Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Институт повышения квалификации
и переподготовки кадров

Кафедра «Информатика»

А. И. Рябченко

СРЕДСТВА ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ

КУРС ЛЕКЦИЙ
по одноименной дисциплине
для слушателей специальности 1-40 01 73
«Программное обеспечение
информационных систем»
заочной формы обучения

Гомель 2013

УДК 004.421+004.43(075.8)
ББК 32.973.26-018.1я73
Р98

Рекомендовано кафедрой «Информатика» ГГТУ им. П. О. Сухого
(*протокол № 13 от 24.05.2013 г.*)

Рецензент: проф. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
д-р техн. наук *И. А. Мурашко*

Рябченко, А. И.

Р98 Средства визуального программирования приложений : курс лекций по одноим. дисциплине для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заоч. формы обучения / А. И. Рябченко. – Гомель : ГГТУ им. П. О Сухого, 2013. – 168 с. Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://library.gstu.by. – Загл. с титул. экрана.

Представлены теоретические сведения по дисциплине «Средства визуального программирования приложений». Рассмотрены вопросы, связанные с разработкой приложений с использованием библиотеки визуальных компонентов Delphi.

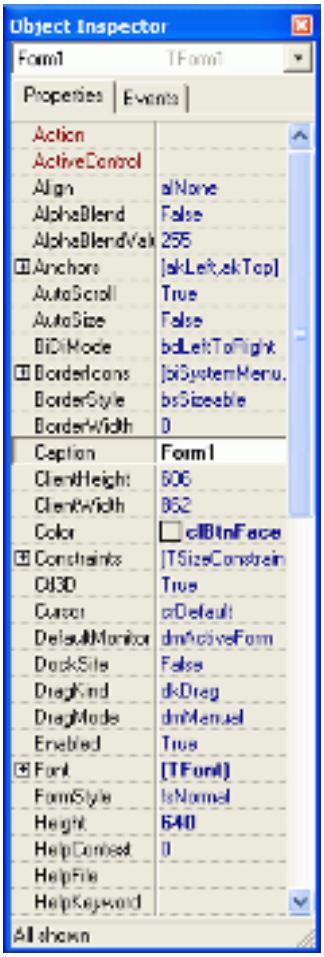
Приведен обзор визуальных компонент, применяемых при разработке типовых приложений в ОС Windows. Большое внимание уделено разработке программ для работы с локальными и сетевыми базами данных с использованием технологий BDE и ADO. Описаны возможности создания визуальных приложений, способных формировать предварительный просмотр и печать данных, использовать COM-технологии для передачи данных в стандартные приложения пакета MS Office.

Для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения ИПК и ПК.

УДК 004.421+004.43(075.8)
ББК 32.973.26-018.1я73

Тема 1. Создание приложений с использованием визуальных компонент Delphi

Основным контейнером для размещения визуальных компонент при программировании в Delphi является форма. Рассмотрим ее основные свойства.



Action - определяет объект *TAction*. Это объект служит для быстрой привязки действий к компонентам, в особенности - к пунктам меню и панелям инструментов. Но может быть привязан и к форме. Для управления *TAction* служат редакторы *TActionList* со страницы *Standard* и *TActionManager* со страницы *Additional*.

ActiveControl - определяет элемент, который имеет в данный момент фокус ввода. Если выбрать какой-либо объект во время разработки (design-time), то при запуске приложения этот объект и будет иметь фокус ввода. Также свойство может быть полезно и во время выполнения (run-time) - можно узнать, какой объект "держит" фокус в данный момент, а также можно переместить фокус на любой из объектов.

Пример: разместим на форме 2 кнопки - Button1 и Button2, а также *TTimer* (страница *System*). Выбрав элемент Timer1, дважды щёлкнем в Инспекторе объектов напротив надписи OnTimer на вкладке Events, т.е. создадим обработчик события и напишем следующее: *ActiveControl:=Button2;* Теперь, запустив программу, каждую секунду фокус будет перемещаться на Button2.

