllObjects: TList readonly;
  property Parent: TfrxComponent;
  property Page: TfrxPage readonly;
  property Report: TfrxReport readonly;
  property IsDesigning: Boolean;
  property IsLoading: Boolean;
  property IsPrinting: Boolean;
  property BaseName: String;
  property Left: Extended;
  property Top: Extended;
  property Width: Extended;
  property Height: Extended;
  property AbsLeft: Extended readonly;
  property AbsTop: Extended readonly;
  property Font: TFont;
  property ParentFont: Boolean;
  property Restrictions: TfrxRestrictions;
  property Visible: Boolean;
end;
```

- `Clear` – clears object contents and deletes all its child objects.
- `CreateUniqueName` – creates unique name for object placed into report.
- `LoadFromStream` – loads object contents and all its child objects from stream.
- `SaveToStream` – saves object to stream.
- `SetBounds` – set object coordinates and size
- `FindObject` – searches for object with specified name among child objects.
- `GetDescription` – returns object's description.

The following methods are called when modifying corresponding properties. If additional handling is needed, you can override them:

- `SetParent`
- `SetLeft`

- `SetTop`
- `SetWidth`
- `SetHeight`
- `SetFont`
- `SetParentFont`
- `SetVisible`
- `FontChanged`

The following properties are defined in `TfrxComponent` class:

- `Objects` – list of child objects;
- `AllObjects` – list of all subordinate objects;
- `Parent` – link to parent object;
- `Page` – link to report page, which object belongs to;
- `Report` – link to report, which object belongs to;
- `IsDesigning` – "True," if designer is running;
- `IsLoading` – "True," if object is being loaded from stream;
- `IsPrinting` – "True", if object is being printed out;
- `BaseName` – object basic name. This value is used in `CreateUniqueName` method;
- `Left` – object X coordinate (relatively to parent);
- `Top` - object Y coordinate (relatively to parent);
- `Width` – object width;
- `Height` – object height;
- `AbsLeft` – X object absolute coordinate;
- `AbsTop` – Y object absolute coordinate;
- `Font` – object font;
- `ParentFont` – if "True," then uses parent object font settings;
- `Restrictions` – set of flags, which restrict one or another object operation;
- `Visible` – object visibility.

The next basic class is `TfrxReportComponent`. Objects of this type can be placed into report design. This class contains `Draw` method for object drawing as well as `BeforePrint` / `GetData` / `AfterPrint` methods, which are called on report running.

```
TfrxReportComponent = class(TfrxComponent)
public
  procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual; abstract;
  procedure BeforePrint; virtual;
  procedure GetData; virtual;
  procedure AfterPrint; virtual;
  function GetComponentText: String; virtual;
  property OnAfterPrint: TfrxNotifyEvent;
  property OnBeforePrint: TfrxNotifyEvent;
end;
```

Draw method is called on object drawing. Parameters are the following:

- Canvas – canvas;
- Scale – scale by X-axis and Y-axis;
- Offset – offset relatively canvas edges.

`BeforePrint` method is called right before object handling (during report building process). This method saves object state.

`GetData` method is called to load data into object.

`AfterPrint` is called after object handling. This method restores object state.

`TfrxDialogComponent` class is basic one for writing non-visual components, which can be placed to dialogue form in report.

```
TfrxDialogComponent = class(TfrxReportComponent)
public
  property Bitmap: TBitmap;
  property Component: TComponent;
published
  property Left;
  property Top;
end;
```

`TfrxDialogControl` class is basic one for writing common control, which can be placed on a dialogue form in report. This class contains a large number of general properties and events shared by most common controls.

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
end;
```

When writing your own custom control element, you should inherit from this class, transfer required properties to “published” section, and then specify new properties for your common control. Control element writing will be discussed in detail in the next chapter.

`TfrxView` class is basic one for most components, which can be placed on the report design page. Objects of this type have parameters such as Frame and Fill, and also can be connected to a data source. Most FastReport standard objects are inherited from this class.

```
TfrxView = class(TfrxReportComponent)
protected
  FX, FY, FX1, FY1, FDX, FDY, FFrameWidth: Integer;
  FScaleX, FScaleY: Extended;
  FCanvas: TCanvas;
  procedure BeginDraw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
  procedure DrawBackground;
  procedure DrawFrame;
  procedure DrawLine(x, y, x1, y1, w: Integer);
public
  function IsDataField: Boolean;
  property BrushStyle: TBrushStyle;
  property Color: TColor;
  property DataField: String;
  property DataSet: TfrxDataSet;
  property Frame: TfrxFrame;
published
  property Align: TfrxAlign;
  property Printable: Boolean;
  property ShiftMode: TfrxShiftMode;
  property TagStr: String;
  property Left;
  property Top;
  property Width;
  property Height;
  property Restrictions;
  property Visible;
  property OnAfterPrint;
  property OnBeforePrint;
end;
```

The following methods are defined in this class:

- `BeginDraw` - method is called from `Draw` method and calculates integer-valued coordinates and drawing area sizes. Calculated values are presented as `FX`, `FY`, `FX1`, `FY1`, `FDX`, and `FDY` variables. Frame width (it is placed in `FFrameWidth`) is also calculated;
- `DrawBackground` - draws object background;
- `DrawFrame` - draws object frame;
- `DrawLine` – draws line with specified coordinates and width;
- `IsDataField` returns "True," if `DataSet` and `DataField` properties contain nonempty values.

One can refer to the following properties after calling `BeginDraw` method:

- `FX`, `FY`, `FX1`, `FY1`, `FDX`, `FDY`, `FFrameWidth` are object frame coordinates, sizes and width calculated according to Scale and Offset parameters;
- `FScaleX`, `FScaleY` are scales, which are copies of ScaleX and ScaleY parameters from `Draw` method;
- `FCanvas` is canvas, which is a copy of Canvas parameter from `Draw` method.

Following properties, which are general for most report objects, are defined in this class:

- `BrushStyle` – object filling style;
- `Color` – object filling color;
- `DataField` - data field name, which object is connected to;

- `DataSet` - data source;
- `Frame` – object frame;
- `Align` - object aligning relatively to its parent;
- `Printable` – defines whether given object should be printed out;
- `ShiftMode` is object shifting mode in cases when stretchable object is placed over given one;
- `TagStr` - field for user information storage.

`TfrxStretcheable` class is basic one for writing components, which modify their height depending on data placed in it.

```
TfrxStretcheable = class(TfrxView)
public
  function CalcHeight: Extended; virtual;
  function DrawPart: Extended; virtual;
  procedure InitPart; virtual;
published
  property StretchMode: TfrxStretchMode;
end;
```

Objects of given class can be stretched, and also "broken" into pieces in cases when object does not find room on output page. At the same time, object is displayed piecemeal until all its data is displayed.

Following methods are defined in this class:

- `CalcHeight` is to calculate and return object height according to data placed in it;
- `InitPart` is called before object splitting;
- `DrawPart` redraws next data chunk placed in object. “Return value” is value of unused space where it was impossible to display data.

Custom Report Components Writing

FastReport has a great number of components, which can be placed on a report design page. They are: text, picture, line, geometrical figure, OLE, Rich, bar code, diagram etc. You can also write your own custom component, and then attach it to FastReport.

In FastReport several classes are defined, from which components are inherited. For more details, see "FastReport Classes Hierarchy" chapter. The `TfrxView` class is of primary interest to us, since most report components are inherited from it.

One should have at least the `Draw` method defined in the `TfrxReportComponent` basic class.

```
procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
```

This method is called when component is painted in designer, in preview window, and during output printing. `TfrxView` overrides this method for drawing object frame and background. This method should draw component contents on "Canvas" drawing surface. Object coordinates and sizes are stored in `AbsLeft`, `AbsTop`, `Width` and `Height` properties accordingly.

