

Быстрые отчеты

Руководство программиста FastReport FMX

Версия 2026.2

© 2008-2026 ООО Быстрые отчеты

Работа с компонентом TfrxReport

Чтение и запись отчета

Запуск дизайнера

Выполнение отчета

Предварительный просмотр

Печать отчета

Запись и считывание готового отчета

Экспорт отчета

Создание собственных окон предварительного просмотра

Построение композитных отчетов (пакетная печать)

Интерактивные отчеты

Доступ к объектам отчета из кода

Создание формы отчета с помощью кода

Создание диалоговой формы с помощью кода

Изменение свойств страницы отчета

Построение отчета с помощью кода

Печать массива

Печать TStringList

Печать файла

Печать TStringGrid

Печать TTable, TQuery

Наследование отчетов

Многопоточность

Кэширование отчета

MDI архитектура

Чтение и запись отчета

По умолчанию форма отчета хранится вместе с формой проекта, т.е. в файле DFM. В большинстве случаев этого достаточно и вам не нужно принимать специальных мер для загрузки отчета. Если вы решили хранить форму отчета в файле или в BLOb-поле БД (это дает большую гибкость - вы можете изменять отчет, не перекомпилируя саму программу), вам придется использовать методы компонента TfrxReport для чтения/записи отчета:

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

Загружает отчет из файла с заданным именем. Если второй параметр равен True и файл не найден, то генерирует исключение. Если файл загружен успешно, возвращает True.

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

Загружает отчет из потока.

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

Записывает отчет в файл с заданным именем.

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

Записывает отчет в поток.

> Файл с формой отчета имеет по умолчанию расширение FR3.

Примеры:

Pascal:

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

C++:

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

Запуск дизайнера

Вызов дизайнера отчета осуществляется методом TfrxReport.DesignReport. Дизайнер должен быть включен в ваш проект (для этого достаточно использовать компонент TfrxDesigner или включить модуль frxDesgn в список uses).

Пример:

```
frxReport1.DesignReport;
```

Метод DesignReport имет два параметра по умолчанию:

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

Первый параметр определяет, надо ли запускать дизайнер в модальном режиме; второй - будет ли окно дизайнера являться дочерним (для работы в MDI приложении).

Выполнение отчета

Запуск отчета на выполнение осуществляется одним из двух методов TfrxReport:

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

Запускает отчет на выполнение и показывает результат в окне предварительного просмотра. Если параметр ClearLastReport равен False, то отчет добавляется к ранее построенному, иначе ранее построенный отчет очищается (по умолчанию).

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

Запускает отчет на выполнение, без окна предварительного просмотра. Назначение параметра такое же, как и в методе ShowReport. Если отчет был построен успешно, возвращает True.

В большинстве случаев удобнее использовать первый метод. Он сразу показывает окно предварительного просмотра, отчет тем временем продолжает строиться.

Параметр ClearLastReport удобно использовать в случае, когда к ранее построенному отчету надо добавить еще один (такая техника применяется для пакетной печати отчетов, будет рассмотрена позже).

Пример:

```
frxReport1.ShowReport;
```

Предварительный просмотр

Показать отчет в окне предварительного просмотра можно двумя способами: вызовом метода TfrxReport.ShowReport (см. "Выполнение отчета"), либо с помощью метода TfrxReport.ShowPreparedReport. Во втором случае построения отчета не происходит, а показывается уже готовый отчет. Это значит, что вы должны его предварительно построить с помощью метода PrepareReport, либо загрузить ранее построенный отчет из файла (см. "Запись и считывание готового отчета").

Пример:

Pascal:

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

C++:

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

В этом случае сначала полностью строится отчет, затем показывается в окне просмотра. Построение большого отчета может занять много времени, именно поэтому вместо PrepareReport/ShowPreparedReport лучше использовать асинхронный метод ShowReport.

Задать настройки просмотра по умолчанию можно с помощью свойства TfrxReport.PreviewOptions.

Печать отчета

В большинстве случаев печатать отчет вы будете из окна предварительного просмотра. Распечатать отчет "вручную" можно с помощью метода TfrxReport.Print, например:

```
frxReport1.Print;
```

При этом будет выведен диалог, в котором можно настроить параметры печати. Задать настройки по умолчанию, а также отключить диалог печати можно с помощью свойства TfrxReport.PrintOptions.

Запись и считывание готового отчета

Это можно сделать из окна предварительного просмотра. Вручную это делается с помощью методов TfrxReport.PreviewPages:

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

Назначение и параметры аналогичны соответствующим методам TfrxReport.

Файл с готовым отчетом имеет по умолчанию расширение FP3.

Пример:

Pascal:

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

C++:

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

> Обратите внимание, что после считывания готового отчета его просмотр осуществляется с помощью метода ShowPreparedReport!

Экспорт отчета

Это можно сделать из окна предварительного просмотра. Вручную это делается с помощью метода TfrxReport.Export. В единственном параметре этого метода надо указать тот фильтр экспорта, который вы хотите использовать:

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

Компонент фильтра экспорта должен быть доступен (вы должны положить его на форму своего проекта) и настроен соответствующим образом.

Создание собственных окон предварительного просмотра

FastReport показывает отчеты в стандартном окне предварительного просмотра. Если по каким-то причинам вас это не устраивает, можно создать собственную форму предварительного просмотра. Для этих целей служит компонент TfrxPreview из палитры компонентов FastReport. Чтобы вывести отчет в ваше окно, надо присвоить ссылку на этот компонент свойству TfrxReport.Preview.