Align - определяет выравнивание формы на экране. Свойство принимает одно из следующих значений:

- alBottom - по нижнему краю;
- alClient - вся пользовательская (клиентская) область;
- alCustom - выравнивание определяется вызовом методом объекта-родителя;
- alLeft - по левому краю;
- alNone - без выравнивания;
- alRight - по правому краю;
- alTop - по верхнему краю.

AlphaBlend - включает/выключает прозрачность формы.

AlphaBlend = True

AlphaBlendValue - задаёт степень непрозрачности формы: 0 - форма полностью невидима, 255 - полностью видима. Прозрачность активируется только при установке свойства AlphaBlend в True.

Anchors - определяет направления, по которым компоненты "привязываются" к форме. Пример: если установить у формы значения akLeft и akRight этого свойства в True, и точно также сделать у кнопки, то при изменении ширины формы размер кнопки (ширина) также будет изменяться.

AutoScroll - включает автоматическое появление полос прокрутки (Scroll bars) на форме, когда размеров формы недостаточно для отображения всех элементов.

AutoSize - включает автоматическое изменение размеров формы согласно позициям размещённых на ней элементов.

BiDiMode - определяет двунаправленное отображение элемента. В некоторых языках письмо осуществляется не слево-направо, а наоборот. Это свойство создано как раз для этой цели.

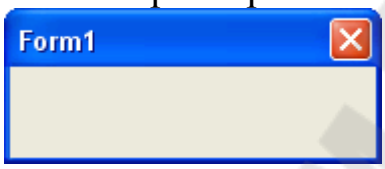
BorderIcons - определяет множество кнопок, которые отображаются в заголовке окна:

- biSystemMenu - единственный элемент, который не является кнопкой - отвечает за системное меню окна, которое вызывается комбинацией клавиш [Alt]+[Пробел].
- biMinimize - кнопка сворачивания (минимизации) окна;
- biMaximize - кнопка разворачивания окна;
- biHelp - кнопка справки.

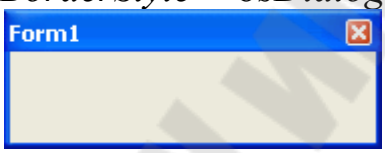
Если хотя бы одна из кнопок сворачивания и разворачивания включена, то независимо от состояния другой, отображаются обе (но вторая естественно неактивна). Если выключены обе, они не отображаются вообще. Это не зависит от Delphi - так устроена ОС Windows.

BorderStyle - определяет поведение границ окна и общий тип окна:

- bsDialog - диалоговое окно (из кнопок - только "Закрыть", иконки в заголовке окна нет);
- bsNone - "чистый лист" (отсутствие у окна границ и заголовка) - применяется обычно для создания заставок во время запуска программы;
- bsSingle - обычное окно, но с запретом изменения размеров;
- bsSizeable - обычное окно (по умолчанию) - размеры формы можно изменять;
- bsSizeToolWin - упрощённое окно с уменьшенным заголовком;
- bsToolWindow - упрощённое окно с уменьшенным заголовком без возможности изменения размеров.



BorderStyle = bsDialog



BorderStyle = bsToolWindow

BorderWidth - ширина границы окна в пикселах. Граница является невидимой и расположена в пользовательской части формы.

Caption - текст заголовка формы.

ClientHeight, **ClientWidth** - размер клиентской (пользовательской) части формы, т.е. той, на которой располагаются компоненты.

Color - цвет формы.

Constraints - определяет минимальные и максимальные размеры высоты и ширины формы в пикселах. 0 - любое значение, т.е. без ограничений.

Ctl3D - свойство определяет 3D-вид формы. При выключенном - "плоское" изображение.

Cursor - курсор мыши в тот момент, когда он находится над формой.

DefaultMonitor - определяет, на каком мониторе появится форма. Имеет смысл применять это свойство только при наличии более, чем одного монитора (например, если несколько экранов).

DockSite, **DragKing** и **DragMode** - определяют поведение формы при осуществлении операций Drag&Drop.

Enabled - отвечает за общую активность формы. Если установлено в False, форма недоступна.