ScaleX and ScaleY parameters define object scaling in X-axis and Y-axis respectively. These parameters are equal 1 at 100% zoom and can vary, if user modifies zooming either in designer or in preview window. OffsetX and OffsetY parameters point shifting by X-axis and Y-axis. Thus, when taking these parameters into account, the upper left corner coordinate will be as follows:

```
X := Round(AbsLeft * ScaleX + OffsetX);
```

To simplify operations with coordinates, `BeginDraw` method (which has parameters similar to `Draw` method) is defined in `TfrxView` class

```
procedure BeginDraw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); virtual;
```

It should be called in the first line of `Draw` method. This method performs coordinates transformation into `FX`, `FY`, `FX1`, `FY1`, `FDX`, `FDY`, `FFrameWidth` integer values, which can be later used in TCanvas methods. This method also copies Canvas, ScaleX, and ScaleY values into `FCanvas`, `FScaleX`, `FScaleY` variables to which one can refer from any class method.

There are also two methods for drawing backgrounds for and frames of objects in `TfrxView` class.

```
    procedure DrawBackground;
    procedure DrawFrame;
```

`BeginDraw` method should be called before calling these methods.

Let us examine creating a component which will display an arrow.

```
type
  TfrxArrowView = class(TfrxView)
  public
    { we should override only two methods }
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
    class function GetDescription: String; override;
  published
    { Place required properties into the published section }
    property BrushStyle;
    property Color;
    property Frame;
  end;

class function TfrxArrowView.GetDescription: String;
begin
  { component description will be displayed next to its icon in toolbar }
  Result := 'Arrow object';
end;

procedure TfrxArrowView.Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended);
begin
  { call this method to perform coordinates transformation }
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  with Canvas do
  begin
    { set colors }
    Brush.Color := Color;
    Brush.Style := BrushStyle;
    Pen.Width := FFrameWidth;
    Pen.Color := Frame.Color;
    { draw arrow }
    Polygon(
      [Point(FX, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY),
      Point(FX1, FY + FDY div 2),
      Point(FX + FDX * 38 div 60, FY1),
      Point(FX + FDX * 38 div 60, FY + FDY * 3 div 4),
      Point(FX, FY + FDY * 3 div 4)]);
  end;
end;

{ registration }
var
  Bmp: TBitmap;
initialization
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(hInstance, 'frxArrowView');
  frxObjects.RegisterObject(TfrxArrowView, Bmp);

finalization
  { delete component from list of available ones }
  frxObjects.Unregister(TfrxArrowView);
  Bmp.Free;

end.
```

To create a component which displays any data from a DB one should transfer `DataSet`, `DataField` properties into the “published” section, and then override `GetData` method. Let us examine it using the `TfrxCheckBoxView` standard component as an example.

This component can be connected to a DB field via the `DataSet` and `DataField` properties, which are declared in `TfrxView` basic class. In addition, this component has the `Expression` property, into which an expression can be placed. As soon as it is calculated, result will be placed into `Checked` property. This component displays a cross, if

`Checked` property equals “True.” Below you can see the component’s initial declaration text (its most important parts).

```
  TfrxCheckBoxView = class(TfrxView)
  private
    FChecked: Boolean;
    FExpression: String;
    procedure DrawCheck(ARect: TRect);
  public
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
    procedure GetData; override;
  published
    property Checked: Boolean read FChecked write FChecked default True;
    property DataField;
    property DataSet;
    property Expression: String read FExpression write FExpression;
  end;

procedure TfrxCheckBoxView.Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended);
begin
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  DrawBackground;
  DrawCheck(Rect(FX, FY, FX1, FY1));
  DrawFrame;
end;

procedure TfrxCheckBoxView.GetData;
begin
  inherited;
  if IsDataField then
    FChecked := DataSet.Value[DataField]
  else if FExpression <> '' then
    FChecked := Report.Calc(FExpression);
end;
```

Custom Common Controls Writing

FastReport contains a set of common controls, which can be placed on dialogue forms inside the report. They are as follows:

```
TfrxLabelControl
TfrxEditControl
TfrxMemoControl
TfrxButtonControl
TfrxCheckBoxControl
TfrxRadioButtonControl
TfrxListBoxControl
TfrxComboBoxControl
TfrxDateEditControl
TfrxImageControl
TfrxBevelControl
TfrxPanelControl
TfrxGroupBoxControl
TfrxBitBtnControl
TfrxSpeedButtonControl
TfrxMaskEditControl
TfrxCheckListBoxControl
```

These control elements correspond to Delphi component palette standard controls. If standard functionality does not satisfy you, you can create your own common control and use it in your reports.

Basic class for all common controls is `TfrxDialogControl` class declared in frxClass file:

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  class function GetDescription: String; virtual;
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
end;
```

To create your own control element, you should inherit from this class and override at least the constructor and `GetDescription` methods. It will be necessary to create common control and initialize it via `InitControl` method in constructor. `GetDescription` method is to return common control description. As you can see from `TfrxDialogControl` class description, it already contains huge number of properties and methods in public section. You need to transfer any necessary properties/events into "published" section of your common control, and also to create new properties, which are typical for your element.

Common control registration and deleting is performed via `frxObjects` global object methods declared in frxDsgnIntf file:

```
frxObjects.RegisterObject(ClassRef: TfrxComponentClass; ButtonBmp: TBitmap);
frxObjects.Unregister(ClassRef: TfrxComponentClass);
```

During registration you should specify control class name and its picture. ButtonBmp size should be 16x16 pixels.

Let us examine the example of the common control, which has simplified functionality of the standard Delphi `TBitBtn` control, for example.

```
uses frxClass, frxDsgnIntf, Buttons;

type
  TfrxBitBtnControl = class(TfrxDialogControl)
  private
    FButton: TBitBtn;
    procedure SetKind(const Value: TBitBtnKind);
    function GetKind: TBitBtnKind;
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; override;
    property Button: TBitBtn read FButton;
  published
    { add new properties }
    property Kind: TBitBtnKind read GetKind write SetKind default bkCustom;
    { these properties are already declared in  parent class }
    property Caption;
    property OnClick;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;

constructor TfrxBitBtnControl.Create(AOwner: TComponent);
begin
  { default constructor }
  inherited;
  { create  required common control }
  FButton := TBitBtn.Create(nil);
  FButton.Caption := 'BitBtn';
  { initialize it }
  InitControl(FButton);
  { it will have such size by default }
  Width := 75;
  Height := 25;
end;
```

```
class function TfrxBitBtnControl.GetDescription: String;
begin
  Result := 'BitBtn control';
end;

procedure TfrxBitBtnControl.SetKind(const Value: TBitBtnKind);
begin
  FButton.Kind := Value;
end;

function TfrxBitBtnControl.GetKind: TBitBtnKind;
begin
  Result := FButton.Kind;
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  {Load picture from resource. Of course, you should beforehand place it there.}
  Bmp.LoadFromResourceName(hInstance, 'frxBitBtnControl');
  frxObjects.RegisterObject(TfrxBitBtnControl, Bmp);

finalization
  frxObjects.Unregister(TfrxBitBtnControl);
  Bmp.Free;

end.
```

Event Handler Description

What should be done, if it is necessary to define a new event handler, which does not belong to the basic class? Let us examine it using the `TfrxEditControl` common control as an example:

```
  TfrxEditControl = class(TfrxDialogControl)
  private
    FEdit: TEdit;
    { new event }
    FOnChange: TfrxNotifyEvent;
    procedure DoOnChange(Sender: TObject);
    ...
  public
    constructor Create(AOwner: TComponent); override;
    ...
  published
    { new event }
    property OnChange: TfrxNotifyEvent read FOnChange write FOnChange;
    ...
  end;

constructor TfrxEditControl.Create(AOwner: TComponent);
begin
  ...
  { connect our handler }
  FEdit.OnChange := DoOnChange;
  InitControl(FEdit);
  ...
end;

procedure TfrxEditControl.DoOnChange(Sender: TObject);
begin
  { call event handler }
  if Report <> nil then
    Report.DoNotifyEvent(Sender, FOnChange);
end;
```

It is important to notice that the event handler in FastReport is a procedure declared in report script. The string containing its name will be a link to the handler. That is why, for example, unlike the Delphi `TNotifyEvent` type, which is method address, handler type, in FastReport it is string (`TfrxNotifyEvent` type is declared as String[63]).

Component Registration in Script System

To refer to our component from script, it is necessary to register its class, its properties, and methods in the script system. Register code, according to FastReport requirements, it may be placed in a file with the same name as component code file, adding RTTI suffix (for example, frxBitBtnRTTI.pas in our case). See more about classes

registration, their methods and properties in FastScript script library documentation.