При использовании компонента TfrxPreview обычно возникает два вопроса: он не реагирует на клавиши управления (стрелки, PgUp, PgDown и пр.) и на колесо прокрутки мыши (если она им оборудована). Чтобы заставить компонент реагировать на клавиши, достаточно установить фокус управления на этот компонент:

```
frxPreview1.SetFocus;
```

Это можно сделать, например, в событии OnShow формы. Для того, чтобы компонент реагировал на колесо прокрутки мыши, надо создать обработчик события формы OnMouseWheel и вызвать в нем метод TfrxPreview.MouseWheelScroll:

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

Построение композитных отчетов (пакетная печать)

В некоторых случаях требуется организовать печать нескольких отчетов одним заданием, или формирование и отображение нескольких отчетов в одном окне просмотра. Для этого в FastReport есть средства, позволяющие построить новый отчет в дополнение к уже существующему. Метод TfrxReport.PrepareReport имеет один необязательный параметр ClearLastReport: Boolean, по умолчанию он равен True. Этот параметр определяет, надо ли очищать страницы ранее построенного отчета. Следующий пример показывает, как построить пакет из двух отчетов:

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();
```

Мы загружаем первый отчет и строим его, не показывая на экране. Затем загружаем второй отчет в этот же объект TfrxReport и строим его с параметром ClearLastReport = False. При этом отчет добавляется к ранее построенному. После этого мы показываем готовый отчет в окне предварительного просмотра.

Нумерация страниц

Вы можете использовать системные переменные Page, Page#, TotalPages, TotalPages# для отображения номера страницы/общего количества страниц. В композитных отчетах эти переменные работают следующим образом:

`Page` - номер страницы в текущем отчете

`Page#` - номер страницы в пакете

`TotalPages` - всего страниц в текущем отчете (отчет должен быть двухпроходным)

`TotalPages#` - всего страниц в пакете.

Объединение отчетов в пакете

Как говорилось выше, свойство страницы отчета "Печатать на предыдущей странице" (PrintOnPreviousPage) позволяет печатать страницы "внахлест", т.е. на свободном месте предыдущей страницы. В композитных отчетах это позволяет начать формирование нового отчета на свободном месте последней страницы предыдущего отчета. Для этого надо включить свойство "Печатать на предыдущей странице" у первой страницы отчета.

Интерактивные отчеты

В интерактивных отчетах можно определить реакцию на щелчок мышью на том или ином объекте отчета в окне предварительного просмотра. Например, пользователь может щелкнуть мышью на строке данных и тем самым сформировать новый отчет с детальными данными по выбранной строке.

Интерактивным можно сделать любой отчет. Для этого надо всего лишь создать обработчик для события TfrxReport.OnClickObject. Ниже приведен пример такого обработчика:

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;
    }
  }
}
```

В обработчике OnClickObject вы можете делать следующее:

- менять содержимое переданного в обработчик объекта или страницы (при этом надо выставить флаг Modified, чтобы учесть изменения);
- вызывать метод TfrxReport.PrepareReport для построения отчета заново.

В этом примере щелчок на объекте с именем Memo1 выведет сообщение с содержимым этого объекта. При щелчке на Memo2 выведется диалог, в котором можно поменять содержимое объекта. Установка флага Modified в True позволяет зафиксировать изменения и отобразить их на экране.

Таким же образом можно определить другую реакцию на щелчок, например, формирование нового отчета. Здесь необходимо отметить следующее. В версии FastReport 3 один компонент TfrxReport может отображать

отчет только в одном окне предварительного просмотра (в отличие от версии 2). Поэтому формировать новый отчет придется либо в отдельном объекте TfrxReport, либо в том же, но с затиранием текущего отчета.

Для подсказки пользователю, на какие объекты можно щелкать мышью, удобно менять форму курсора при прохождении мыши над объектом. Для этого достаточно установить свойству Cursor у нужных объектов значение, отличное от crDefault. Это делается в дизайнере отчета.

Еще один момент касается определения того, на каком объекте пользователь щелкнул мышью. В простых отчетах это можно определить по имени объекта или его содержимому. Однако это не всегда удается сделать в более сложных случаях. Например, необходимо сформировать детальный отчет по выбранной строке данных. Пользователь щелкнул на объекте Memo1 с содержимым '12'. К какой строке данных относится этот объект? Для этого нужно знать первичный ключ, который однозначно идентифицирует эту строку. FastReport позволяет сопоставить каждому объекту отчета строку, содержащую любые данные (в нашем случае - данные первичного ключа). Эта строка хранится в свойстве TagStr.

Рассмотрим это на примере отчета, который входит в демо FRDemo.exe - 'Simple list'. Это список клиентов некоторой фирмы - имя клиента, адрес, контактное лицо и пр. Источник данных - таблица Customer.db из демонстрационной базы DBDEMOS. У этой таблицы есть первичный ключ - поле CustNo, которое в отчете не присутствует. Наша задача - по щелчку на любом объекте из сформированного отчета определить, к какой записи он относится, т.е. узнать значение первичного ключа. Для этого достаточно занести в свойство TagStr всех объектов, которые лежат на бэнде Master Data, следующее значение:

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

Во время формирования отчета содержимое свойства TagStr вычисляется таким же образом, как и содержимое текстовых объектов, т.е. вместо всех переменных подставляются их значения. Переменная в данном случае - это то, что заключено в квадратные скобки. Поэтому после формирования отчета в свойстве TagStr объектов, лежащих на Master Data, будут содержаться строки типа '1005', '2112' и т.д. Простое преобразование из строки в число даст нам значение первичного ключа, по которому можно найти нужную запись.

Если первичный ключ составной, т.е. содержит несколько полей, то содержимое свойства TagStr может быть следующим:

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

После построения отчета свойство TagStr будет содержать значения типа '1000;1', из которых также нетрудно извлечь значения ключа.

Доступ к объектам отчета из кода

Объекты FastReport (такие, как страница отчета, бэнд, мемо-объект) напрямую недоступны из вашего кода. Это означает, что вы не можете обратиться к объекту по его имени, как, например, обращаетесь к кнопке на вашей форме. Чтобы обратиться к объекту, надо найти его с помощью метода TfrxReport.FindObject:

Pascal:

```
var
  Memo1: TfrxMemoView;

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

C++:

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

после этого можно обращаться к свойствам и методам объекта. К страницам отчета можно обращаться с помощью свойства TfrxReport.Pages:

Pascal:

```
var
  Page1: TfrxReportPage;

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

C++:

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

Обратите внимание - к первой странице отчета надо обращаться по индексу [1]. Индекс 0 имеет страница "Данные".

Создание формы отчета с помощью кода

Как правило, большинство отчетов вы будете создавать с помощью дизайнера. Тем не менее, в некоторых случаях (например, форма отчета заранее неизвестна) необходимо создавать отчет вручную, с помощью кода.

Для создания отчета вручную необходимо выполнить следующие шаги:

- очистить отчет
- добавить источники данных
- добавить страницу "Данные"
- добавить страницу отчета
- добавить бэнды на страницу
- настроить свойства бэндов и подключить их к данным
- добавить объекты на каждый бэнд
- настроить свойства объектов и подключить их к данным

Рассмотрим создание простого отчета типа "список". Предполагается, что у нас есть компоненты frxReport1: TfrxReport и frxDBDataSet1: TfrxDBDataSet (последний подключен к данным из DBDEMOS, таблица Customer.db). Наш отчет будет содержать одну страницу с бэндами report title и master data. На бэнде report title будет объект с текстом "Hello FastReport!", а на master data - объект со ссылкой на поле "CustNo".

