FastReport VCL Programmer Manual

Version 2024.1

Working with TfrxReport component

Loading and saving a report

By default, a report form is stored together with the project form, i.e. in a DFM file. In most cases, no more operations required, and you thus would not need to take special measures to load a report. If you decided to store a report form in a file or in the DB BLOb-field (this provides great flexibility, i.e. you can modify a report without recompiling the program), you would have to use the `TfrxReport` methods for report loading and saving:

```
function LoadFromFile(const FileName: String; ExceptionIfNotFound: Boolean = False): Boolean;
```

Loads a report from the file with the given name. If the second parameter is equal to "True" and the file is not found, then it generates an exception. If the file is loaded successfully, it returns "True."

```
procedure LoadFromStream(Stream: TStream);
```

Loads a report from the stream.

```
procedure SaveToFile(const FileName: String);
```

Saves a report to a file with the specified name.

```
procedure SaveToStream(Stream: TStream);
```

Saves a report to a stream.

> File with a report form has the "FR3" extension by default.

Examples:

Pascal:

```
frxReport1.LoadFromFile('c:\1.fr3');
frxReport1.SaveToFile('c:\2.fr3');
```

C++:

```
frxReport1->LoadFromFile("c:\\1.fr3");
frxReport1->SaveToFile("c:\\2.fr3");
```

Designing a report

Calling the report designer is performed via the TfrxReport.DesignReport method. A designer must be included in your project (it is enough to either use the `TfrxDesigner` component, or add the "frxDesgn" unit into the uses list).

The `DesignReport` method takes two default parameters:

```
procedure DesignReport(Modal: Boolean = True; MDIChild: Boolean = False);
```

The Modal parameter determines whether the designer should be modal. The MDIChild parameter allows to make a designer window a MDI child window.

Example:

```
frxReport1.DesignReport;
```

Running a report

Applying one of the following two `TfrxReport` methods starts a report:

```
procedure ShowReport(ClearLastReport: Boolean = True);
```

Starts a report and displays the result in the preview window. If the “ClearLastReport” parameter is equal to “False,” then the report will be added to the previously constructed one, otherwise the previously constructed report will be cleared (by default).

```
function PrepareReport(ClearLastReport: Boolean = True): Boolean;
```

Starts a report, without opening the preview window. The parameter assignment is the same as in the “ShowReport” method. If a report was constructed successfully, it returns “True.”

In most cases, it is more convenient to use the first method. It displays the preview window right away, while a report continues to be constructed.

The “ClearLastReport” parameter is convenient to use in case when it is necessary to add another report to the previously constructed one (such technique is used for batch report printing).

Example:

```
frxReport1.ShowReport;
```

Previewing a report

It is possible to display a report in the preview window in two ways: either by calling the `TfrxReport.ShowReport` method (described above), or with the help of the `TfrxReport.ShowPreparedReport` method. In the second case, the report construction is not performed, but a finished report is displayed. That means, that you should either construct it beforehand with the help of the `PrepareReport` method, or load a previously constructed report from the file (see Loading and saving a finished report).

Example:

Pascal:

```
if frxReport1.PrepareReport then
  frxReport1.ShowPreparedReport;
```

C++:

```
if(frxReport1->PrepareReport(true))
  frxReport1->ShowPreparedReport();
```

In this case, report construction is finished first, and after that it is displayed in the preview window. Construction of a large report can take a lot of time, and that is why it is better to use the `ShowReport` anisochronous method, than the `PrepareReport` / `ShowPreparedReport` one. One can assign preview settings by default via the TfrxReport.PreviewOptions property.

Printing a report

In most cases, you will print a report from the preview window. To print a report manually, you should use the `TfrxReport.Print` method, for example:

```
frxReport1.LoadFromFile(...);
frxReport1.PrepareReport;
frxReport1.Print;
```

At the same time, the dialogue, in which printing parameters can be set, will appear. You can assign settings by default, and disable a printing dialogue with the help of the `TfrxReport.PrintOptions` property.

Loading and saving a finished report

It can be executed from the preview window. This also can be performed manually with the help of the `TfrxReport.PreviewPages` methods:

```
function LoadFromFile(const FileName: String; ExceptionIfNotFound: Boolean = False): Boolean;
procedure SaveToFile(const FileName: String);
procedure LoadFromStream(Stream: TStream);
procedure SaveToStream(Stream: TStream);
```

Assignment and the parameters are similar to the corresponding `TfrxReport` methods. A file, which contains the finished report, has “FP3” extension by default.

Example:

Pascal:

```
frxReport1.PreviewPages.LoadFromFile('c:\1.fp3');
frxReport1.ShowPreparedReport;
```

C++:

```
frxReport1->PreviewPages->LoadFromFile("c:\\1.fp3");
frxReport1->ShowPreparedReport();
```

> Note: after finished report loading is completed, its previewing is executed via the ShowPreparedReport method!

Exporting a report

It can be performed from a preview window. The operation can also be executed manually, via the `TfrxReport.Export` method. In the parameter of this method you should specify the export filter you want to use:

```
frxReport1.Export(frxHTMLExport1);
```

The export filter component must be available (you must put it on the form of your project) and be adjusted correctly.

Creating a custom preview window

FastReport displays reports in the standard preview window. If it does not suit you for some reason, a custom preview form may be created. For this purpose, the `TfrxPreview` component from the FastReport component palette was designed. To display a report, the link to this component should be assigned to the `TfrxReport.Preview` property.

There is two typical problems when using `TfrxPreview` component. It does not handle keys (arrows, PgUp, PgDown etc) and mouse wheel (if any). To make `TfrxPreview` working with keys, pass the focus to it (it can be done, for example, in the `OnShow` event handler of a form):

```
frxPreview.SetFocus;
```

To make `TfrxPreview` working with mouse scroll, you have to create `OnMouseWheel` event handler of a form and call `TfrxPreview.MouseWheelScroll` method in this handler:

```
procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
  WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);
begin
  frxPreview1.MouseWheelScroll(WheelDelta, Shift, MousePos);
end;
```

Building a composite report (batch printing)

In some cases it is required to organize printing of several reports at once, or capsulate and present several reports in one preview window. To perform this, there are tools in FastReport, which allow building a new report in addition to an already existing one. The `TfrxReport.PrepareReport` method has the optional «ClearLastReport» Boolean parameter, which is equal to «True» by default. This parameter defines whether it is necessary to clear pages of the previously built report. The following code shows how to build a batch from two reports:

Pascal:

```
frxReport1.LoadFromFile('1.fr3');
frxReport1.PrepareReport;
frxReport1.LoadFromFile('2.fr3');
frxReport1.PrepareReport(False);
frxReport1.ShowPreparedReport;
```

C++:

```
frxReport1->LoadFromFile("1.fr3");
frxReport1->PrepareReport(true);
frxReport1->LoadFromFile("2.fr3");
frxReport1->PrepareReport(false);
frxReport1->ShowPreparedReport();
```

We load the first report and build it without displaying. Then we load the second one into the same `TfrxReport` object and build it with the «ClearLastReport» parameter, equal to «False». This allows the second report to be added to the one previously built. After that, we display a finished report in the preview window.