```
uses fs_iinterpreter, frxBitBtn, frxClassRTTI;

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;

constructor TFunctions.Create(AScript: TfsScript);
begin
  inherited Create(AScript);
  with AScript do
  begin
    { register class, and then define its parent }
    AddClass(TfrxBitBtnControl, 'TfrxDialogControl');
    { if there are several common controls in your unit, they can be registered right here }
    { for example, AddClass(TfrxAnotherControl, 'TfrxDialogControl'); }
  end;
end;

initialization
  fsRTTIModules.Add(TFunctions);

end.
```

Component Editor Writing

Any common control editor (it can be called from element context menu or by double-clicking) creates OnClick blank event handler by default. This behavior can be replaced by writing a custom editor. In addition, the editor allows adding your own items to component context menu.

Basic class for all editors is declared in frxDsgnIntf file:

```
TfrxComponentEditor = class(TObject)
 protected
   function AddItem(Caption: String; Tag: Integer;
     Checked: Boolean = False): TMenuItem;
 public
   function Edit: Boolean; virtual;
   function HasEditor: Boolean; virtual;
   function Execute(Tag: Integer; Checked: Boolean): Boolean; virtual;
   procedure GetMenuItems; virtual;
   property Component: TfrxComponent readonly;
   property Designer: TfrxCustomDesigner readonly;
 end;
```

If your editor does not create its own items in contextual menu, you will need to override two methods, i.e. `Edit` and `HasEditor`. The first method performs necessary actions (for example, displays dialogue box) and returns “True,” if component state was modified. `HasEditor` method should return “True” if your component has editor. If it either returns “False” or you do not override this method, editor will not be called. This becomes necessary, if your component does not have an editor and you wish to add items into component context menu.

If editor adds items into context menu, you should override `GetMenuItems` (in this method, you can create a menu with help of calling `AddItem` function) and `Execute` (this method is called, when you select one of your items in component menu; response to selected menu item should be described here) methods.

Editor registration is performed via procedure described in “frxDsgnIntf” file:

```
frxComponentEditors.Register(ComponentClass: TfrxComponentClass; ComponentEditor:
TfrxComponentEditorClass);
```

The first parameter is class name, for which editor is to be created. The second parameter is editor class name.

Let us examine simple editor for our common control, which will display a window with our element name and add "Enabled" and "Visible" items to element context menu (when items are selected, `Enabled` and `Visible` properties will change). Editor code, according to FastReport requirements, can be placed in a file having the same name as file with the component’s code, adding Editor suffix (for example, frxBitBtnEditor.pas in our case).

```
uses frxClass, frxDsgnIntf, frxBitBtn;

type
  TfrxBitBtnEditor = class(TfrxComponentEditor)
  public
    function Edit: Boolean; override;
    function HasEditor: Boolean; override;
    function Execute(Tag: Integer; Checked: Boolean): Boolean; override;
    procedure GetMenuItems; override;
  end;

function TfrxBitBtnEditor.Edit: Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := False;
  {  Component property is edited component. In this case, it is TfrxBitBtnControl }
  c := TfrxBitBtnControl(Component);
  ShowMessage('This is ' + c.Name);
end;

function TfrxBitBtnEditor.HasEditor: Boolean;
begin
  Result := True;
end;

function TfrxBitBtnEditor.Execute(Tag: Integer; Checked: Boolean): Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := True;
  c := TfrxBitBtnControl(Component);
  if Tag = 1 then
    c.Enabled := Checked
  else if Tag = 2 then
    c.Visible := Checked;
end;

procedure TfrxBitBtnEditor.GetMenuItems;
var
  c: TfrxBitBtnControl;
begin
  c := TfrxBitBtnControl(Component);
  {  AddItem method parameters: menu item name, its tag and Checked/Unchecked condition }
  AddItem('Enabled', 1, c.Enabled);
  AddItem('Visible', 2, c.Visible);
end;

initialization
  frxComponentEditors.Register(TfrxBitBtnControl, TfrxBitBtnEditor);

end.
```

Property Editor Writing

When you select a component in the designer, its properties are displayed in object inspector. You can create your own editor for any property. “Font” property standard editor can exemplify that: if this property is selected, the `...` button appears in right part of line; call standard "font properties" dialogue box by clicking this button. One more example is “Color” property editor. It shows standard colors and color specimens names in drop-down list.

Base class for all property editors is described in “frxDsgnIntf” unit:

```
  TfrxPropertyEditor = class(TObject)
  protected
    procedure GetStrProc(const s: String);
    function GetFloatValue: Extended;
    function GetOrdValue: Integer;
    function GetStrValue: String;
    function GetVarValue: Variant;
    procedure SetFloatValue(Value: Extended);
    procedure SetOrdValue(Value: Integer);
    procedure SetStrValue(const Value: String);
    procedure SetVarValue(Value: Variant);
  public
    constructor Create(Designer: TfrxCustomDesigner); virtual;
    destructor Destroy; override;
    function Edit: Boolean; virtual;
    function GetAttributes: TfrxPropertyAttributes; virtual;
    function GetExtraLBSize: Integer; virtual;
    function GetValue: String; virtual;
    procedure GetValues; virtual;
    procedure SetValue(const Value: String); virtual;
    procedure OnDrawLBItem(Control: TWinControl; Index: Integer; ARect: TRect; State: TOwnerDrawState);
virtual;
    procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); virtual;
    property Component: TPersistent readonly;
    property frComponent: TfrxComponent readonly;
    property Designer: TfrxCustomDesigner readonly;
    property ItemHeight: Integer;
    property PropInfo: PPropInfo readonly;
    property Value: String;
    property Values: TStrings readonly;
  end;
```

You also can inherit from any of the following classes which themselves realize some basic functionality for working with properties of corresponding types:

```
  TfrxIntegerProperty = class(TfrxPropertyEditor)
  TfrxFloatProperty = class(TfrxPropertyEditor)
  TfrxCharProperty = class(TfrxPropertyEditor)
  TfrxStringProperty = class(TfrxPropertyEditor)
  TfrxEnumProperty = class(TfrxPropertyEditor)
  TfrxClassProperty = class(TfrxPropertyEditor)
  TfrxComponentProperty = class(TfrxPropertyEditor)
```

Several properties are defined in this class:

- `Component` - link to parent component (not to property itself!), to which the given property belongs;

- `frComponent` - the same, but casted to `TfrxComponent` type (for convenience in some cases);
- `Designer` – link to report designer;
- `ItemHeight` - item height, in which property is displayed. It can be useful in OnDrawXXX;
- `PropInfo` - link to PPropInfo structure, which contains information about edited property;
- `Value` - property value displayed as string;
- `Values` - list of values. This property is to be filled in `GetValue` method, if "paValueList" attribute is defined (see below).

The following methods are service ones. They can be used to get or set edited property value.