Pascal:

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

{ очищаем отчет }
frxReport1.Clear;

{ добавляем источник данных в список доступных для отчета }
frxReport1.DataSets.Add(frxDBDataSet1);

{ добавляем страницу "Данные" }
DataPage := TfrxDataPage.Create(frxReport1);

{ добавляем страницу }
Page := TfrxReportPage.Create(frxReport1);

{ создаем уникальное имя }
Page.CreateUniqueName;

{ устанавливаем размеры полей, бумаги и ориентацию по умолчанию }
Page.SetDefaults;

{ меняем ориентацию бумаги }
Page.Orientation := poLandscape;

{ добавляем report title }
```

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

{ для бэнда достаточно установить координату Top и высоту }
{ обе координаты - в пикселах }
Band.Top := 0;
Band.Height := 20;

{ добавляем объект на report title }
Memo := TfrxMemoView.Create(Band);
Memo.CreateUniqueName;
Memo.Text := 'Hello FastReport!';
Memo.Height := 20;

{ этот объект будет растянут на ширину бэнда }
Memo.Align := baWidth;

{ добавляем master data }
DataBand := TfrxMasterData.Create(Page);
DataBand.CreateUniqueName;
DataBand.DataSet := frxDBDataSet1;

{ координата Top не должна пересекать ранее добавленный бэнд! }
DataBand.Top := 100;
DataBand.Height := 20;

{ добавляем объект на master data }
Memo := TfrxMemoView.Create(DataBand);
Memo.CreateUniqueName;

{ подключаем к данным }
Memo.DataSet := frxDBDataSet1;
Memo.DataField := 'CustNo';
Memo.SetBounds(0, 0, 100, 20);

{ выравниваем текст по правому краю объекта }
Memo.HAlign := haRight;

{ показываем отчет }
frxReport1.ShowReport;
```

C++:

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

// очищаем отчет
frxReport1->Clear();

// добавляем источник данных в список доступных для отчета
frxReport1->DataSets->Add(frxDBDataset1);

// добавляем страницу "Данные"
DataPage = new TfrxDataPage(frxReport1);

// добавляем страницу
Page = new TfrxReportPage(frxReport1);

// создаем уникальное имя
Page->CreateUniqueName();

// устанавливаем размеры полей, бумаги и ориентацию по умолчанию
```

```
Page->SetDefaults();

// меняем ориентацию бумаги
Page->Orientation = poLandscape;

// добавляем report title
Band = new TfrxReportTitle(Page);
Band->CreateUniqueName();

// для бэнда достаточно установить координату Top и высоту
// обе координаты - в пикселах
Band->Top = 0;
Band->Height = 20;

// добавляем объект на report title
Memo = new TfrxMemoView(Band);
Memo->CreateUniqueName();
Memo->Text = "Hello FastReport!";
Memo->Height = 20;

// этот объект будет растянут на ширину бэнда
Memo->Align = baWidth;

// добавляем master data
DataBand = new TfrxMasterData(Page);
DataBand->CreateUniqueName();
DataBand->DataSet = frxDBDataset1;

// координата Top не должна пересекать ранее добавленный бэнд!
DataBand->Top = 100;
DataBand->Height = 20;

// добавляем объект на master data
Memo = new TfrxMemoView(DataBand);
Memo->CreateUniqueName();

// подключаем к данным
Memo->DataSet = frxDBDataset1;
Memo->DataField = "CustNo";
Memo->SetBounds(0, 0, 100, 20);

// выравниваем текст по правому краю объекта
Memo->HAlign = haRight;

// показываем отчет
frxReport1->ShowReport(true);
```

Поясним некоторые моменты.

Все источники данных, которые будут использованы в отчете, необходимо добавить в список источников данных. В нашем случае это делается с помощью строки frxReport1.DataSets.Add(frxDBDataSet1). Если этого не сделать, отчет работать не будет.

Страница "Данные" в отчете должна присутствовать для того, чтобы можно было создавать внутренние источники данных в отчете.

Вызов Page.SetDefaults не обязателен - в этом случае страница будет иметь формат A4 и поля по 0мм. SetDefaults устанавливает поля по 10мм, и берет размер и ориентацию страницы, которую имеет принтер по умолчанию.

Добавляя бэнды на страницу, вы должны следить, чтобы они не пересекались друг с другом. Для этого достаточно установить соответствующие координаты Top и Height. Координаты Left и Width менять смысла не имеет – горизонтальный бэнд всегда имеет ширину страницы, на которой он расположен (это не так, если речь идет о вертикальном бэнде – в этом случае надо устанавливать свойства Left и Width, а свойства Top и

Height не менять). Кроме того, следует помнить, что порядок расположения бэндов на странице имеет значение. Всегда располагайте бэнды таким образом, как вы это сделали бы в дизайнере.

Координаты и размеры объектов задаются в пикселах. Т.к. свойства Left, Top, Width, Height всех объектов имеют тип Extended, вы можете указывать нецелочисленные значения. Для перевода пикселов в сантиметры и дюймы определены следующие константы:

```
fr01cm = 3.77953; // 96 / 25.4
fr1cm  = 37.7953;
fr01in = 9.6;
fr1in  = 96;
```

Например, задать высоту бэнда, равную 5мм, можно так:

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

Создание диалоговой формы с помощью кода

В отчете, как мы знаем, могут содержаться диалоговые формы. Следующий пример демонстрирует создание диалоговой формы, на которой расположена кнопка OK:

Pascal:

```
{ для работы с диалоговыми объектами надо использовать этот модуль }
uses frxDCtrl;

var
  Page: TfrxDialogPage;
  Button: TfrxButtonControl;

{ добавляем страницу }
Page := TfrxDialogPage.Create(frxReport1);

{ создаем уникальное имя }
Page.CreateUniqueName;

{ устанавливаем размеры }
Page.Width := 200;
Page.Height := 200;

{ устанавливаем позицию }
Page.Position := poScreenCenter;

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

{ показываем отчет }
frxReport1.ShowReport;
```

C++:

```
// для работы с диалоговыми объектами надо использовать этот модуль
#include "frxDCtrl.hpp"

TfrxDialogPage * Page;
TfrxButtonControl * Button;

// добавляем страницу
Page = new TfrxDialogPage(frxReport1);

// создаем уникальное имя
Page->CreateUniqueName();

// устанавливаем размеры
Page->Width = 200;
Page->Height = 200;

// устанавливаем позицию
Page->Position = poScreenCenter;

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

// показываем отчет
frxReport1->ShowReport(true);
```

Изменение свойств страницы отчета

Иногда необходимо бывает сменить настройки страницы отчета из кода, например, поменять ориентацию или размер бумаги. Класс TfrxReportPage содержит следующие свойства, определяющие размер листа:

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

Свойство PaperSize задает формат бумаги. Это одно из стандартных значений, определенных в Windows.pas, например, DMPAPER_A4. Если присвоить значение этому свойству, FastReport сам заполнит свойства PaperWidth и PaperHeight (размеры бумаги в миллиметрах). Если в качестве формата задать значение DMPAPER_USER (или 256), это означает, что задан пользовательский размер бумаги. В этом случае свойства PaperWidth и PaperHeight надо заполнить самостоятельно.

Следующий пример показывает, как поменять параметры первой страницы (предполагается, что у нас уже есть отчет):

Pascal:

```
var
  Page: TfrxReportPage;

{ первая страница отчета имеет индекс [1]. Индекс [0] у страницы "Данные". }
Page := TfrxReportPage(frxReport1.Pages[1]);

{ меняем размер }
Page.PaperSize := DMPAPER_A2;

{ меняем ориентацию бумаги }
Page.Orientation := poLandscape;
```

C++:

```
TfrxReportPage * Page;

// первая страница отчета имеет индекс [1]. Индекс [0] у страницы "Данные".
Page = (TfrxReportPage *)frxReport1.Pages[1];

// меняем размер
Page->PaperSize = DMPAPER_A2;

// меняем ориентацию бумаги
Page->Orientation = poLandscape;
```

Построение отчета с помощью кода

Построением отчета обычно занимается ядро FastReport. Оно выводит бэнды отчета в определенной последовательности столько раз, сколько имеется данных, формируя таким образом готовый отчет. Иногда необходимо вывести отчет нестандартной формы, который ядро FastReport сформировать не в состоянии. В этом случае можно воспользоваться возможностью построения отчета вручную, с помощью события TfrxReport.OnManualBuild. Если определить обработчик этого события, ядро FastReport передаст управление ему. При этом распределение обязанностей по формированию отчета меняется следующим образом:

Ядро:

- подготовка отчета к формированию (инициализация скрипта, источников данных, формирование дерева бэндов)
- все вычисления (агрегатные функции, обработчики событий)
- формирование новых страниц/колонок (автоматический вывод page/column header/footer, report title/summary)
- прочая рутинная работа

Обработчик:

- вывод бэндов в определенном порядке

Т.е., суть обработчика OnManualBuild состоит в том, чтобы давать ядру FastReport команды на вывод определенных бэндов. Все остальное ядро сделает самостоятельно: сформирует новую страницу, когда место на текущей закончится, выполнит скрипты и т.д.

Ядро представлено классом TfrxCustomEngine. Ссылка на экземпляр этого класса находится в свойстве TfrxReport.Engine. Ядро имеет следующие свойства и методы:

Свойство или метод	Описание
procedure NewColumn	Формирует новую колонку. Если колонка была последней, формирует новую страницу.
procedure NewPage	Формирует новую страницу.
procedure ShowBand(Band: TfrxBand)	Показывает бэнд.
procedure ShowBand(Band: TfrxBandClass)	Показывает бэнд заданного типа.
function FreeSpace: Extended	Возвращает количество свободного места на странице (в пикселах). После вывода очередного бэнда это значение уменьшается.
property CurColumn: Integer	Возвращает/устанавливает номер текущей колонки.
property CurX: Extended	Возвращает/устанавливает текущую позицию X.

Свойство или метод	Описание
property CurY: Extended	Возвращает/устанавливает текущую позицию Y. После вывода очередного бэнда это значение увеличивается.
property DoublePass: Boolean	Является ли отчет двухпроходным.
property FinalPass: Boolean	Является ли текущий проход последним.
property FooterHeight: Extended	Возвращает высоту page footer.
property HeaderHeight: Extended	Возвращает высоту page header.
property PageHeight: Extended	Возвращает высоту области печати страницы.
property PageWidth: Extended	Возвращает ширину области печати страницы.
property TotalPages: Integer	Возвращает количество страниц в готовом отчете (только на втором проходе двухпроходного отчета).

Приведем пример простого обработчика. В отчете имеется два бэнда master data, не подключенных к данным. Обработчик выведет эти бэнды в чередующемся порядке, каждый по 6 раз. После шести бэндов будет сделан небольшой промежуток.

Pascal:

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

{ находим нужные бэнды }
Band1 := frxReport1.FindObject('MasterData1') as TfrxMasterData;
Band2 := frxReport1.FindObject('MasterData2') as TfrxMasterData;
for i := 1 to 6 do
begin
  { выводим бэнды друг за другом }
  frxReport1.Engine.ShowBand(Band1);
  frxReport1.Engine.ShowBand(Band2);
  { делаем небольшой промежуток }
  if i = 3 then
    frxReport1.Engine.CurY := frxReport1.Engine.CurY + 10;
end;
```

C++:

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

// находим нужные бэнды
Band1 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData1"));
Band2 := dynamic_cast <TfrxMasterData *> (frxReport1->FindObject("MasterData2"));
for(i = 1; i <= 6; i++)
{
  // выводим бэнды друг за другом
  frxReport1->Engine->ShowBand(Band1);
  frxReport1->Engine->ShowBand(Band2);
  // делаем небольшой промежуток
  if(i == 3)
    frxReport1->Engine->CurY += 10;
}
```

Следующий пример выведет две группы бэндов рядом друг с другом.

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;
}
```

Печать массива

Исходный код примера находится в каталоге FastReport Demos\PrintArray (FastReport Demos\BCB Demos\PrintArray). Поясним некоторые моменты.