Numbering of pages in a composite report

You can use the «Page», «Page#», «TotalPages» and «TotalPages#» system variables for displaying a page number or a total number of pages. In composite reports, these variables work in the following way:

`Page` – page number in the current report

`Page#` - page number in the batch

`TotalPages` – total number of pages in the current report (a report must be a two-pass one)

`TotalPages#` - total number of pages in a batch.

Combination of pages in a composite report

As it was said above, the `PrintOnPreviousPage` property of the report design page lets you splice pages when printing, i.e. using free space of the previous page. In composite reports, it allows to start creation of a new report on free space of the previous report's last page. To perform this, one should enable the `PrintOnPreviousPage` property of the first design page of each successive report.

Interactive reports

In interactive reports, one can define a reaction for mouse-click on any of the report objects in a preview window. For example, a user can click on the data line, and thus run a new report with detailed data of the selected line.

Any report can become interactive. To perform this, you only need to create a `TfrxReport.OnClickObject` event handler. Here is a code example of this handler below:

Pascal:

```
procedure TForm1.frxReport1ClickObject(Page: TfrxPage; View: TfrxView;
  Button: TMouseButton; Shift: TShiftState; var Modified: Boolean);
begin
  if View.Name = 'Memo1' then
    ShowMessage('Memo1 contents:' + #13#10 + TfrxMemoView(View).Text);
  if View.Name = 'Memo2' then
  begin
    TfrxMemoView(View).Text := InputBox('Edit', 'Edit Memo2 text:', TfrxMemoView(View).Text);
    Modified := True;
  end;
end;
```

C++:

```
void __fastcall TForm1::frxReport1ClickObject(TfrxView *Sender,
      TMouseButton Button, TShiftState Shift, bool &Modified)
{
  TfrxMemoView * Memo;
  if(Memo =  dynamic_cast <TfrxMemoView *> (Sender))
  {
    if(Memo->Name == "Memo1")
      ShowMessage("Memo1 contents:\n\r" + Memo->Text);
    if(Memo->Name == "Memo2")
    {
      Memo->Text = InputBox("Edit", "Edit Memo2 text:", Memo->Text);
       Modified = true;
    }
  }
}
```

In the OnClickObject handler, you can do the following:

- modify contents of an object or a page, passed to the handler (thus, the «Modified» flag should be specified, so that the modifications would be taken into consideration);
- call the `TfrxReport.PrepareReport` method for reconstructing/rebuilding a report.

In this example, clicking on the object with the «Memo1» name results in displaying a message with the contents of this object. When clicking on the «Memo2,» a dialogue is displayed, where the contents of this object can be modified. Setting of the «Modified» flag to «True» allows holding and displaying alterations.

In the same way, a different reaction for a click can be defined; it may, for example, run a new report. It is necessary to NOTE the following. One `TfrxReport` component can display only one report in the preview window. That is why one should run a report either in a separate `TfrxReport` object, or in the same one, but the current report

must be erased.

To give a prompting indication about clickable objects to the end user, we can modify the mouse cursor when it passes over a clickable object in the preview window. To do this, select the desired object in the report designer and set it's cursor property to something other than crDefault.

One more detail concerns the defining clickable objects. In simple reports, this can be defined either in the object's name, or in its contents. However, this cannot always be performed in more complicated cases. For example, a detailed report should be created in a selected data line. A user clicked on the «Memo1» object with the '12' contents. What data line does this object refer to? That is why you should know the primary key, which identifies this line unambiguously. FastReport enables to assign a string, containing any data (in our case the data of the primary key), to every report's object. This string is stored in the `TagStr` property.

Let us illustrate this process by an example of a report, which is included in the FastReportDemo.exe - 'Simple list' demo. This is the list of clients of a company, containing such data as «client's name,» «address,» «contact person,» etc. The data source is the «Customer.db» table from the DBDEMOS demo database. This table has a primary key, i.e. the «CustNo» field, which is not presented in the report. Our task is to determine what record it refers to by clicking on any object from the finished report, which means to get the value of the primary key. To perform this, it is sufficient to enter the following value into the `TagStr` property of all the objects, lying on the Master Data band:

```
[Customers."CustNo"]
```

During a report's building, the `TagStr` property's contents are calculated in the same way, as contents of text objects are calculated; this means that the variables' values are substituted in place of all variables. A variable in this particular case is what is enclosed into the square brackets. That is why the lines of the '1005', '2112', etc. types will be contained in the `TagStr` property of the objects lying on the Master Data after report building. A simple conversion from a string into an integer will give us a value of the primary key, with which a required record can be found.

If the primary key is composite (i.e. it contains several fields) the `TagStr` property's contents can be the following:

```
[Table1."Field1"];[Table1."Field2"]
```

After constructing a report, the `TagStr` property contains values of the '1000;1' type, from which it is rather not difficult to get values of a key as well.

Access report objects from a code

FastReport' s objects (such as report page, band, memo-object) are not directly accessible from your code. This means that you cannot address the object by its name, as, for example, when you addressing to a button on your form. To address an object, it should be found with the help of the `TfrxReport.FindObject` method:

Pascal:

```
var
  Memo1: TfrxMemoView;
  Memo1 := frxReport1.FindObject('Memo1') as TfrxMemoView;
```

C++:

```
TfrxMemoView * Memo = dynamic_cast <TfrxMemoView *> (frxReport1->FindObject("Memo1"));
```

after that, one can address the object's properties and methods. You can address the report's pages using the `TfrxReport.Pages` property:

Pascal:

```
var
  Page1: TfrxReportPage;
  Page1 := frxReport1.Pages[1] as TfrxReportPage;
```

C++:

```
TfrxReportPage * Page1 = dynamic_cast <TfrxReportPage *> (frxReport1->Pages[1]);
```

Creating a report form from code

As a rule, you will create most reports using the designer. Nevertheless, in some cases (for example, when the report's form is unknown) it is necessary to create a report manually, from code.