```
function GetFloatValue: Extended;
function GetOrdValue: Integer;
function GetStrValue: String;
function GetVarValue: Variant;
procedure SetFloatValue(Value: Extended);
procedure SetOrdValue(Value: Integer);
procedure SetStrValue(const Value: String);
procedure SetVarValue(Value: Variant);
```

You should use methods, which correspond to property type. Thus, use `GetOrdValue` and `SetOrdValue` methods, if property is of "Integer" type. These methods are also used for working with property of `TObject` type, since such property contains 32-bit object address. In this case, it is sufficient to do cast of the following type, for example:

```
MyFont := TFont(GetOrdValue)
```

To create your own editor, it is necessary to inherit from basic class and override one or several methods defined in public section (this depends on property type and functionality you wish to realize). One of methods you surely have to override is `GetAttributes` method. This method is to return set of property attributes. Attributes are defined in the following way:

```
TfrxPropertyAttribute = (paValueList, paSortList, paDialog, paMultiSelect, paSubProperties, paReadOnly,
paOwnerDraw);
TfrxPropertyAttributes = set of TfrxPropertyAttribute;
```

Attribute assignment is achieved as follows:

- paValueList - property represents dropping down list of values. (This function is exemplified in "Color" property). If this attribute is present, `GetValues` method should be overridden;
- paSortList - sorts list elements. It is used together with paValueList;
- paDialog - property has editor. If this attribute is present, the `...` button is displayed in the right part of editing line. Edit method is called on by clicking on it;
- paMultiSelect – allows given property editing in some objects selected at the same time. Some properties (such as "Name", etc) do not have this attribute;
- paSubProperties - property is object of `TPersistent` type and has its own properties, which are also should be displayed. (This function is exemplified in "Font" property);

- paReadOnly - it is impossible to modify value in editor line. Some properties, being "Class" or "Set" types, possess this attribute;
- paOwnerDraw - property value drawing is performed via `OnDrawItem` method. If "paValueList" attribute is defined, then drop-down list drawing is performed via `OnDrawLBItem` method.

Edit method is called in two cases: either by selecting property, by double-clicking its value, or (if property has paDialog attribute) by clicking the `...` button. This method should return "True," if property value was modified.

`GetValue` method should return property value as string (it will be displayed in object inspector). If you inherit from `TfrxPropertyEditor` basic class, it is necessary to override this method.

`SetValue` method is to set property value transferred as string. If you inherit from `TfrxPropertyEditor` basic class, it is necessary to override this method.

`GetValues` method should be overridden in case you defined "paValueList" attribute. This method should fill `Values` property with values.

The following three methods allow performing manual property value drawing (Color property editor works in the same way). These methods are called, if you define "paOwnerDraw" attribute.

`OnDrawItem` method is called when drawing property value in object inspector (when property is not selected; otherwise its value is simply displayed in editing line). For example, Color property editor draws rectangle, filled with color according to value, to the left of property value.

`GetExtraLBSize` method is called in case you defined "paValueList" attribute. This method returns number of pixels, by which "Drop-Down List" width should be adjusted in order to find room for displayed picture. By default, this method returns value corresponding to cell height for property enveloping. If you need to deduce picture, with width larger than its height, the given method should be overridden.

`OnDrawLBItem` method is called when drawing string in drop-down list, if you defined paValueList attribute. In fact, this method is `TListBox.OnDrawItem` event handler and has the same set of parameters.

Property editor registration is performed via procedure described in frxDsgnIntf file:

```
procedure frxPropertyEditors.Register(PropertyType: PTypeInfo; ComponentClass: TClass; const
PropertyName: String; EditorClass: TfrxPropertyEditorClass);
```

- PropertyType - information about property type, transferred via "TypeInfo" system function, for example TypeInfo(String);
- ComponentClass – component name, with property you want to edit (may be nil);
- PropertyName - name of property you want to edit (may be blank string);
- EditorClass - property editor name

It is necessary to specify "PropertyType" parameter only. "ComponentClass" and/or "PropertyName" parameters may be blank. This allows to register editor either to any property of PropertyType type, to any property of concrete ComponentClass components and its successors, or to PropertyName concrete property of concrete component (or any component, if ComponentClass parameter is blank).

Let us examine three property editors examples. Editor code, according to FastReport requirements, can be placed in a file having the same name as file with code of the component, and adding the Editor suffix.

```
{ TFont property editor displays editor button(...) }
{ inherit from ClassProperty }

type
 TfrxFontProperty = class(TfrxClassProperty)
  public
    function Edit: Boolean; override;
    function GetAttributes: TfrxPropertyAttributes; override;
  end;

function TfrxFontProperty.GetAttributes: TfrxPropertyAttributes;
begin
  {  property has nested properties and  editor. It cannot be edited manually }
  Result := [paMultiSelect, paDialog, paSubProperties, paReadOnly];
end;

function TfrxFontProperty.Edit: Boolean;
var
  FontDialog: TFontDialog;
begin
  { create standard dialogue }
  FontDialog := TFontDialog.Create(Application);
  try
    { take  property value }
    FontDialog.Font := TFont(GetOrdValue);
    FontDialog.Options := FontDialog.Options + [fdForceFontExist];
    { display dialogue }
    Result := FontDialog.Execute;
    { bind new value }
    if Result then
      SetOrdValue(Integer(FontDialog.Font));
  finally
    FontDialog.Free;
  end;
end;

{ registration }

frxPropertyEditors.Register(TypeInfo(TFont), nil, '', TfrxFontProperty);
```

```
{ TFont.Name property editor displays  drop-down list of available fonts }
{ inherit from StringProperty, as property is of string type }

type
  TfrxFontNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

function TfrxFontNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxFontNameProperty.GetValues;
begin
  Values.Assign(Screen.Fonts);
end;

{ registration }

frxPropertyEditors.Register(TypeInfo(String), TFont, 'Name', TfrxFontNameProperty);
```

```
{ TPen.Style property editor displays picture, which is pattern of selected style }

type
  TfrxPenStyleProperty = class(TfrxEnumProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    function GetExtraLBSize: Integer; override;
    procedure OnDrawLBItem(Control: TWinControl; Index: Integer;
      ARect: TRect; State: TOwnerDrawState); override;
    procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); override;
  end;

function TfrxPenStyleProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList, paOwnerDraw];
end;

{ method draws thick horizontal line with selected style }
procedure HLine(Canvas: TCanvas; X, Y, DX: Integer);
var
  i: Integer;
begin
  with Canvas do
  begin
    Pen.Color := clBlack;
    for i := 0 to 1 do
    begin
      MoveTo(X, Y - 1 + i);
      LineTo(X + DX, Y - 1 + i);
    end;
  end;
end;

{ drawing drop-down list }
procedure TfrxPenStyleProperty.OnDrawLBItem(Control: TWinControl; Index: Integer; ARect: TRect; State:
TOwnerDrawState);
begin
  with TListBox(Control), TListBox(Control).Canvas do
  begin
    FillRect(ARect);
    TextOut(ARect.Left + 40, ARect.Top + 1, TListBox(Control).Items[Index]);
    Pen.Color := clGray;
    Brush.Color := clWhite;
    Rectangle(ARect.Left + 2, ARect.Top + 2, ARect.Left + 36, ARect.Bottom - 2);
    Pen.Style := TPenStyle(Index);
    HLine(TListBox(Control).Canvas, ARect.Left + 3, ARect.Top + (ARect.Bottom - ARect.Top) div 2, 32);
    Pen.Style := psSolid;
  end;
end;