Font - шрифт, используемый на форме.

FormStyle - стиль формы или её поведение в MDI-приложении (многооконное приложение, где дополнительные формы располагаются "внутри" основной формы). Значения:

- fsNormal - обычная форма (значение по умолчанию);
- fsMDIChild - дочерняя (подчинённая) форма MDI-приложения;
- fsMDIForm - главная форма MDI-приложения;
- fsStayOnTop - форма находится поверх всех окон на экране.

Height - высота формы в пикселах. В отличие от ClientWidth является высотой с учётом заголовка и границ формы.

HelpContext, **HelpFile**, **HelpKeyword**, **HelpType** - свойства для связи формы с файлом справки в формате **.hlp*.

Hint - текст всплывающей подсказки.

HorzScrollBar - свойство определяет внешний вид и поведение горизонтальной полосы прокрутки окна.

Icon - значок (иконка) формы. Отображается в заголовке слева от заголовка. Задаётся файлом в формате **.ico*.

KeyPreview - если свойство установлено в True, то при нажатии клавиш сначала будут вызываться обработчики формы, а только затем обработчики того компонента, который в данный момент имеет фокус ввода. События, связанные с нажатием клавиш - OnKeyDown(), OnKeyPress(), OnKeyUp().

Left - позиция формы на экране (левого верхнего угла) в пикселах.

Menu - позволяет выбрать один из компонентов-меню, который станет главным меню окна, т.е. будет отображаться вверху.

Name - имя формы как объекта. Может содержать только латинские буквы, цифры и знак подчёркивания, и не может начинаться с цифры. Фактически, это то имя, по которому в программе можно обратиться к форме.

ObjectMenuItem - используется при работе с OLE-объектами и позволяет связать пункт меню и OLE-объект: когда объект выделен, пункт меню активен и наоборот.

OldCreateOrder - определяет, когда происходят события OnCreate() и OnDestroy() формы. Если установлено в False, то OnCreate() произойдёт после вызова всех конструкторов, а OnDestroy() - после вызова всех деструкторов. Начальное значение - False, изменять не рекомендуется.

ParentBiDiMode - изменение свойства BiDiMode согласно значению объекта-предка формы.

ParentFont - изменение шрифта (Font) согласно значению объекта-предка.
PixelsPerInch - пропорции шрифта в системе (точек на дюйм).
PopupMenu - позволяет указать контекстное меню (объект TPopupMenu) для формы. Это меню вызывается нажатием правой кнопки мыши.
Position - определяет начальную позицию формы на экране, т.е. в момент её появления. Основные значения:

- poDesigned - появление в том месте, в каком форма расположена в design-time;
- poDesktopCenter - по центру рабочего стола (рекомендуемое значение);
- poScreenCenter - по центру экрана;
- poMainFormCenter - по центру главной формы приложения (для главной формы не имеет смысла).

PrintScale - определяет размеры формы при выводе её изображения на печать.
Scaled - включает масштабирование формы в соответствии с заданным значением свойства PixelsPerInch.
ScreenSnap - если установлено в True, то форма будет автоматически "прилипать" к краям экрана в момент перемещения.
SnapBuffer - определяет расстояние (в пикселах), на котором форма будет "прилипать" к краю экрана.
ShowHint - включает/выключает показ всплывающей подсказки (Hint).
Tag - специальное свойство, которое есть у всех объектов. Специального применения для этого свойства нет, поэтому оно используется для разных целей в конкретной ситуации. Свойство удобно в том случае, если нужно хранить некоторое целое число - не придётся заводить дополнительную переменную.
Top - позиция формы (левого верхнего угла) на экране в пикселах.
TransparentColor - включает/выключает прозрачность определённого цвета формы.
TransparentColorValue - задаёт цвет, который будет прозрачным.

TransparentColor = True

UseDockManager - используется при реализации Drag&Drop технологии, предоставляя дополнительные возможности этого метода взаимодействия.

VertScrollBar - определяет внешний вид и поведение вертикальной полосы прокрутки окна.