To create a report manually, one should perform the following steps in order:

- clear the report component
- add data sources
- add the "Data" page
- add report's page
- add bands on a page
- set bands' properties, and then connect them to the data
- add objects on each band
- set objects' properties, and then connect them to the data

Let us examine creation of a simple report of the «list» type. Assume that we have the following components: frxReport1: TfrxReport and frxDBDataSet1: TfrxDBDataSet (the last one is connected to data from the DBDEMOS, the «Customer.db» table). Our report will contain one page with the «Report Title» and «Master Data» bands. On the «Report Title» band there will be an object with the "Hello FastReport!" text, and the «Master Data» one will contain an object with a link to the "CustNo" field.

Pascal:

```
var
  DataPage: TfrxDataPage;
  Page: TfrxReportPage;
  Band: TfrxBand;
  DataBand: TfrxMasterData;
  Memo: TfrxMemoView;

{ clear a report }
frxReport1.Clear;

{ add a dataset to the list of ones accessible for a report }
frxReport1.DataSets.Add(frxDBDataSet1);

{ add the "Data" page }
DataPage := TfrxDataPage.Create(frxReport1);

{ add a page }
Page := TfrxReportPage.Create(frxReport1);

{ create a unique name }
Page.CreateUniqueName;

{ set sizes of fields, paper and orientation by default }
Page.SetDefaults;

{ modify paper's orientation }
Page.Orientation := poLandscape;

{ add a report title band}
```

```
Band := TfrxReportTitle.Create(Page);
Band.CreateUniqueName;

{ it is sufficient to set the «Top» coordinate and height for a band }
{ both coordinates are in pixels }
Band.Top := 0;
Band.Height := 20;

{ add an object to the report title band }
Memo := TfrxMemoView.Create(Band);
Memo.CreateUniqueName;
Memo.Text := 'Hello FastReport!';
Memo.Height := 20;

{ this object will be stretched according to band’s width }
Memo.Align := baWidth;

{ add the masterdata band }
DataBand := TfrxMasterData.Create(Page);
DataBand.CreateUniqueName;
DataBand.DataSet := frxDBDataSet1;

{ the Top coordinate should be greater than the previously added band’s top + height}
DataBand.Top := 100;
DataBand.Height := 20;

{ add an object on master data }
Memo := TfrxMemoView.Create(DataBand);
Memo.CreateUniqueName;

{ connect to data }
Memo.DataSet := frxDBDataSet1;
Memo.DataField := 'CustNo';
Memo.SetBounds(0, 0, 100, 20);

{ adjust the text to the right object’s margin }
Memo.HAlign := haRight;

{ show the report }
frxReport1.ShowReport;
```

C++:

```
TfrxDataPage * DataPage;
TfrxReportPage * Page;
TfrxBand * Band;
TfrxMasterData * DataBand;
TfrxMemoView * Memo;

// clear a report
frxReport1->Clear();

// add a dataset to the list of ones accessible for a report
frxReport1->DataSets->Add(frxDBDataset1);

// add the "Data" page
DataPage = new TfrxDataPage(frxReport1);

// add a page
Page = new TfrxReportPage(frxReport1);

// create a unique name
Page->CreateUniqueName();

// set sizes of fields, paper and orientation by default
```

```
Page->SetDefaults();

// modify paper's orientation
Page->Orientation = poLandscape;

// add a report title band
Band = new TfrxReportTitle(Page);
Band->CreateUniqueName();

// it is sufficient to set the «Top» coordinate and height for a band
// both coordinates are in pixels
Band->Top = 0;
Band->Height = 20;

// add an object to the report title band
Memo = new TfrxMemoView(Band);
Memo->CreateUniqueName();
Memo->Text = "Hello FastReport!";
Memo->Height = 20;

// this object will be stretched according to band's width
Memo->Align = baWidth;

// add the masterdata band
DataBand = new TfrxMasterData(Page);
DataBand->CreateUniqueName();
DataBand->DataSet = frxDBDataset1;

// the Top coordinate should be greater than the previously added band's top + height
DataBand->Top = 100;
DataBand->Height = 20;

// add an object on master data
Memo = new TfrxMemoView(DataBand);
Memo->CreateUniqueName();

// connect to data
Memo->DataSet = frxDBDataset1;
Memo->DataField = "CustNo";
Memo->SetBounds(0, 0, 100, 20);

// adjust the text to the right object's margin
Memo->HAlign = haRight;

// show the report
frxReport1->ShowReport(true);
```

Let us explain some details.

All the data sources, which are to be used in the report, must be added to the list of data sources. In our case, this is performed using the

```
frxReport1.DataSets.Add(frxDBDataSet1)
```

line. Otherwise, a report will not work.

The "Data" page is necessary for inserting internal datasets such as `TfrxADOTable` into the report. Such datasets can be placed only to the "Data" page.

The call for `Page.SetDefaults` is not necessary, since in this case a page will have the A4 format and margins of 0 mm. SetDefaults sets 10mm margins and takes page size and alignment, which a printers have by default.

While adding bands to a page, you should make sure they do not overlap each other. To perform this, it is sufficient to set the «Top» and «Height» coordinates. There is no point in modifying the «Left» and «Width» coordinates, since a band always has the width of the page, on which it is located (in case of vertical bands it's not true – you should set Left and Width properties and don't care about Top and Height). One should note, that the order of bands' location on a page is of great importance. Always locate bands in the same way you would do it in the designer.

Objects' coordinates and sizes are set in pixels. Since the `Left`, `Top`, `Width`, and `Height` properties of all objects have the «Extended» type, you can point out non-integer values. The following constants are defined for converting pixels into centimeters and inches:

```
fr01cm = 3.77953;
fr1cm  = 37.7953;
fr01in = 9.6;
fr1in  = 96;
```

For example, a band's height equal to 5 mm can be set as follows:

```
Band.Height := fr01cm * 5;
Band.Height := fr1cm * 0.5;
```

Creating a dialogue form from a code

As we know, a report can contain dialogue forms. The following example shows how to create a dialogue form, with an «OK» button:

Pascal:

```
{ for working with dialogue objects the following unit should be used }
uses frxDCtrl;

var
  Page: TfrxDialogPage;
  Button: TfrxButtonControl;

{ add a page }
Page := TfrxDialogPage.Create(frxReport1);

{ create a unique name }
Page.CreateUniqueName;

{ set sizes }
Page.Width := 200;
Page.Height := 200;

{ set a position }
Page.Position := poScreenCenter;

{ add a button }
Button := TfrxButtonControl.Create(Page);
Button.CreateUniqueName;
Button.Caption := 'OK';
Button.ModalResult := mrOk;
Button.SetBounds(60, 140, 75, 25);

{ show a report }
frxReport1.ShowReport;
```