{ drawing property value }
procedure TfrxPenStyleProperty.OnDrawItem(Canvas: TCanvas; ARect: TRect);
begin
  with Canvas do
  begin
    TextOut(ARect.Left + 38, ARect.Top, Value);
    Pen.Color := clGray;
    Brush.Color := clWhite;
    Rectangle(ARect.Left, ARect.Top + 1, ARect.Left + 34, ARect.Bottom - 4);
    Pen.Color := clBlack;
    Pen.Style := TPenStyle(GetOrdValue);
    HLine(Canvas, ARect.Left + 1, ARect.Top + (ARect.Bottom - ARect.Top) div 2 - 1, 32);
    Pen.Style := psSolid;
  end;
end;
```

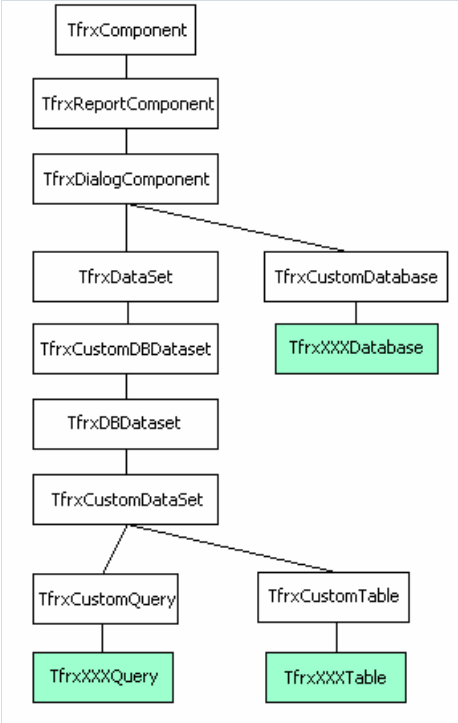
```
{ return  picture width }
function TfrxPenStyleProperty.GetExtraLBSize: Integer;
begin
  Result := 36;
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(TPenStyle), TPen, 'Style', TfrxPenStyleProperty);
```

Custom DB Engines Writing

FastReport allows building reports not only on the basis of data defined in application. You can define your own data sources (connections to DB, queries) right in report as well. FastReport is supplied with engines for ADO, BDE, IBX, DBX, FIB. You can create your own engine, and then connect it to FastReport.

The illustration below shows classes hierarchy intended for creating DB engines. New engine components are highlighted with green color.



As you can see, a standard set of DB engine components includes Database, Table and Query. You can create all these components or some of them (for example, many DB have no component of Table type). You can also create components, which are not included in the standard set (for example, StoredProc).

Let us examine the basic classes in detail.

`TfrxDialogComponent` is the basic class for all non-visual components, which can be placed on a FastReport report design dialogue form. There are not any important properties or methods defined in it.

`TfrxCustomDatabase` class is basic class for DB components of “Database” type.

```
TfrxCustomDatabase = class(TfrxDialogComponent)
protected
  procedure SetConnected(Value: Boolean); virtual;
  procedure SetDatabaseName(const Value: String); virtual;
  procedure SetLoginPrompt(Value: Boolean); virtual;
  procedure SetParams(Value: TStrings); virtual;
  function GetConnected: Boolean; virtual;
  function GetDatabaseName: String; virtual;
  function GetLoginPrompt: Boolean; virtual;
  function GetParams: TStrings; virtual;
public
  procedure SetLogin(const Login, Password: String); virtual;
  property Connected: Boolean read GetConnected write SetConnected default False;
  property DatabaseName: String read GetDatabaseName write SetDatabaseName;
  property LoginPrompt: Boolean read GetLoginPrompt write SetLoginPrompt default True;
  property Params: TStrings read GetParams write SetParams;
end;
```

The following properties are defined in this class:

- `Connected` – whether DB connection is active;
- `DatabaseName` – database name;
- `LoginPrompt` – whether to ask login when connecting to DB;
- `Params` – connection parameters.

From the given class a component of `TDatabase` type is inherited. For its creation it is necessary to override all virtual methods and place the necessary properties in the to published section. Also it is necessary to add properties specific for your component.

`TfrxDataset` , `TfrxCustomDBDataset` , `TfrxDBDataset` classes provide the functions of data access. FastReport core uses these components for navigation and addressing data entering fields. In this case they are part of common hierarchy and are of no interest to us.

`TfrxCustomDataSet` is a basic class of DB components derived from `TDataSet` . Components, inherited from this class are “Query,” “Table,” and “StoredProc” clones. As a matter of fact, this class wraps over `TDataSet` .

```
TfrxCustomDataset = class(TfrxDBDataSet)
protected
  procedure SetMaster(const Value: TDataSource); virtual;
  procedure SetMasterFields(const Value: String); virtual;
public
  property DataSet: TDataSet;
  property Fields: TFields readonly;
  property MasterFields: String;
  property Active: Boolean;
published
  property Filter: String;
  property Filtered: Boolean;
  property Master: TfrxDBDataSet;
end;
```

The following properties are defined in the class:

- `DataSet` is a link to buried object of `TDataSet` type;
- `Fields` is a link to DataSet.Fields;

- `Active` - whether data set is active;
- `Filter` - expression for filtering;
- `Filtered` – whether filtering is active;
- `Master` is a link to master dataset in master-detail relationship.
- `MasterFields` is list of fields like field1=field2. Used for master-detail relations.

`TfrxCustomTable` – basic class for DB components of Table type. Class covers component of Table class.

```
TfrxCustomTable = class(TfrxCustomDataset)
protected
  function GetIndexFieldNames: String; virtual;
  function GetIndexName: String; virtual;
  function GetTableName: String; virtual;
  procedure SetIndexFieldNames(const Value: String); virtual;
  procedure SetIndexName(const Value: String); virtual;
  procedure SetTableName(const Value: String); virtual;
published
  property MasterFields;
  property TableName: String read GetTableName write SetTableName;
  property IndexName: String read GetIndexName write SetIndexName;
  property IndexFieldNames: String read GetIndexFieldNames write SetIndexFieldNames;
end;
```

The following properties are defined in class:

- `TableName` – table name;
- `IndexName` – index name;
- `IndexFieldNames` – index field names.

Component of Table type is inherited from this class. For its creation it is necessary to define required properties, Database as usual. Also it is necessary to override virtual means from `TfrxCustomDataset`, `TfrxCustomTable` classes.

`TfrxCustomQuery` is basic class for DB components of "Query" type. This class is cover for Query type component.

```
TfrxCustomQuery = class(TfrxCustomDataset)
protected
  procedure SetSQL(Value: TStrings); virtual; abstract;
  function GetSQL: TStrings; virtual; abstract;
public
  procedure UpdateParams; virtual; abstract;
published
  property Params: TfrxParams;
  property SQL: TStrings;
end;
```

`SQL` and `Params` properties (which are general for all Query components) are defined in the class. Since different Query components have different parameters realization (for example, `TParams` and `TParameters`), "Params" property has `TfrxParams` type and is a wrapper for concrete parameters type.

The following methods are defined in this class:

- `SetSQL` is to set `SQL` component property of "Query" type;

- `GetSQL` is to get `SQL` component property of "Query" type;
- `UpdateParams` is to copy parameters values into component of Query type. If Query component parameters are of `TParams` type, copying is performed via `frxParamsToTParams` standard procedure.

Let us illustrate DB engine creation using the IBX example. Full engine original text can be found in SOURCE\IBX directory. Below are some quotations from source text with our comments.

IBX components around which we will build the wrapper are `TIBDatabase`, `TIBTable`, and `TIBQuery`. Accordingly, our components will be named `TfrxIBXDatabase`, `TfrxIBXTable` and `TfrxIBXQuery`.

Component for Delphi palette

`TfrxIBXComponents` is the first component we should create; it will be placed into FastReport component palette when registering engine (in the Delphi environment). As soon as this component is placed into a project, Delphi automatically adds link to our engine unit into "Uses" list. One should assign one more task in this component, i.e. to define `DefaultDatabase` property in it, which refers to existing connection to DB. By default, all `TfrxIBXTable` and `TfrxIBXQuery` components will refer to this connection. It is necessary to inherit component from `TfrxDBComponents` class:

```
  TfrxDBComponents = class(TComponent)
  public
    function GetDescription: String; virtual; abstract;
  end;
```

Description should be returned by one function only, for example "IBX Components". `TfrxIBXComponents` component implementation is as follows:

```
type
  TfrxIBXComponents = class(TfrxDBComponents)
  private
    FDefaultDatabase: TIBDatabase;
    FOldComponents: TfrxIBXComponents;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetDescription: String; override;
  published
    property DefaultDatabase: TIBDatabase read FDefaultDatabase write FDefaultDatabase;
  end;

var
  IBXComponents: TfrxIBXComponents;

constructor TfrxIBXComponents.Create(AOwner: TComponent);
begin
  inherited;
  FOldComponents := IBXComponents;
  IBXComponents := Self;
end;

destructor TfrxIBXComponents.Destroy;
begin
  if IBXComponents = Self then
    IBXComponents := FOldComponents;
  inherited;
end;

function TfrxIBXComponents.GetDescription: String;
begin
  Result := 'IBX';
end;
```

We define IBXComponents global variable, which will refer to `TfrxIBXComponents` component copy. If you place component into project several times (though it is senseless), you will nevertheless be able to save link to previous component and restore it after deleting component.

A link to connection to DB, which already exists in project, can be placed into `DefaultDatabase` property. The way we will write `TfrxIBXTable`, `TfrxIBXQuery` components allows them to use this connection by default (actually, this is what we need IBXComponents global variable for).

Database component

The following component is `TfrxIBXDatabase` one. It represents a wrapper over `TIBDatabase`.

```
  TfrxIBXDatabase = class(TfrxCustomDatabase)
  private
    FDatabase: TIBDatabase;
    FTransaction: TIBTransaction;
    function GetSQLDialect: Integer;
    procedure SetSQLDialect(const Value: Integer);
  protected
    procedure SetConnected(Value: Boolean); override;
    procedure SetDatabaseName(const Value: String); override;
    procedure SetLoginPrompt(Value: Boolean); override;
    procedure SetParams(Value: TStrings); override;
    function GetConnected: Boolean; override;
    function GetDatabaseName: String; override;
    function GetLoginPrompt: Boolean; override;
    function GetParams: TStrings; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetDescription: String; override;
    procedure SetLogin(const Login, Password: String); override;
    property Database: TIBDatabase read FDatabase;
  published
    { list TIBDatabase properties. Note – some properties are already exist in base class }
    property DatabaseName;
    property LoginPrompt;
    property Params;
    property SQLDialect: Integer read GetSQLDialect write SetSQLDialect;
    {  Connected property should be placed last! }
    property Connected;
  end;

constructor TfrxIBXDatabase.Create(AOwner: TComponent);
begin
  inherited;
  { create component – connection }
  FDatabase := TIBDatabase.Create(nil);
  { create component - transaction (specificity of  IBX) }
  FTransaction := TIBTransaction.Create(nil);
  FDatabase.DefaultTransaction := FTransaction;
  { do not forget this line! }
  Component := FDatabase;
end;

destructor TfrxIBXDatabase.Destroy;
begin
  { delete  transaction }
  FTransaction.Free;
  { connection will be deleted automatically in parent class }
  inherited;
end;

{ component description will be displayed next to icon in objects toolbar }
class function TfrxIBXDatabase.GetDescription: String;
begin
  Result := 'IBX Database';
end;

{ redirect component properties to  cover properties and vice versa }
function TfrxIBXDatabase.GetConnected: Boolean;
```

```
begin
  Result := FDatabase.Connected;
end;

function TfrxIBXDatabase.GetDatabaseName: String;
begin
  Result := FDatabase.DatabaseName;
end;

function TfrxIBXDatabase.GetLoginPrompt: Boolean;
begin
  Result := FDatabase.LoginPrompt;
end;

function TfrxIBXDatabase.GetParams: TStrings;
begin
  Result := FDatabase.Params;
end;

function TfrxIBXDatabase.GetSQLDialect: Integer;
begin
  Result := FDatabase.SQLDialect;
end;

procedure TfrxIBXDatabase.SetConnected(Value: Boolean);
begin
  FDatabase.Connected := Value;
  FTransaction.Active := Value;
end;

procedure TfrxIBXDatabase.SetDatabaseName(const Value: String);
begin
  FDatabase.DatabaseName := Value;
end;

procedure TfrxIBXDatabase.SetLoginPrompt(Value: Boolean);
begin
  FDatabase.LoginPrompt := Value;
end;

procedure TfrxIBXDatabase.SetParams(Value: TStrings);
begin
  FDatabase.Params := Value;
end;

procedure TfrxIBXDatabase.SetSQLDialect(const Value: Integer);
begin
  FDatabase.SQLDialect := Value;
end;

{ this method is used by DB connection wizard }
procedure TfrxIBXDatabase.SetLogin(const Login, Password: String);
begin
  Params.Text := 'user_name=' + Login + #13#10 + 'password=' + Password;
end;
```

As you can see, this is not that complicated. We create FDatabase: TIBDatabase object, and then define properties we want the designer to have. “Get” and “Set” methods are written for each property.

Table component

Next component is `TfrxIBXTable`. It inherits from `TfrxCustomDataSet` standard class. All basic functionality (operating with list of fields, master-detail, basic properties) is already realized in basic class. We only need to define properties, which are specific to the given component.

```
  TfrxIBXTable = class(TfrxCustomTable)
  private
    FDatabase: TfrxIBXDatabase;
    FTable: TIBTable;
    procedure SetDatabase(const Value: TfrxIBXDatabase);
  protected
    procedure Notification(AComponent: TComponent; Operation: TOperation); override;
    procedure SetMaster(const Value: TDataSource); override;
    procedure SetMasterFields(const Value: String); override;
    procedure SetIndexFieldNames(const Value: String); override;
    procedure SetIndexName(const Value: String); override;
    procedure SetTableName(const Value: String); override;
    function GetIndexFieldNames: String; override;
    function GetIndexName: String; override;
    function GetTableName: String; override;
  public
    constructor Create(AOwner: TComponent); override;
    constructor DesignCreate(AOwner: TComponent; Flags: Word); override;
    class function GetDescription: String; override;
    procedure BeforeStartReport; override;
    property Table: TIBTable read FTable;
  published
    property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
  end;

constructor TfrxIBXTable.Create(AOwner: TComponent);
begin
  { create component – table }
  FTable := TIBTable.Create(nil);
  { assign link to  DataSet property from  basic class – do not forget this string! }
  DataSet := FTable;
  { assign link to connection to DB by default }
  SetDatabase(nil);
  { after that basic constructor may be called in}
  inherited;
end;

{ this constructor is called at moment of adding components to report. It connects table to
TfrxIBXDatabase component automatically, if it is already present. }
constructor TfrxIBXTable.DesignCreate(AOwner: TComponent; Flags: Word);
var
  i: Integer;
  l: TList;
begin
  inherited;
  l := Report.AllObjects;
  for i := 0 to l.Count - 1 do
    if TObject(l[i]) is TfrxIBXDatabase then
    begin
      SetDatabase(TfrxIBXDatabase(l[i]));
      break;
    end;
end;

class function TfrxIBXTable.GetDescription: String;
begin
```

```
  Result := 'IBX Table';
end;