Visible - определяет видимость формы на экране.

Width - ширина окна в пикселах, включая границы.

WindowMenu - свойство-аналог свойства Menu, но используемое при создании MDI-форм.

WindowState - одно из состояний окна:

- wsNormal - обычное состояние (занимает часть экрана);
- wsMinimized - окно свёрнуто;
- wsMaximized - окно развёрнуто на весь экран.

Примечания

Стоит сделать несколько примечаний насчёт свойств.

- Свойства прозрачности формы (*AlphaBlend*, *AlphaBlendValue*, *TransparentColor* и *TransparentColorValue*) корректно работают только на ОС Windows XP и следующих версиях. В предыдущих версиях ОС изменение значения этих свойств не производит визуального изменения формы.
- Свойства, названия которых начинаются со слова *Parent* (англ. - *родитель*), в большинстве случаев связывают значения некоторых свойств со значениями соответствующих свойств объекта-родителя. Так, кнопка (TButton) имеет свойство ParentFont и свойство Font, отвечающее за шрифта текста на этой кнопке. Но и сама форма имеет свойство Font. В результате, если у кнопки установить *ParentFont* в True, а затем изменить шрифт у формы, то шрифт у кнопки изменится соответствующим образом. Это позволяет быстро изменять одни и те же свойства у большого числа компонент. Другие

подобные свойства - *ParentShowHint*, *ParentColor*, *ParentBiDiMode*.

- Свойство *Cursor*, отвечающее за курсор, есть у большинства компонент. Но при перемещении курсора его вид изменяется на тот, который задан у самого "дальнего" объекта. Т.е. если и формы и у кнопки заданы разные формы курсора, то при перемещении над кнопкой будет использоваться курсор, заданный у самой кнопки. Число "вложений" одних компонент в другие может быть довольно большим.

Разработка MDI-приложения

MDI расшифровывается как multiple document interface (многодокументный интерфейс). В приложениях с MDI, в основном (родительском) окне можно окрыть более одного дочернего окна. Данная возможность обычно используется в электронных таблицах или текстовых редакторах.

Каждое MDI приложение имеет три основные составляющие:

1. Одну (и только одну) родительскую форму MDI,
2. Одну и более (обычно больше) дочерних форм MDI,
3. основное меню MDI.

MDI "мать"

Как уже упоминалось, в проекте MDI приложения может присутствовать только один MDI контейнер (родительская форма) и он должен быть стартовой формой.

Для создания основного окна MDI приложения проделайте следующие шаги:

- Запустите Delphi и выберите File | New Application... Delphi создаст новый проект с одной формой под названием form1 (по умолчанию).
- В свойстве Name присвойте форме имя frMain.
- Установите свойство FormStyle в fsMDIform.
- Сохраните этот проект (имя проекта на Ваше усмотрение, например prMDIExample), вместе с uMain.pas в только что созданной директории.

Как Вы успели заметить, для создания основной формы MDI, мы установили свойство FormStyle в fsMDIform. В каждом приложении только одна форма может иметь свойство fsMDIform.

MDI "дети"

Каждое родительское окно MDI нуждается по крайней мере в одной дочерней форме. Дочерние формы MDI - это простые формы,

за исключением того, что их видимая часть ограничена размерами родительского окна. Так же при минимизации такого окна, оно помещается не в панель задач, а остаётся внутри родительского окна (на панель задач попадёт только родительское окно).

Теперь давайте создадим дополнительные формы, а точнее дочерние. Просто выберите File | New Form. Будет создан новый объект формы с именем form1 (по умолчанию). При помощи Object Inspector измените свойство Name в форме form1 на frChild, а свойство FormStyle на fsMDIChild. Сохраните эту форму с соответствующим ей файлом как uchild.pas. Обратите внимание, что при помощи данного свойства мы можем превратить любую существующую форму в дочернюю форму MDI.

Ваше приложение может включать множество дочерних MDI форм такого же или другого типа.

MDI приложение может включать в себя и самые обычные формы, но в отличие от дочерних, они будут отображаться как обычные модальные диалоговые окна (такие как about box, или файловый диалог).