C++:

```
// for working with dialogue objects the following unit should be used
#include "frxDCtrl.hpp"

TfrxDialogPage * Page;
TfrxButtonControl * Button;

// add a page
Page = new TfrxDialogPage(frxReport1);

// create a unique name
Page->CreateUniqueName();

// set sizes
Page->Width = 200;
Page->Height = 200;

// set a position
Page->Position = poScreenCenter;

// add a button
Button = new TfrxButtonControl(Page);
Button->CreateUniqueName();
Button->Caption = "OK";
Button->ModalResult = mrOk;
Button->SetBounds(60, 140, 75, 25);

// show a report
frxReport1->ShowReport(true);
```

Modifying report page’s properties

Sometimes it is necessary to modify report page settings (for example, to modify paper alignment or size) from a code. The `TfrxReportPage` class contains the following properties, defining the size of the page:

```
    property Orientation: TPrinterOrientation default poPortrait;
    property PaperWidth: Extended;
    property PaperHeight: Extended;
    property PaperSize: Integer;
```

The `PaperSize` property sets paper format. This is one of the standard values, defined in the Windows.pas (for example, DMPAPER_A4). If a value to this property is assigned, FastReport fills the `PaperWidth` and `PaperHeight` properties automatically (paper size in millimeters). Setting the DMPAPER_USER (or 256) value as a format, would mean that custom paper size is set. In this case, the `PaperWidth` and `PaperHeight` properties should be filled manually.

The following example shows, how to modify parameters of the first page (it is assumed that we already have a report):

Pascal:

```
var
  Page: TfrxReportPage;

{ the first report’s page has [1] index. [0] is the Data page. }
Page := TfrxReportPage(frxReport1.Pages[1]);

{ modify the size }
Page.PaperSize := DMPAPER_A2;

{ modify the paper orientation }
Page.Orientation := poLandscape;
```

C++:

```
TfrxReportPage * Page;

// the first report’s page has [1] index. [0] is the Data page.
Page = (TfrxReportPage *)frxReport1.Pages[1];

// modify the size
Page->PaperSize = DMPAPER_A2;

// modify the paper orientation
Page->Orientation = poLandscape;
```

Report construction with the help of a code

The FastReport engine usually is responsible for report's constructing. It shows report's bands in a particular order as many times, as the datasource to which it is connected requires, thus forming a finished report. Sometimes it is necessary to create a report of a non-standard form, which FastReport's engine is unable to produce. In this case, one can use the ability of constructing a report manually, with the help of the `TfrxReport.OnManualBuild` event. If to define a handler of this event, the FastReport engine sends management to it. At the same time, allocation of responsibilities for forming a report is changed in the following way:

FastReport engine:

- report's preparation (script, data sources initialization, bands' tree forming)
- all calculations (aggregate functions, event handlers)
- new pages/columns' forming (automatic showing a page/column header/footer, report title/summary)
- other routine work

Handler:

- bands' presentation in a certain order

The OnManualBuild handler's essence is to issue commands concerning presenting certain bands to the FastReport's engine. The engine itself will do the rest: a new page will be created, as soon as there is no place in the current one; execution of scripts will be performed, etc.

The engine is represented with the `TfrxCustomEngine` class. A link to the instance of this class is located in the `TfrxReport.Engine` property.

Property or method	Description
procedure NewColumn	Creates a new column. If a column is the last one, it creates a new page.
procedure NewPage	Creates a new page.
procedure ShowBand(Band: TfrxBand)	Shows a band.
procedure ShowBand(Band: TfrxBandClass)	Shows a band of the given type.
function FreeSpace: Extended	Returns the amount of free space on the page (in pixels). After the next band is presented, this value descends.
property CurColumn: Integer	Returns/sets the current column's number
property CurX: Extended	Returns/sets the current X position.
property CurY: Extended	Returns/sets the current Y position. After the next band is presented, this value ascends.
property DoublePass: Boolean	Defines whether a report is a two-pass one.

Property or method	Description
property FinalPass: Boolean	Defines whether the current pass is the last one.
property FooterHeight: Extended	Returns the page footer height.
property HeaderHeight: Extended	Returns the page header height.
property PageHeight: Extended	Returns the height of the page's printable region.
property PageWidth: Extended	Returns the width of the page's printable region.
property TotalPages: Integer	Returns the number of pages in a finished report (only on the second pass of the two-pass report).

Let us give an example of a simple handler. There is two «Master Data» bands in a report, which are not connected to data. The handler presents these bands in an interlaced order, six times each one. After six bands, a small gap is made.

Pascal:

```
var
  i: Integer;
  Band1, Band2: TfrxMasterData;

{ find required bands }
Band1 := frxReport1.FindObject('MasterData1') as TfrxMasterData;
Band2 := frxReport1.FindObject('MasterData2') as TfrxMasterData;
for i := 1 to 6 do
begin
  { lead/deduce bands one after another }
  frxReport1.Engine.ShowBand(Band1);
  frxReport1.Engine.ShowBand(Band2);
  { make a small gap }
  if i = 3 then
    frxReport1.Engine.CurY := frxReport1.Engine.CurY + 10;
end;
```

C++:

```
int i;
TfrxMasterData * Band1;
TfrxMasterData * Band2;

// find required bands
Band1 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData1"));
Band2 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData2"));
for(i = 1; i <= 6; i++)
{
  // lead/deduce bands one after another
  frxReport1->Engine->ShowBand(Band1);
  frxReport1->Engine->ShowBand(Band2);
  // make a small gap
  if(i == 3)
    frxReport1->Engine->CurY += 10;
}
```

The next example shows two groups of bands alongside each other.

Pascal:

```
var
  i, j: Integer;
  Band1, Band2: TfrxMasterData;
  SaveY: Extended;

Band1 := frxReport1.FindObject('MasterData1') as TfrxMasterData;
Band2 := frxReport1.FindObject('MasterData2') as TfrxMasterData;
SaveY := frxReport1.Engine.CurY;

for j := 1 to 2 do
begin
  for i := 1 to 6 do
  begin
    frxReport1.Engine.ShowBand(Band1);
    frxReport1.Engine.ShowBand(Band2);
    if i = 3 then
      frxReport1.Engine.CurY := frxReport1.Engine.CurY + 10;
  end;
  frxReport1.Engine.CurY := SaveY;
  frxReport1.Engine.CurX := frxReport1.Engine.CurX + 200;
end;
```

C++:

```
int i, j;
TfrxMasterData * Band1;
TfrxMasterData * Band2;
Extended SaveY;

Band1 = dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData1"));
Band2 = dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData2"));
SaveY = frxReport1->Engine->CurY;

for(j = 1; j <= 2; j++)
{
  for(i = 1; i <= 6; i++)
  {
    frxReport1->Engine->ShowBand(Band1);
    frxReport1->Engine->ShowBand(Band2);
    if(i == 3)
      frxReport1->Engine->CurY += 10;
  }
  frxReport1->Engine->CurY = SaveY;
  frxReport1->Engine->CurX += 200;
}
```

Printing an array

The primary example's code is located in the «FastReport Demos\PrintArray» ("FastReport Demos\BCB Demos\PrintArray") directory. Let us explain several details.