{ trace TfrxIBXDatabase component deleting. We address this component in FDatabase property. Otherwise we
can get error. }
procedure TfrxIBXTable.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FDatabase) then
    SetDatabase(nil);
end;

procedure TfrxIBXTable.SetDatabase(const Value: TfrxIBXDatabase);
begin
  {  Database property of TfrxIBXDatabase type, and not of TIBDatabase one! }
  FDatabase := Value;
  { if value <> nil, connect table to selected component }
  if Value <> nil then
    FTable.Database := Value.Database
  { otherwise, try to connect to DB by default, defined in TfrxIBXComponents component }
  else if IBXComponents <> nil then
    FTable.Database := IBXComponents.DefaultDatabase
  { if there were no TfrxIBXComponents for some reason, reset to nil }
  else
    FTable.Database := nil;
  { if connection was a success DBConnected flag should be put }
  DBConnected := FTable.Database <> nil;
end;

function TfrxIBXTable.GetIndexFieldNames: String;
begin
  Result := FTable.IndexFieldNames;
end;

function TfrxIBXTable.GetIndexName: String;
begin
  Result := FTable.IndexName;
end;

function TfrxIBXTable.GetTableName: String;
begin
  Result := FTable.TableName;
end;

procedure TfrxIBXTable.SetIndexFieldNames(const Value: String);
begin
  FTable.IndexFieldNames := Value;
end;

procedure TfrxIBXTable.SetIndexName(const Value: String);
begin
  FTable.IndexName := Value;
end;

procedure TfrxIBXTable.SetTableName(const Value: String);
begin
  FTable.TableName := Value;
end;

procedure TfrxIBXTable.SetMaster(const Value: TDataSource);
begin
  FTable.MasterSource := Value;
end;

procedure TfrxIBXTable.SetMasterFields(const Value: String);
begin
  FTable.MasterFields := Value;
  FTable.IndexFieldNames := Value;
end;
```

```
{ we need to implement this method in some cases }
procedure TfrxIBXTable.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

Query component

Finally, let's examine the last component, `TfrxIBXQuery`. It inherits from `TfrxCustomQuery` basic class, in which necessary properties are already defined. We only need to define `Database` property and override `SetMaster` method. Other methods realization is similar to `TfrxIBXTable` component.

```
  TfrxIBXQuery = class(TfrxCustomQuery)
  private
    FDatabase: TfrxIBXDatabase;
    FQuery: TIBQuery;
    procedure SetDatabase(const Value: TfrxIBXDatabase);
  protected
    procedure Notification(AComponent: TComponent; Operation: TOperation); override;
    procedure SetMaster(const Value: TDataSource); override;
    procedure SetSQL(Value: TStrings); override;
    function GetSQL: TStrings; override;
  public
    constructor Create(AOwner: TComponent); override;
    constructor DesignCreate(AOwner: TComponent; Flags: Word); override;
    class function GetDescription: String; override;
    procedure BeforeStartReport; override;
    procedure UpdateParams; override;
    property Query: TIBQuery read FQuery;
  published
    property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
  end;

constructor TfrxIBXQuery.Create(AOwner: TComponent);
begin
  { create component - query }
  FQuery := TIBQuery.Create(nil);
  { assign link to it to DataSet property from basic class - do not forget this line! }
  Dataset := FQuery;
  { assign link to connection to DB by default }
  SetDatabase(nil);
  { after that basic constructor may be called in }
  inherited;
end;

constructor TfrxIBXQuery.DesignCreate(AOwner: TComponent; Flags: Word);
var
  i: Integer;
  l: TList;
begin
  inherited;
  l := Report.AllObjects;
  for i := 0 to l.Count - 1 do
    if TObject(l[i]) is TfrxIBXDatabase then
    begin
      SetDatabase(TfrxIBXDatabase(l[i]));
      break;
    end;
end;

class function TfrxIBXQuery.GetDescription: String;
begin
  Result := 'IBX Query';
end;

procedure TfrxIBXQuery.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited;
```

```
  if (Operation = opRemove) and (AComponent = FDatabase) then
    SetDatabase(nil);
end;

procedure TfrxIBXQuery.SetDatabase(const Value: TfrxIBXDatabase);
begin
  FDatabase := Value;
  if Value <> nil then
    FQuery.Database := Value.Database
  else if IBXComponents <> nil then
    FQuery.Database := IBXComponents.DefaultDatabase
  else
    FQuery.Database := nil;
  DBConnected := FQuery.Database <> nil;
end;

procedure TfrxIBXQuery.SetMaster(const Value: TDataSource);
begin
  FQuery.DataSource := Value;
end;

function TfrxIBXQuery.GetSQL: TStrings;
begin
  Result := FQuery.SQL;
end;

procedure TfrxIBXQuery.SetSQL(Value: TStrings);
begin
  FQuery.SQL := Value;
end;

procedure TfrxIBXQuery.UpdateParams;
begin
  { in this method it is sufficient to assign values from Params into FQuery.Params }
  { this is performed via standard procedure }
  frxParamsToTParams(Self, FQuery.Params);
end;

procedure TfrxIBXQuery.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

Registering components

All components registration is performed in “Initialization” section.

```
initialization
  { use standard pictures indexes 37,38,39 instead of pictures}
  frxObjects.RegisterObject1(TfrxIBXDataBase, nil, '', '', 0, 37);
  frxObjects.RegisterObject1(TfrxIBXTable, nil, '', '', 0, 38);
  frxObjects.RegisterObject1(TfrxIBXQuery, nil, '', '', 0, 39);
finalization
  frxObjects.Unregister(TfrxIBXDataBase);
  frxObjects.Unregister(TfrxIBXTable);
  frxObjects.Unregister(TfrxIBXQuery);
end.
```

This is quite enough to use our DB components in reports. There are two more things left at this stage: register DB classes in the script system in order to make them available in the script, and to register several property editors (for example, TfrxIBXTable.TableName) to make working with the component easier.

It is better to store the script registration code in a separate file with RTTI suffix. See more about classes registration in script system in corresponding chapter. Here is example of such file:

```
unit frxIBXRTTI;
interface
{$I frx.inc}

implementation
uses
  Windows, Classes, fs_iinterpreter, frxIBXComponents
{$IFDEF Delphi6}
, Variants
{$ENDIF};

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;

{ TFunctions }
constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
  begin
    AddClass(TfrxIBXDatabase, 'TfrxComponent');
    AddClass(TfrxIBXTable, 'TfrxCustomDataset');
    AddClass(TfrxIBXQuery, 'TfrxCustomQuery');
  end;
end;

initialization
  fsRTTIModules.Add(TFunctions);
end.
```

Property editors

It is recommended to place property editors code in a separate file with Editor suffix. In our case, it was necessary to write editors for `TfrxIBXDatabase.DatabaseName`, `TfrxIBXTable.IndexName`, `TfrxIBXTable.TableName` properties. See more about writing properties editors in corresponding chapter. Below is example of such file:

```
unit frxIBXEditor;

interface
{$I frx.inc}

implementation

uses
  Windows, Classes, SysUtils, Forms, Dialogs, frxIBXComponents, frxCustomDB,
  frxDsgnIntf, frxRes, IBDatabase, IBTable
{$IFDEF Delphi6}
, Variants
{$ENDIF};

type
  TfrxDatabaseNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    function Edit: Boolean; override;
  end;

  TfrxTableNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

  TfrxIndexNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

{ TfrxDatabaseNameProperty }
function TfrxDatabaseNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { this property possesses  editor }
  Result := [paDialog];
end;

function TfrxDatabaseNameProperty.Edit: Boolean;
var
  SaveConnected: Bool;
  db: TIBDatabase;
begin
  { get link to TfrxIBXDatabase.Database }
  db := TfrxIBXDatabase(Component).Database;
  { create standard OpenDialog }
  with TOpenDialog.Create(nil) do
  begin
    InitialDir := GetCurrentDir;
    { we are interested in *.gdb files }
    Filter := frxResources.Get('ftDB') + ' (*.gdb)|*.gdb|' + frxResources.Get('ftAllFiles') + '
(*.*)|*.*';
    Result := Execute;
    if Result then
    begin
```