Для печати массива используем отчет с одним бэндом Master Data, который будет выведен столько раз, сколько элементов в массиве. Для этого положим на форму компонент TfrxUserDataSet и настроим его свойства (можно сделать это в коде, как в нашем примере):

```
RangeEnd := reCount
RangeEndCount := кол-во элементов в массиве
```

После этого подключим дата-бэнд к компоненту TfrxUserDataSet. Для отображения элемента массива положим на бэнд Master Data текстовый объект со строкой [element] внутри. Переменная 'element' заполняется в событии TfrxReport.OnGetValue.

Печать TStringList

Исходный код примера находится в каталоге FastReport Demos\PrintStringList (FastReport Demos\BCB Demos\PrintStringList). Принцип работы абсолютно такой же, как в примере с массивом.

Печать файла

Исходный код примера находится в каталоге FastReport Demos\PrintFile (FastReport Demos\BCB Demos\PrintFile). Поясним некоторые моменты.

Для печати файла используется отчет с бэндом Master Data, который будет напечатан один раз (для этого его надо подключить к источнику данных, содержащему одну запись - выбрать из списка источник с именем "Одна строка"/"Single row"). У бэнда включено растягивание (Stretch) и разрыв (Allow Split). Это означает, что бэнд растянется таким образом, чтобы уместить все имеющиеся на нем объекты. Если при этом бэнд не поместится на странице, он будет выведен по частям на отдельных страницах.

Содержимое файла отображается объектом "Текст", который содержит переменную [file]. Эта переменная, как и в предыдущих примерах, заполняется в событии TfrxReport.OnGetValue. У объекта также включено растягивание (пукнт Stretch из контекстного меню или свойство StretchMode = smActualHeight).

Печать TStringGrid

Исходный код примера находится в каталоге FastReport Demos\PrintStringGrid (FastReport Demos\BCB Demos\PrintStringGrid). Поясним некоторые моменты.

Компонент TStringGrid представляет собой таблицу с произвольным количеством строк и столбцов. Т.е. при печати отчет будет расти не только в высоту, но и в ширину. Чтобы напечатать такой компонент, используем объект Cross-tab (он становится доступен при добавлении в проект компонента TfrxCrossObject). Этот объект как раз предназначен для печати табличных данных с заранее неизвестным количеством строк и столбцов. Объект имеет две разновидности: TfrxCrossView для печати пользовательских данных, и TfrxDBCrossView для печати специально подготовленных данных из таблицы БД.

Используем TfrxCrossView. Объект предварительно надо настроить. Для этого зайдем в дизайнер отчета и вызовем редактор объекта, сделав на нем двойной щелчок мышью. Нам нужно указать степень вложенности заголовков строк и колонок, а также количество значений в ячейках таблицы. В нашем случае все эти значения должны быть равны 1. В нашем примере также отключены заголовки строк и колонок, и суммарные значения по строкам и колонкам.

Заполнять объект значениями из StringGrid необходимо в событии TfrxReport.OnBeforePrint. Значение добавляется с помощью метода TfrxCrossView.AddValue. Его параметры следующие: составной индекс строки, колонки и значение ячейки (также составное - ведь объект может содержать несколько значений в ячейке).

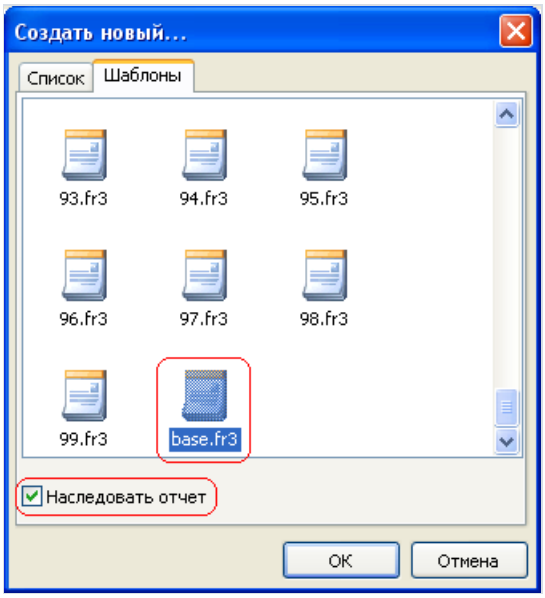
Печать TTable, TQuery

Исходный код примера находится в каталоге FastReport Demos\PrintTable (FastReport Demos\BCB Demos\PrintTable). Принцип работы тот же, что в примере с TStringGrid. В данном случае индекс строки - ее порядковый номер, индекс колонки - имя поля таблицы, значение ячейки - значение поля таблицы. Важный момент - в редакторе объекта Cross-tab надо отключить функции для элемента ячейки (поскольку в ячеке содержатся данные разного характера, это приведет к ошибке формирования таблицы) и отключить сортировку для заголовка колонки (иначе колонки будут отсортированы по алфавиту).

Наследование отчетов

Наследование отчетов было описано в соответствующей главе в "Руководстве пользователя". Здесь мы опишем некоторые ключевые моменты.

Если вы храните отчеты в файлах, вам нужно указать каталог, в котором FastReport будет искать базовые отчеты. Точнее, это каталог, содержимое которого будет показано в окнах "Файл|Новый..." и "Отчет|Настройки...":



Для этого используется свойство компонента TfrxDesigner.TemplateDir. По умолчанию оно пустое, FastReport показывает шаблоны в каталоге, где находится исполняемый файл (.exe). В это свойство можно поместить либо абсолютный путь, либо относительный.

Если вы храните отчеты в базе данных, придется написать немного кода для загрузки базового отчета и для получения списка доступных шаблонов. Для загрузки базового отчета используйте обработчик события TfrxReport.OnLoadTemplate:

```
property OnLoadTemplate: TfrxLoadTemplateEvent read FOnLoadTemplate write FOnLoadTemplate;
TfrxLoadTemplateEvent = procedure(Report: TfrxReport; const TemplateName: String) of object;
```

Обработчик этого события должен загрузить отчет с именем TemplateName в компонент Report. Вот возможный пример такого обработчика:

```
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;
```

Для получения списка доступных шаблонов (в окнах "Файл|Новый..." и "Отчет|Настройки...") надо использовать обработчик события TfrxDesigner.OnGetTemplateList:

```
property OnGetTemplateList: TfrxGetTemplateListEvent read FOnGetTemplateList write FOnGetTemplateList;
TfrxGetTemplateListEvent = procedure(List: TStrings) of object;
```