To print an array, we use a report with one «Master Data» band, which will be presented as many times, as there are elements in the array. To do this, place a `TfrxUserDataSet` component on the form, and then set it's properties (it is possible to do it in a code, as shown in our example):

```
RangeEnd := reCount
RangeEndCount := a number of elements in an array
```

After that, we connect the data-band to the `TfrxUserDataSet` component. To represent the array element, place a text object with the [element] line inside the «Master Data» band. The «element» variable is filled using a `TfrxReport.OnGetValue` event.

Printing a TStringList

The primary example's code is located in the «FastReport Demos\PrintStringList» («FastReport Demos\BCB Demos\PrintStringList») directory. The method is the same, as in the example with an array.

Printing a file

The primary example's code is located in the «FastReport Demos\PrintFile» («FastReport Demos\BCB Demos\PrintFile») directory. Let us explain several details.

For printing, you should use a report with a «Master Data» band, which will be printed once (to perform this, it should be connected to a data source, which contains one record; select a source named "Single row" from the list). Stretching («Stretch») and splitting («Allow Split») are enabled in the band. This means, that the way the band is stretched allows finding room for all objects located in it. However, if a band does not find room in a page, it will be presented partially in separate pages.

File contents are presented via the «Text» object, which contains the [file] variable. This variable, as in the previous examples, is filled in the `TfrxReport.OnGetValue` event. Stretching is also enabled in the object (the «Stretch» item from the contextual menu or the `StretchMode` property = smActualHeight).

Printing a TStringGrid

The initial example's code is located in the «FastReport Demos\PrintStringGrid» («FastReport Demos\BCB Demos\PrintStringGrid»") directory. Let us explain some details.

The `TStringGrid` component represents a table with several rows and columns. That means that a report stretches not only by height, but by width as well. To print such component, let us use the «Cross-tab» object (it becomes available when the `TfrxCrossObject` component is added to the project). This object is responsible only for printing table data with a number of rows and columns unknown beforehand. The object has two versions: `TfrxCrossView` for user's data printing, and `TfrxDBCrossView` for printing the specially prepared data from the DB table.

Let us use the `TfrxCrossView`. The object should be preliminarily set. To perform this, let us enter report's designer and call the object editor by double-clicking on it. We must set the number of the rows and columns' titles nesting, and the number of values in the table cells. In our case, all these values must be equal to «1». In our example, the rows and columns' titles and the total values of lines and columns are disabled as well.

It is necessary to fill the object with values from the StringGrid in the `TfrxReport.OnBeforePrint` event. A value is added via the `TfrxCrossView.AddValue` method. Its parameters are the following: composite index of a line, a column and the cell's value (which is composite as well, since an object can contain several values in a cell).

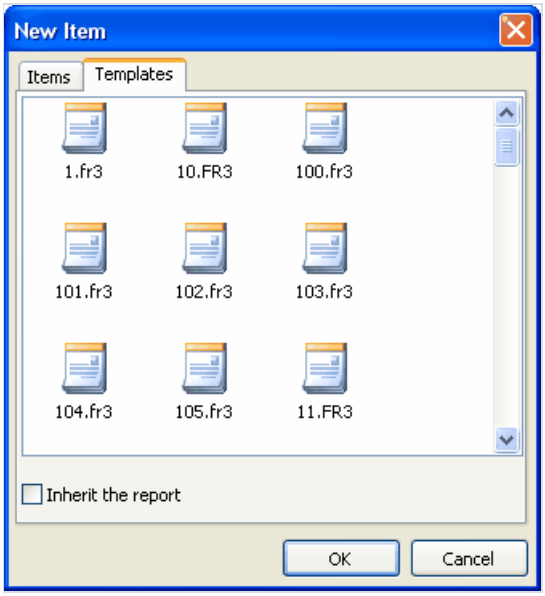
Printing TTable and TQuery

The initial example's code is located in the «FastReport's Demos\PrintTable» («FastReport Demos\BCB Demos\PrintTable»") directory. The principle of work is the same, as in the example with the `TStringGrid`. In this case, the row's index is its sequence number, the column's index is the name of a table field, and the cell's value is the table field's value. It is important to notice that the functions for cell's elements must be disabled in the «Cross-tab» object editor (since in a cell there are data of various kinds, this leads to the error in table creation) and the table title's sorting must be disabled too (otherwise columns will be sorted alphabetically).

Report inheritance

The report inheritance was described in the User's manual. We will describe some key moments here.

If you store your reports in files, you need to set up the folder name which FastReport will use to search the base report. This folder's content will be displayed in the "File|New..." and "Report|Options..." dialogs:



The `TfrxDesigner.TemplateDir` property is used for this purpose. By defalt it is empty, FastReport will search for base reports in the folder with your project's executable file (.exe). You can place the absolute or relative path into this property.

If you store your reports in the database, you have to write a code to load the base report from a DB and to get a list of available base reports. Use `TfrxReport.OnLoadTemplate` event to load a base report:

```
property OnLoadTemplate: TfrxLoadTemplateEvent read FOnLoadTemplate write FOnLoadTemplate;

TfrxLoadTemplateEvent = procedure(Report: TfrxReport; const TemplateName: String) of object;
```

This event's handler must load a base report with given TemplateName into Report object. Here is an example of such handler:

```
procedure TForm1.LoadTemplate(Report: TfrxReport; const TemplateName: String);
var
  BlobStream: TStream;
begin
  ADOTable1.First;
  while not ADOTable1.Eof do
  begin
    if AnsiCompareText(ADOTable1.FieldByName('ReportName').AsString, TemplateName) = 0 then
    begin
      BlobStream := TMemoryStream.Create;
      TBlobField(ADOTable1.FieldByName('ReportBlob')).SaveToStream(BlobStream);
      BlobStream.Position := 0;
      Report.LoadFromStream(BlobStream);
      BlobStream.Free;
      break;
    end;
    ADOTable1.Next;
  end;
end;
```

To get a list of available templates, you should use the `TfrxDesigner.OnGetTemplateList` event:

```
property OnGetTemplateList: TfrxGetTemplateListEvent read FOnGetTemplateList write FOnGetTemplateList;

TfrxGetTemplateListEvent = procedure(List: TStrings) of object;
```

This event's handler must return a list of available templates into List parameter. Here is an example of such handler:

```
procedure TForm1.GetTemplates(List: TList);
begin
  List.Clear;
  ADOTable1.First;
  while not ADOTable1.Eof do
  begin
    List.Add(ADOTable1.FieldByName('ReportName').AsString);
    ADOTable1.Next;
  end;
end;
```

FastReport can inherit already created reports. For that you should use the following function:

```
TfrxReport.InheritFromTemplate(const templName: String; InheritMode: TfrxInheritMode = imDefault):
Boolean
```

This function allows to inherit the current loaded report from the indicated report. The first parameter of the function is a file name of parent template, the second one allows to choose inherit mode:

- imDefault - derive the dialogue with offer to rename/delete the duplicates
- imDelete - delete all backup objects
- imRename - rename all backup objects

> Attention! The search of parent template is done referring the current template, that is necessary to keep catalogues structure at report storage place. Fast Report uses relative paths that's why there is no need to

> worry about application transfer (the only exception is when the current pattern and parent template are placed on different carriers or net path is used).