```
    begin
      SaveConnected := db.Connected;
      db.Connected := False;
      { if dialogue is completed successfully, assign new DB name }
      db.DatabaseName := FileName;
      db.Connected := SaveConnected;
    end;
    Free;
  end;
end;

{ TfrxTableNameProperty }
function TfrxTableNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  {  property represents list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxTableNameProperty.GetValues;
var
  t: TIBTable;
begin
  inherited;
  { get link to  TIBTable  component }
  t := TfrxIBXTable(Component).Table;
  { fill list of tables available }
  if t.Database <> nil then
    t.DataBase.GetTableNames(Values, False);
end;

{ TfrxIndexProperty }
function TfrxIndexNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  {  property represents list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxIndexNameProperty.GetValues;
var
  i: Integer;
begin
  inherited;
  try
    { get link to TIBTable component }
    with TfrxIBXTable(Component).Table do
      if (TableName <> '') and (IndexDefs <> nil) then
      begin
        { update indexes }
        IndexDefs.Update;
        { fill list of indexes available }
        for i := 0 to IndexDefs.Count - 1 do
          if IndexDefs[i].Name <> '' then
            Values.Add(IndexDefs[i].Name);
      end;
  except
  end;
end;

initialization
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXDataBase, 'DatabaseName',
TfrxDataBaseNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable, 'TableName', TfrxTableNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable, 'IndexName', TfrxIndexNameProperty);
end.
```

Custom Functions Connection to Report

FastReport has a large number of built in standard functions, for use in report designing. There is also the ability to add your own custom functions. Function connection is performed using the “FastScript” script library interface, which is included in FastReport (to learn more about FastScript, refer to it’s library manual).

An example of how procedures and/or functions can be connected. Function’s parameter quantity and type can vary. One cannot transfer parameters of “Set” and “Record" type, as they are not supported by FastScript. You must transfer such parameters as simpler types, for example, to transfer `TRect` as X0, Y0, X1, Y1: Integer. See more about process of adding functions with different parameters in FastScript documentation.

On the Delphi form declare the function or procedure and its code.

```
function TForm1.MyFunc(s: String; i: Integer): Boolean;
begin
// necessary logic
end;

procedure TForm1.MyProc(s: String);
begin
// necessary logic
end;
```

create the onuser function handler for the report component.

```
function TForm1.frxReport1UserFunction(const MethodName: String;
var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;
```

Use the report component’s add method to add to the function list (usually in on create or on show event of the Delphi form).

```
  frxReport1.AddFunction('function MyFunc(s: String; i: Integer): Boolean');
  frxReport1.AddFunction('procedure MyProc(s: String)');
```

Connected function can now be used in report script; furthermore, one can refer to it from objects of `TfrxMemoView` type. This function is also displayed in "Data tree" window’s function page tab. In this window functions are split into categories, and when you select any function, a hint about this function appears in the bottom pane of the window.

Modify code sample above to register functions in separate category, and display function description hint:

```
frxReport1.AddFunction('function MyFunc(s: String; i: Integer): Boolean', 'My functions', ' MyFunc
function always returns True');
frxReport1.AddFunction('procedure MyProc(s: String)', 'My functions', ' MyProc procedure does not do
anything');
```

Your added items will now appear under the category “My Functions”.

If you want to register functions in one of the existing categories, use one of the following category names:

'ctString' – string function;

'ctDate' - date/time functions;

'ctConv' - conversion functions;

'ctFormat' - formatting;

'ctMath' - mathematical functions;

'ctOther' - other functions.

If blank category name is specified, the function is placed in the functions tree root.

If you wish to add a large number of functions, it is recommended to placet all logic in a separate library unit. Here is an example:

```
unit myfunctions;

interface

implementation

uses SysUtils, Classes, fs_iinterpreter;
// you can also add a reference to any other external library here

type
  TFunctions = class(TfsRTTIModule)
  private
    function CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String; var Params:
Variant): Variant;
  public
    constructor Create(AScript: TfsScript); override;
  end;

function MyFunc(s: String; i: Integer): Boolean;
begin
// necessary logic
end;

procedure MyProc(s: String);
begin
// necessary logic
end;

{ TFunctions }
constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
    AddMethod('function MyFunc(s: String; i: Integer): Boolean', CallMethod, 'My functions', ' MyFunc
function always returns True');
    AddMethod('procedure MyProc(s: String)', CallMethod, 'My functions', ' MyProc procedure does not do
anything'');
  end;
end;

function TFunctions.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String; var
Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;

initialization
  fsRTTIModules.Add(TFunctions);
end.
```

Save the file with a .pas extension then add a reference to it in the uses clause of your Delphi project's form and all your custom functions will be available for use in any report component. No need to write code to add these functions to each `TfrxReport`, and no need to write additional code for each report component's "onuserfunction " handler.

Custom Wizards Writing

You can extend FastReport functionality with help of custom wizards. FastReport, for example, contains standard “Report Wizard,” which is called from “File|New…” menu.

There are two types of wizards supported in FastReport. The first type includes wizards already mentioned, called from “File|New…” menu. The second one includes wizards, which can be called from “Wizards” toolbar.

Basic class for any wizard is `TfrxCustomWizard`, defined in “frxClass” file.

```
TfrxCustomWizard = class(TComponent)
Public
  Constructor Create(AOwner: TComponent); override;
  class function GetDescription: String; virtual; abstract;
  function Execute: Boolean; virtual; abstract;
  property Designer: TfrxCustomDesigner read FDesigner;
  property Report: TfrxReport read FReport;
end;
```

To write your own wizard, it is necessary to inherit from this class and override at least `GetDescription` and `Execute` methods. The first one returns wizard name; the second one is called when running the wizard; it must return “True,” if wizard finished working successfully and made any changes to report. During wizard work, you can call designer and report methods and properties properly via `Designer` and `Report` properties.

Wizard registration and deleting is performed via procedures described in “frxDsgnIntf” file:

```
frxWizards.Register(ClassRef: TfrxWizardClass; ButtonBmp: TBitmap; IsToolbarWizard: Boolean = False);
frxWizards.Unregister(ClassRef: TfrxWizardClass);
```

At registration, one enters wizard class name, its picture, and specifies if wizard is placed in “Wizards” toolbar. If wizard should be placed in toolbar, ButtonBmp size must be either 16x16 pixels, or otherwise 32x32 pixels.

Let us examine primitive wizard, which is being registered in "File|New..." menu, and then adds new page to report.

```
uses frxClass, frxDsgnIntf;

type
  TfrxMyWizard = class(TfrxCustomWizard)
  public
    class function GetDescription: String; override;
    function Execute: Boolean; override;
  end;

class function TfrxMyWizard.GetDescription: String;
begin
  Result := 'My Wizard';
end;

function TfrxMyWizard.Execute: Boolean;
var
  Page: TfrxReportPage;
begin
  { lock any drawings in designer }
  Designer.Lock;
  { create new page in report }
  Page := TfrxReportPage.Create(Report);
  { create unique name for page }
  Page.CreateUniqueName;
  { set paper sizes and orientation by default }
  Page.SetDefaults;
  { update report pages and switch focus to last added page }
  Designer.ReloadPages(Report.PagesCount - 1);
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { load picture from resource; of course, you should place it there first }
  Bmp.LoadFromResourceName(hInstance, 'frxMyWizard');
  frxWizards.Register(TfrxMyWizard, Bmp);

finalization
  frxWizards.Unregister(TfrxMyWizard);
  Bmp.Free;
end.
```