Обработчик этого события должен загрузить список доступных отчетов в параметр List. Вот возможный пример такого обработчика:

```
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 может наследовать и уже созданные отчеты, для этого нужно воспользоваться функцией

```
TfrxReport.InheritFromTemplate(const templName: String; InheriteMode: TfrxInheriteMode = imDefault):
Boolean.
```

Функция позволяет наследовать текущий загруженный отчет от указанного отчета. Первый параметр функции - путь и имя шаблона предка, второй - параметр позволяет выбрать режим наследования(imDefault - выводить диалог с предложением переименовать/удалить дубликаты, imDelete - удаляь все дублирующиеся объекты, imRename - переименовывать все дублирующиеся объекты).

> Внимание! Поиск шаблона предка осуществляется относительно текущего шаблона, т.е. необходимо соблюдать структуру каталогов в месте хранения отчетов. Fast Report использует относительные пути, поэтому не стоит беспокоиться о переносе приложения (исключением является, когда текущий шаблон и

шаблон-предок расположены на разных носителях или используется сетевой путь).

Многопоточность

FastReport может работать независимо в разных потоках, но имеется ряд особенностей:

- Нельзя создавать TfrxDBDataSet с одинаковыми именами даже в разных потоках, т.к. для поиска используется "глобальный список" и обращение всегда будет происходить к первому созданному TfrxDBDataSet (использование глобального списка можно отключить, по умолчанию он активен);
- Если во врем выполнения отчета происходит изменение св-в объектов(для примера Memo1.Left := Memo1.Left + 10 в скрипте), то следует помнить, что при последующим выполнении если св-во TfrxReport.EngineOptions.DestroyForms := False шаблон отчета уже будет модифицирован и его необходимо заново загрузить или использовать TfrxReport.EngineOptions.DestroyForms := True. При восстановлении из потока нельзя использовать интерактивные отчеты, т.к. объекты скрипта уничтожаются после восстановления, поэтому в некоторых случаях рациональней использовать TfrxReport.EngineOptions.DestroyForms := False и восстанавливать шаблон самостоятельно при следующем цикле построения.

При необходимости глобальный список, по которому осуществляется поиск экземпляров TfrxDBDataSet, можно отключить.

```
{ DestroyForms можно отключить, если каждый раз восстанавливать отчет из файла или потока }
FReport.EngineOptions.DestroyForms := False;
FReport.EngineOptions.SilentMode := True;

{ Данное св-во отключает поиск по глобальному списку }
FReport.EngineOptions.UseGlobalDataSetList := False;

{ EnabledDataSets играет роль локального списка, нужно устанавливать до загрузки шаблона }
FReport.EnabledDataSets.Add(FfrxDataSet);
FReport.LoadFromFile(ReportName);
FReport.PrepareReport;
```

Кэширование отчета

Отчеты и их данные можно кэшировать как в памяти (для увеличения быстродействия), так и в файл на диске (для экономии ресурсов ОЗУ).

Можно выделить следующие виды кэширования в Fast Report:

- TfrxReport.EngineOptions.UseFileCache - если св-во установлено в True, то весь текст и объекты построенного отчета сохраняются во временный файл на диске, при этом TfrxReport.EngineOptions.MaxMemoSize указывает, сколько мегабайт отводится шаблону в ОЗУ.
- TfrxReport.PreviewOptions.PagesInCache - кол-во страниц которое может хранится в памяти, существенно ускоряет работу предварительного просмотра, но расходует много памяти (особенно если в шаблоне есть рисунки) .
- TfrxReport.PreviewOptions.PictureCacheInFile - если св-во включено, то все картинки построенного отчета сохраняются во временный файл на диске, значительно уменьшает потребление памяти в отчетах с больным кол-вом рисунков, но снижает быстродействие.

MDI архитектура

В Fast Report есть возможность создания MDI приложений, как для предварительного просмотра, так и для дизайнера. Пример Исходный код примера находится в каталоге FastReport Demos\MDI Designer.

Стоит отметить, что для каждого окна предварительного просмотра или дизайнера нужно создавать свой TfrxReport, иначе все окна будут ссылаться на один и тот же отчет.

Работа со списком переменных

Переменные были подробно рассмотрены в соответствующей главе руководства пользователя. Кратко напомним суть. В отчете можно определить одну или несколько переменных. Каждой переменной можно присвоить значение или выражение, которое будет автоматически вычисляться при обращении к переменной. Переменные можно визуально вставлять в отчет, пользуясь окном "Дерево данных". Переменные удобно использовать для замещения сложных выражений, которые часто используются в отчете.

Для работы с переменными необходимо использовать модуль frxVariables. Переменная представлена классом TfrxVariable.

```
TfrxVariable = class(TCollectionItem)
published
  // Имя переменной
  property Name: String;

  // Значение переменной
  property Value: Variant;
end;
```

Список переменных представлен классом TfrxVariables. Он содержит все необходимые методы для работы со списком.

```
TfrxVariables = class(TCollection)
public
  // Добавляет переменную в конец списка
  function Add: TfrxVariable;

  // Добавляет переменную в указанную позицию списка
  function Insert(Index: Integer): TfrxVariable;

  // Возвращает индекс переменной с заданным именем
  function IndexOf(const Name: String): Integer;

  // Добавляет переменную в заданную категорию
  procedure AddVariable(const ACategory, AName: String; const AValue: Variant);

  // Удаляет категорию и все ее переменные
  procedure DeleteCategory(const Name: String);

  // Удаляет переменную
  procedure DeleteVariable(const Name: String);

  // Возвращает список категорий
  procedure GetCategoriesList(List: TStrings; ClearList: Boolean = True);

  // Возвращает список переменных в заданной категории
  procedure GetVariablesList(const Category: String; List: TStrings);

  // Список переменных
  property Items[Index: Integer]: TfrxVariable readonly;

  // Значения переменных
  property Variables[Index: String]: Variant; default;
end;
```

Если список переменных велик, удобно сгруппировать его по категориям. Например, имея список переменных:

```
название предприятия
р/счет
итого
итого ндс
```

мы можем представить его в виде:

```
Реквизиты
  название предприятия
  р/счет
Итоги
  итого
  итого ндс
```

Имеются следующие ограничения:

- обязательно наличие хотя бы одной категории
- категории образуют первый уровень дерева данных, переменные - второй
- категории не могут быть вложенными
- имена переменных должны быть уникальны в пределах всего списка, а не в пределах категории

Создание списка переменных

Ссылка на список переменных отчета хранится в свойстве TfrxReport.Variables. Для создания списка вручную надо выполнить следующие шаги:

- очистить список
- создать категорию
- создать переменные
- повторить шаги 2 и 3 для создания других категорий.

Очистка списка переменных

Производится с помощью метода TfrxVariables.Clear:

Pascal:

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

C++:

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

Добавление категории

Вы должны обязательно создать хотя бы одну категорию. Физически и переменные, и категории хранятся в одном списке. Категория отличается от переменной наличием пробела в качестве первого символа имени. Все переменные, находящиеся в списке после категории, считаются принадлежащими этой категории.

Добавить категорию в список можно двумя способами:

Pascal:

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

C++:

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

или

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";
```

Добавление переменной

Переменные можно добавлять после того, как добавлена категория. Все переменные, находящиеся в списке после категории, считаются принадлежащими этой категории. Имена переменных должны быть уникальны в пределах всего списка, а не в пределах категории.

Добавить переменную в список можно несколькими способами:

Pascal:

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

C++:

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

этот способ добавляет переменную, если ее не существует, либо изменяет значение существующей переменной.

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;
```

Оба способа добавляют переменную в конец списка, таким образом, она добавляется в последнюю категорию. Если переменную надо добавить в определенную позицию списка, используйте метод Insert:

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;
```

Если надо добавить переменную в определенную категорию, используйте метод AddVariable:

Pascal:

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

C++:

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

Удаление переменной

Pascal:

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

C++:

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

Удаление категории

Для удаления категории вместе со всеми ее переменными используйте следующий код:

Pascal:

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

C++:

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

Изменение значения переменной

Можно сделать двумя способами:

Pascal:

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

C++:

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

или

Pascal:

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

{ ищем переменную }
Index := frxReport1.Variables.IndexOf('My Variable 2');

{ если нашли, меняем значение }
if Index <> -1 then
begin
  Variable := frxReport1.Variables.Items[Index];
  Variable.Value := 10;
end;
```

C++:

```
int Index;
TfrxVariable * Variable;

// ищем переменную
Index = frxReport1->Variables->IndexOf("My Variable 2");

// если нашли, меняем значение
if(Index != -1)
{
  Variable = frxReport1->Variables->Items[Index];
  Variable->Value = 10;
}
```

Следует отметить важный момент. При обращении к переменной, определенной в списке переменных, происходит вычисление ее значения, если тип переменной – строковый. Это означает, что переменная со значением Table1."Field1" вернет на самом деле значение поля БД, а не строку 'Table1."Field1"'. При присвоении строковых значений таким переменным следует быть осторожным. Например, при выполнении отчета следующий код вызовет ошибку "переменная test не определена":

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

поскольку FastReport при обращении к переменной My Variable будет пытаться вычислить ее значение. Правильно передавать строковые переменные надо так:

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

В этом случае значение переменной – строка 'test' – будет выведено без ошибок. Однако учтите два важных ограничения при использовании такого метода:

- в строке не должно быть одинарных кавычек. Все одинарные кавычки надо продублировать;
- в строке не должно быть символов #13#10.

Учитывая вышесказанное, в некоторых случаях удобнее передавать значения переменных через скриптовые переменные.

Скриптовые переменные

В отличие от рассмотренных выше переменных, которые физически находятся в списке переменных отчета, скриптовые переменные объявляются средствами FastScript и находятся в скрипте отчета. Рассмотрим отличия переменных:

	Переменные отчета	Переменные скрипта
Расположение	В списке переменных отчета, TfrxReport.Variables.	В скрипте, TfrxReport.Script.Variables.
Название переменной	Может содержать любые символы.	Может содержать любые символы, но если вы обращаетесь к переменной в скрипте отчета, ее имя должно соответствовать требованиям к идентификатору языка Pascal.
Значение переменной	Может быть любого типа. Переменные строкового типа вычисляются каждый раз при обращении к ним, т.е. по сути являются выражениями.	Может быть любого типа. Вычисления значения не происходит, переменные ведут себя так же как и языковые переменные.
Визуальность	Список переменных отчета доступен в окне "Дерево данных".	Переменная в отчете не видна, программист должен знать о ее наличии.

Работа со скриптовыми переменными очень проста: достаточно объявить переменную и присвоить ей значение. Обе эти операции делаются одной строкой:

Pascal:

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

C++:

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

При этом, если переменная не существовала, она создается (переменная типа Variant), иначе ей просто присваивается значение. Как вы уже поняли, при передаче строковых значений в данном случае дополнительные кавычки не нужны.

Передача переменных в TfrxReport.OnGetValue

Наконец, последний способ, позволяющий передать значение переменной в отчет, это использование обработчика события TfrxReport.OnGetValue. Этот способ удобен тем, что позволяет передавать динамические данные (меняющиеся от записи к записи), тогда как два предыдущих способа позволяют передавать только статические данные.

Рассмотрим применение данного способа на примере. Создадим отчет и положим на лист объект "Текст" со следующим текстом внутри:

```
[My Variable]
```

Теперь создадим обработчик события TfrxReport.OnGetValue:

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

Запустив отчет, мы увидим, что переменная отображена правильно. Обработчик события OnGetValue вызывается, если в тексте обнаружена неизвестная переменная. Обработчик должен вернуть значение переменной.

Работа со стилями

Для начала напомним, что такое стиль, набор стилей, библиотека стилей.

Стиль - это элемент, который имеет имя и свойства, определяющие оформление - цвет, шрифт, рамка. Стиль задает оформление объекта отчета. Объекты типа TfrxMemoView имеют свойство Style: String, которое задает имя стиля. При присвоении значения этому свойству оформление стиля копируется в объект.

Набор стилей - это несколько стилей, относящихся к одному отчету. Компонент TfrxReport имеет свойство Styles, которое ссылается на внутренний объект типа TfrxStyles. Набор стилей также имеет имя. Набор стилей задает оформление целого отчета.

Библиотека стилей - это несколько наборов стилей. С помощью библиотеки стилей удобно осуществлять выбор конкретного набора для оформления отчета.

Класс TfrxStyleItem представляет собой стиль.