Multithreading

FastReport can operate independently in different threads, but there are some features:

- You can not create `TfrxDBDataSet` even in different threads, because "global list" is used for search and access will always take place to the first created `TfrxDBDataSet` (you can switch off the use of global list, it is active by default);
- If during report execution there are some changes in object characteristics (for example,Memo1.Left := Memo1.Left + 10 in script), than you need to remember that during the next operation if the property `TfrxReport.EngineOptions.DestroyForms` := False report template will already be modified and it will need to be reloaded or to use `TfrxReport.EngineOptions.DestroyForms` := True. During renewal you can't use interactive reports from the thread, because script's objects are deleted after renewal, that's why in some cases it is efficiently to use `TfrxReport.EngineOptions.DestroyForms` := False and to renew the template on your own during the next building cycle.

If necessary the global list due to which you can search the needed copies of `TfrxDBDataSet` can be switched off.

```
{ DestroyForms can be switched off, if every time you renew a report from a file or from a current }
FReport.EngineOptions.DestroyForms := False;
FReport.EngineOptions.SilentMode := True;

{ This property switches off the search through global list }
FReport.EngineOptions.UseGlobalDataSetList := False;

{ EnabledDataSets plays local list role, you should install it before the template is loaded }
FReport.EnabledDataSets.Add(FfrxDataSet);
FReport.LoadFromFile(ReportName);
FReport.PrepareReport;
```

Reports caching

The reports and it's data can be cached both in memory (for speed increasing) and in file on the disk (for saving RAM recourses). There are several types of caching in FastReport:

- `TfrxReport.EngineOptions.UseFileCache` - if the property is installed in True, than the whole text and objects of built report are saved in temporary file on disk, at that `TfrxReport.EngineOptions.MaxMemoSize` indicates how many MB are meant for the template in RAM.
- `TfrxReport.PreviewOptions.PagesInCache` - the number of pages which can be kept in cache memory greatly increases preview speed, but spends much memory (especially when there are pictures in a template).
- `TfrxReport.PreviewOptions.PictureCacheInFile` - if the property is on, than all the pictures of built report are saved in temporary file on a disk, that greatly reduces memory use in reports with a large amount of pictures, but it reduces the speed.

MDI architecture

In FastReport there is a n opportunity of creating MDI applications both for preview and for designer. The source code of the example is in FastReport Demos\MDI Designer catalogue.

It is worthy to mention that its advisable to create your own `TfrxReport` for each preview window or designer, otherwise all windows will refer to the same report.

Working with a list of variables

The notion of variables was minutely explained in the corresponding chapter. Let us briefly call to mind the main points.

A user can specify one or several variables in a report. A value or an expression, which will be automatically calculated when referring to a variable, can be assigned to every variable. Variables can be visually inserted into a report via the “Data tree” window. It is convenient to use variables for aliasing of compound expressions, which are often used in a report.

It is necessary to use the “frxVariables” unit when working with variables. Variables are represented by the `TfrxVariable` class.

```
TfrxVariable = class(TCollectionItem)
published
  property Name: String; // Name of a variable
  property Value: Variant; // Value of a variable
end;
```

The list of variables is represented by the `TfrxVariables` class. It contains all methods necessary for working with the list.

```
TfrxVariables = class(TCollection)
public
  // Adds a variable to the end of the list
  function Add: TfrxVariable;

  // Adds a variable to the given position of the list
  function Insert(Index: Integer): TfrxVariable;

  // Returns the index of a variable with the given name
  function IndexOf(const Name: String): Integer;

  // Adds a variable to the specified category
  procedure AddVariable(const ACategory, AName: String; const AValue: Variant);

  // Deletes a category and all its variables
  procedure DeleteCategory(const Name: String);

  // Deletes a variable
  procedure DeleteVariable(const Name: String);

  // Returns the list of categories
  procedure GetCategoriesList(List: TStrings; ClearList: Boolean = True);

  // Returns the list of variables in the specified category
  procedure GetVariablesList(const Category: String; List: TStrings);

  // The list of variables
  property Items[Index: Integer]: TfrxVariable readonly;

  // Values of variables
  property Variables[Index: String]: Variant; default;
end;
```

If the list of variables is long, it is convenient to group it by categories. For example, when having the following list

of variables:

```
Customer name
Account number
In total
Total vat
```

one can represent it in the following way:

```
Properties
  Customer name
  Account number
Totals
  In total
  total vat
```

There are the following limitations:

- at least one category must be created
- categories form the first level of the data tree, variables form the second one
- categories cannot be nested
- variables' names must be unique within a whole list, not within a category

Creating a list of variables

A link to the report variables is stored in the `TfrxReport.Variables` property. To create a list manually, the following steps must be performed:

- clear the list
- create a category
- create variables
- repeat the 2 and 3 steps to create another category.

Clearing a list of variables

It is performed with the help of the `TfrxVariables.Clear` method:

Pascal:

```
frxReport1.Variables.Clear;
```

C++:

```
frxReport1->Variables->Clear();
```

Adding a category

It is required to create at least one category. Categories and variables are stored in one list. The category differs from a variable by the “space,” which is the first symbol of the name. All variables located in the list after the category, are considered belonging to this category.

Process of adding a category to the list can be performed in two ways:

Pascal:

```
frxReport1.Variables[' ' + 'My Category 1'] := Null;
```

C++:

```
frxReport1->Variables->Variables[" My Category 1"] = NULL;
```

or

Pascal:

```
var
  Category: TfrxVariable;

Category := frxReport1.Variables.Add;
Category.Name := ' ' + 'My category 1';
```

C++:

```
TfrxVariable * Category;
Category = frxReport1->Variables->Add();
Category->Name = " My category 1";
```

Adding a variable

Variables can be added only after a category is already added. All the variables located in the list after the category, are considered belonging to this category. Variables' names must be unique within the whole list, and not within a category

There are several ways to add a variable to the list:

Pascal:

```
frxReport1.Variables['My Variable 1'] := 10;
```

C++:

```
frxReport1->Variables->Variables["My Variable 1"] = 10;
```

this way adds a variable (if it does not exist already) or modifies a value of the existing variable.