```
TfrxStyleItem = class(TCollectionItem)
public
  // Имя стиля.
  property Name: String;

  // Цвет фона.
  property Color: TColor;

  // Шрифт.
  property Font: TFont;

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

Набор стилей представлен классом TfrxStyles. Он содержит методы для считывания/сохранения набора, добавления, удаления, а также поиска стиля. Файл набора стилей имеет по умолчанию расширение FS3.

```
  TfrxStyles = class(TCollection)
  public
    // Создает набор стилей. Вместо AReport можно указать nil, но в этом случае методом Apply нельзя
будет пользоваться.
    constructor Create(AReport: TfrxReport);

    // Добавляет новый стиль.
    function Add: TfrxStyleItem;

    // Возвращает стиль с указанным именем.
    function Find(const Name: String): TfrxStyleItem;

    // Применяет набор к отчету.
    procedure Apply;

    // Возвращает список имен стилей.
    procedure GetList(List: TStrings);

    // Считывает набор.
    procedure LoadFromFile(const FileName: String);
    procedure LoadFromStream(Stream: TStream);

    // Сохраняет набор.
    procedure SaveToFile(const FileName: String);
    procedure SaveToStream(Stream: TStream);

    // Список стилей.
    property Items[Index: Integer]: TfrxStyleItem; default;

    // Имя набора.
    property Name: String;
  end;
```

Наконец, последний класс TfrxStyleSheet представляет собой библиотеку стилей. Он имеет методы для считывания/сохранения библиотеки, добавления и удаления, а также поиска набора стилей.

```
TfrxStyleSheet = class(TObject)
public
  // Создает библиотеку.
  constructor Create;

  // Очищает библиотеку.
  procedure Clear;

  // Удаляет набор с заданным номером.
  procedure Delete(Index: Integer);

  // Возвращает список имен наборов стилей.
  procedure GetList(List: TStrings);

  // Загружает библиотеку.
  procedure LoadFromFile(const FileName: String);
  procedure LoadFromStream(Stream: TStream);

  // Сохраняет библиотеку.
  procedure SaveToFile(const FileName: String);
  procedure SaveToStream(Stream: TStream);

  // Добавляет новый набор стилей в библиотеку.
  function Add: TfrxStyles;

  // Возвращает количество наборов стилей в библиотеке.
  function Count: Integer;

  // Возвращает набор с заданным именем.
  function Find(const Name: String): TfrxStyles;

  // Возвращает номер набора с заданным именем.
  function IndexOf(const Name: String): Integer;

  // Список наборов стилей.
  property Items[Index: Integer]: TfrxStyles; default;
end;
```

Создание набора стилей

Следующий код демонстрирует создание набора стилей и добавление двух стилей в набор. После этого стили применяются к отчету.

Pascal:

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

Styles := TfrxStyles.Create(nil);

{ первый стиль }
Style := Styles.Add;
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

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

{ применяем набор к отчету }
frxReport1.Styles := Styles;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = new TfrxStyles(NULL);

// первый стиль
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

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

// применяем набор к отчету
frxReport1->Styles = Styles;
```

Можно создать и применить набор иначе:

Pascal:

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

Styles := frxReport1.Styles;
Styles.Clear;

{ первый стиль }
Style := Styles.Add;
Style.Name := 'Style1';
Style.Font.Name := 'Courier New';

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

{ применяем набор к отчету }
frxReport1.Styles.Apply;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

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

// первый стиль
Style = Styles->Add();
Style->Name = "Style1";
Style->Font->Name = "Courier New";

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

// применяем набор к отчету
frxReport1->Styles->Apply();
```

Изменение/добавление/удаление стиля

Изменение стиля с заданным именем:

Pascal:

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

Styles := frxReport1.Styles;

{ ищем первый стиль }
Style := Styles.Find('Style1');

{ меняем размер шрифта }
Style.Font.Size := 12;
```

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

// ищем первый стиль
Style = Styles->Find("Style1");

// меняем размер шрифта
Style->Font->Size = 12;
```

Добавление стиля к набору стилей отчета:

Pascal:

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

Styles := frxReport1.Styles;

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

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

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

Удаление стиля с заданным именем:

Pascal:

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

Styles := frxReport1.Styles;

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

C++:

```
TfrxStyleItem * Style;
TfrxStyles * Styles;

Styles = frxReport1->Styles;

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

После внесенных изменений надо сделать Apply:

```
{ применяем изменения }
frxReport1.Styles.Apply;
```

Сохранение/восстановление набора

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");
```

Очищение стилей отчета

Можно сделать двумя способами:

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

или

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

Создание библиотеки стилей

Следующий пример создает библиотеку и добавляет в нее два набора стилей.

Pascal:

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

StyleSheet := TfrxStyleSheet.Create;

{ первый набор }
Styles := StyleSheet.Add;
Styles.Name := 'Styles1';

{ здесь можно добавить стили в набор Styles }

{ второй набор }
Styles := StyleSheet.Add;
Styles.Name := 'Styles2';

{ здесь можно добавить стили в набор Styles }
```

C++:

```
TfrxStyles * Styles;
TfrxStyleSheet * StyleSheet;

StyleSheet = new TfrxStyleSheet;

// первый набор
Styles = StyleSheet->Add();
Styles->Name = "Styles1";

// здесь можно добавить стили в набор Styles

// второй набор
Styles = StyleSheet->Add();
Styles->Name = "Styles2";

// здесь можно добавить стили в набор Styles
```

Отображение списка наборов стилей и применение выбранного стиля

Наиболее частое применение библиотеки стилей - это отображение доступных наборов стилей в элементе управления типа ComboBox или ListBox. Выбранный пользователем набор затем применяется к отчету.

Отображение списка:

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

Применение выбранного набора к отчету:

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

или

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

Изменение/добавление/удаление набора стилей

Изменение набора с указанным именем:

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

{ ищем нужный набор }
Styles := StyleSheet.Find('Styles2');

{ изменяем стиль с именем Style1 из найденного набора }
with Styles.Find('Style1') do
  Font.Name := 'Arial Black';
```

Добавление набора в библиотеку:

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

{ третий набор }
Styles := StyleSheet.Add;
Styles.Name := 'Styles3';
```

Удаление набора из библиотеки:

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

{ ищем третий набор }
i := StyleSheet.IndexOf('Styles3');

{ если нашли, удаляем }
if i <> -1 then
  StyleSheet.Delete(i);
```

Сохранение/восстановление библиотеки

По умолчанию расширение файла для библиотеки стилей - FSS.

```
var
  StyleSheet: TfrxStyleSheet;

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