Pascal:

```
var
  Variable: TfrxVariable;

Variable := frxReport1.Variables.Add;
Variable.Name := 'My Variable 1';
Variable.Value := 10;
```

C++:

```
TfrxVariable * Variable;

Variable = frxReport1->Variables->Add();
Variable->Name = "My Variable 1";
Variable->Value = 10;
```

Both of the ways add a variable to the end of the list, therefore, it would be added to the last category. If a variable is supposed to be added to a specified position of the list, use the "Insert" method:

Pascal:

```
var
  Variable: TfrxVariable;

Variable := frxReport1.Variables.Insert(1);
Variable.Name := 'My Variable 1';
Variable.Value := 10;
```

C++:

```
TfrxVariable * Variable;

Variable = frxReport1->Variables->Insert(1);
Variable->Name = "My Variable 1";
Variable->Value = 10;
```

If a variable is to be added to the specified category, use the `AddVariable` method:

Pascal:

```
frxReport1.Variables.AddVariable('My Category 1', 'My Variable 2', 10);
```

C++:

```
frxReport1->Variables->AddVariable("My Category 1", "My Variable 2", 10);
```

Deleting a variable

Pascal:

```
frxReport1.Variables.DeleteVariable('My Variable 2');
```

C++:

```
frxReport1->Variables->DeleteVariable("My Variable 2");
```

Deleting a category

To delete a category with all its variables, use the following code:

Pascal:

```
frxReport1.Variables.DeleteCategory('My Category 1');
```

C++:

```
frxReport1->Variables->DeleteCategory("My Category 1");
```

Modifying the variable’s value

There are two ways to modify the value of a variable:

Pascal:

```
frxReport1.Variables['My Variable 2'] := 10;
```

C++:

```
frxReport1->Variables->Variables["My Variable 2"] = 10;
```

or

Pascal:

```
var
  Index: Integer;
  Variable: TfrxVariable;

{ search for the variable }
Index := frxReport1.Variables.IndexOf('My Variable 2');
{ if it is found, change a value }
if Index <> -1 then
begin
  Variable := frxReport1.Variables.Items[Index];
  Variable.Value := 10;
end;
```

C++:

```
int Index;
TfrxVariable * Variable;

// search for the variable
Index = frxReport1->Variables->IndexOf("My Variable 2");
// if it is found, change a value
if(Index != -1)
{
  Variable = frxReport1->Variables->Items[Index];
  Variable->Value = 10;
}
```

It should be noted, that when accessing a report variable its value is calculated if it is of string type. That means the variable which value is 'Table1."Field1"' will return a value of a DB field, but not the 'Table1."Field1"' string. You should be careful when assigning a string-type values to report variables. For example, the next code will raise exception "unknown variable 'test'" when running a report:

```
frxReport1.Variables['My Variable'] := 'test';
```

because FastReport trying to calculate a value of such variable. The right way to pass a string values is:

```
frxReport1.Variables['My Variable'] := '''' + 'test' + '''';
```

In this case the variable value - string 'test' will be shown without errors. But keep in mind that:

- string should not contain single quotes. All single quotes must be doubled;
- string should not contain #13#10 symbols.

In some cases it is easier to pass variables using a script.

Script variables

Instead of report variables, script variables are in the `TfrxReport.Script`. You can define them using FastScript methods. Let's look at some differences between report and script variables::

	Report variables	Script variables
Placement	In the report variables list, `TfrxReport.Variables`.	In the report script, `TfrxReport.Script.Variables`.
Variable name	May contain any symbols.	May contain any symbols. But if you want to use that variable inside the report script, its name should conform to Pascal identificator specifications.
Variable value	May be of any type. Variables of string type are calculated each time you access them, and are, in itself, an expressions.	May be of any type. No calculation is performed, behavior is like standard language variable.
Accessibility	Programmer can see the list of report variables in the "Data tree" window.	The variable is not visible, programmer should know about it.

Working with script variables is easy. Just assign value to the variable this way:

Pascal:

```
frxReport1.Script.Variables['My Variable'] := 'test';
```

C++:

```
frxReport1->Script->Variables->Variables["My Variable"] = "test";
```

In this case FastReport will create a variable if it is not exists, or assign a value to it. There is no need to use extra quotes when assigning a string to that variable.

Passing a variable value in the TfrxReport.OnGetValue

The last way to pass a value to a report is to use `TfrxReport.OnGetValue` event handler. This way is convenient in case you need to pass a dynamic value (that may change from record to record). Two previous ways are useful to pass static values.

Let's look at example of using that way. Let's create the report and lay the "Text" object to it. Type the following text in this object:

```
[My Variable]
```

Now create the `TfrxReport.OnGetValue` event handler:

```
procedure TForm1.frxReport1GetValue(const VarName: String;
  var Value: Variant);
begin
  if CompareText(VarName, 'My Variable') = 0 then
    Value := 'test'
end;
```

Run the report and we will see that variable is shown correctly. The `TfrxReport.OnGetValue` event handler is called each time when FastReport finds unknown variable. The event handler should return a value of that variable.

Working with styles

First of all, let us call to mind, what “style”, “set of styles” and “library of styles” are.

Style is an element, which possesses a name and properties, and determines design attributes, i.e. color, font and frame. The style determines the way a report object should be designed. The objects such as `TfrxMemoView` have the `Style` property, which is a property intended to set the style name. When applying a value to this property, the style design attributes are copied to the object.

A set of styles consists of several styles, which refer to a report. The `TfrxReport` component has the `Styles` property, which refers to the object of the `TfrxStyles` type. The set of styles also possesses a name. The set of styles determines design of a whole report.

A styles library includes several sets of styles. It is convenient to perform a selection of a concrete set for report design with the help of the library.

The `TfrxStyleItem` represents a style.

```
TfrxStyleItem = class(TCollectionItem)
public
  // Style name.
  property Name: String;

  // Background color.
  property Color: TColor;

  // Font.
  property Font: TFont;

  // Frame.
  property Frame: TfrxFrame;
end;
```

The set of styles is represented by the `TfrxStyles` class. It comprises methods for performing such set operations as reading, saving, adding, deleting, as well as searching for a style. The set of styles file has FS3 extension by default.

```
  TfrxStyles = class(TCollection)
  public
    // Creates the styles set. One can specify “nil” instead of “AReport,” however in this case a user
would be unable to use the “Apply” method.
    constructor Create(AReport: TfrxReport);

    // Adds a new style.
    function Add: TfrxStyleItem;

    // Returns the style with the given name.
    function Find(const Name: String): TfrxStyleItem;

    // Applies a set to a report.
    procedure Apply;

    // Returns the list of the styles names.
    procedure GetList(List: TStrings);

    // Reads a set.
    procedure LoadFromFile(const FileName: String);
    procedure LoadFromStream(Stream: TStream);

    // Saves a set.
    procedure SaveToFile(const FileName: String);
    procedure SaveToStream(Stream: TStream);

    // The list of styles.
    property Items[Index: Integer]: TfrxStyleItem; default;

    // A set’s name.
    property Name: String;
  end;
```

In conclusion, the last `TfrxStyleSheet` class represents a styles’ library. It has methods for the library reading/saving, as well as adding, deleting, and style sets’ searching.

```
TfrxStyleSheet = class(TObject)
public
 // Constructs a library.
  constructor Create;

 // Clears a library.
  procedure Clear;

 // Deletes a set with certain number.
  procedure Delete(Index: Integer);

 // Returns the list of the names of styles' sets.
  procedure GetList(List: TStrings);

 // Loads a library.
  procedure LoadFromFile(const FileName: String);
  procedure LoadFromStream(Stream: TStream);

 // Saves a library.
  procedure SaveToFile(const FileName: String);
  procedure SaveToStream(Stream: TStream);

 // Adds a new set of styles to the library.
  function Add: TfrxStyles;

 // Returns a number of styles' sets in the library.
  function Count: Integer;

 // Returns a set with the given name.
  function Find(const Name: String): TfrxStyles;

 // Returns a set number with the given name.
  function IndexOf(const Name: String): Integer;

 // The list of styles' sets.
  property Items[Index: Integer]: TfrxStyles; default;
end;
```

Creation of style sets

The following code demonstrates processes of creation of styles set, and addition of two styles to a set. After these operations are completed, the styles are applied to the report.

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := TfrxStyles.Create(nil);

{ the first style }
Style := Styles.Add;
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

{ the second style }
Style := Styles.Add;
Style.Name := 'Style2';
Style.Font.Name := 'Times New Roman';
Style.Frame.Typ := [ftLeft, ftRight];

{ apply a set to the report }
frxReport1.Styles := Styles;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = new TfrxStyles(NULL);

// the first style
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

// the second style
Style = Styles->Add();
Style->Name = "Style2";
Style->Font->Name = "Times New Roman";
Style->Frame->Typ << ftLeft << ftRight;

// apply a set to the report
frxReport1->Styles = Styles;
```

You can create and use a set in a different way:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;
Styles.Clear;

{ the first style }
Style := Styles.Add;
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

{ the second style }
Style := Styles.Add;
Style.Name := 'Style2';
Style.Font.Name := 'Times New Roman';
Style.Frame.Typ := [ftLeft, ftRight];

{ apply a set to the report }
frxReport1.Styles.Apply;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;
Styles->Clear();

// the first style
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

// the second style
Style = Styles->Add();
Style->Name = "Style2";
Style->Font->Name = "Times New Roman";
Style->Frame->Typ << ftLeft << ftRight;

// apply a set to the report
frxReport1->Styles->Apply();
```

Modifying/adding/deleting a style

Modifying a style with the given name:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ search for a style}
Style := Styles.Find('Style1');

{ modify the font size }
Style.Font.Size := 12;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// search for a style
Style = Styles->Find("Style1");

// modify the font size
Style->Font->Size = 12;
```

Adding a style to the report styles set:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ add }
Style := Styles.Add;
Style.Name := 'Style3';
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// add
Style = Styles->Add();
Style->Name = "Style3";
```

Deleting a style with a given name:

Pascal:

```
var
  Style: TfrxStyleItem;
  Styles: TfrxStyles;

Styles := frxReport1.Styles;

{ delete }
Style := Styles.Find('Style3');
Style.Free;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// delete
Style = Styles->Find("Style3");
delete Style;
```

After modifications are accomplished, you should call the `Apply` method:

```
{ use modifications }
frxReport1.Styles.Apply;
```

Saving/restoring a set

Pascal:

```
frxReport1.Styles.SaveToFile('c:\1.fs3');
frxReport1.Styles.LoadFromFile('c:\1.fs3');
```

C++:

```
frxReport1->Styles->SaveToFile("c:\\1.fs3");
frxReport1->Styles->LoadFromFile("c:\\1.fs3");
```

Clear report styles

It can be performed in two ways:

```
frxReport1.Styles.Clear;
```

or

```
frxReport1.Styles := nil;
```

Styles library creation

The following example illustrates how to create a library and add two sets of styles to it.

Pascal:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

StyleSheet := TfrxStyleSheet.Create;

{ the first set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles1';

{ here one can add styles to the Styles set}

{ the second set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles2';

{ here one can add styles to the Styles set}
```

C++:

```
TfrxStyles * Styles;
TfrxStyleSheet * StyleSheet;

StyleSheet = new TfrxStyleSheet;

// the first set
Styles = StyleSheet->Add();
Styles->Name = "Styles1";

// here one can add styles to the Styles set

// the second set
Styles = StyleSheet->Add();
Styles->Name = "Styles2";

// here one can add styles to the Styles set
```

Displaying a list of style sets, and application of a selected style

Style libraries are frequently used for displaying accessible style sets in such controls as “ComboBox” or “ListBox.” After that, the set, selected by a user, is applied to a report.

Displaying the list:

```
StyleSheet.GetList(ComboBox1.Items);
```

Usage of the selected set to a report:

```
frxReport1.Styles := StyleSheet.Items[ComboBox1.ItemIndex];
```

or

```
frxReport1.Styles := StyleSheet.Find[ComboBox1.Text];
```

Modification/adding/deleting of a styles set

Modification of a set with the specified name:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

{ search for the required set }
Styles := StyleSheet.Find('Styles2');

{ modify a style with the Style1 name from the found set }
with Styles.Find('Style1') do
  Font.Name := 'Arial Black';
```

Adding a set to a library:

```
var
  Styles: TfrxStyles;
  StyleSheet: TfrxStyleSheet;

{ the third set }
Styles := StyleSheet.Add;
Styles.Name := 'Styles3';
```

Deleting a set from a library:

```
var
  i: Integer;
  StyleSheet: TfrxStyleSheet;

{ search for the third set }
i := StyleSheet.IndexOf('Styles3');

{ if find, delete }
if i <> -1 then
  StyleSheet.Delete(i);
```

Saving and loading a styles library

File extension for the styles library is “FSS” by default.

```
var
  StyleSheet: TfrxStyleSheet;

StyleSheet.SaveToFile('c:\1.fss');
StyleSheet.LoadFromFile('c:\1.fss');
```