Естественно, что как на родительском так и на дочернем окнах можно располагать любые элементы управления, однако уже давно сложилась традиция, что на родительской форме располагается панель статуса (status bar) и панель инструментов (toolbar), в то время как на дочерних формах располагаются все остальные контролы, такие как гриды, картинки, поля вводи и т. д.

Автосоздание -> Доступные

Теперь давайте произведём некоторые настройки нашего проекта. Выберите Project | Options, откроется диалог опций проекта (Project Options). В левой панели выберите frChild (Авто-создание форм ("Auto-create forms")), и переместите её в правую панель (Доступные формы (Available forms)). Список правой панели содержит те формы, которые используются Вашим приложением, но которые не созданы автоматически. В MDI приложении, по умолчанию, все дочерние формы создаются автоматически и отображаются в родительской форме.

Создание и отображение...

Как упомянуто выше, настройка не позволяет автоматически создавать дочерние окна, поэтому нам необходимо добавить некоторый код, который будет производить создание объекта формы frChild. Следующую функцию CreateChildForm необходимо

поместить внутри основной формы (MDI родитель) (наряду с заголовком в interface's private):

```
uses uchild;
...
procedure TfrMain.CreateChildForm (const childName : string);
var Child: TfrChild;
begin
   Child := TfrChild.Create(Application);
    Child.Caption := childName;
end;
```

Данный код создаёт одну дочернюю форму с заголовком childName. Не забудьте, что этот код находится разделе "uses uchild".

На закрытие не минимизировать!

Закрытие дочернего окна в MDI приложении всего-навсего минимизирует его в клиентской области родительского окна. Поэтому мы должны обеспечить процедуру OnClose, и установить параметр Action в caFree:

Code:

```
 procedure TfrChild.FormClose(Sender:TObject; var Action: TCloseAction);
begin
    Action := caFree;
end;
```

Обратите внимание, что если форма является дочерней формой MDI, и её свойство BorderIcons установлено в biMinimize (по умолчанию), то опять же по умолчанию параметр Action установлен в caMinimize. Если же в дочерней форме MDI нет этих установок, то по умолчанию Action установлен как caNone, означающий, что при закрытии формы ничего не случится.

MDI родительское меню

Каждое MDI приложение должно иметь основное меню с (если больше ничего нет), опцией выравнивания окон. Поскольку мы предварительно переместили дочернюю форму из Авто-создаваемых (Auto-create) в Доступные (Available) формы, то нам нужен будет код, который (пункт меню) будет создавать дочерние формы.

Для создания дочерних окон в нашем приложении будет использоваться пункт меню "New child". Второе меню (Window)

будет использоваться для выравнивания дочерних окошек внутри родительского окна-формы.

...Создать и отобразить

Нам необходимо сделать обработчик для пункта меню "New child". При нажатии на пунк меню File | New Child нашего приложения, будет вызываться процедура NewChild1Click которая в свою очередь будет вызывать процедуру CreateChildForm (приведённую выше), для создания (следующего) экземпляра формы frChild.

Code:

```
procedure TfrMain.NewChild1Click(Sender: TObject);
begin
  CreateChildForm('Child '+IntToStr(MDIChildCount+1));
end;
```

Только что созданная дочерняя форма будет иметь заголовок в виде "Child x", где x представляет количество дочерних форм внутри MDI формы, как описано ниже.

Закрыть всё

При работе с приложением, имеющим многодокументный интерфейс, всегда необходимо иметь процедуру, закрывающую все дочерние окна.

Code:

```
procedure TfrMain.CloseAll1Click(Sender: TObject);
var i: integer;
begin
for i:= 0 to MdiChildCount - 1 do
   MDIChildren[i].Close;
end;
```

Вам придётся выполнять проверку на предмет наличия несохраненной информации в каждом дочернем окне. Для решения данной задачи лучше всего использовать обработчик события OnCloseQuery.

Свойства MdiChildCount и MDIChildren

MdiChildCount свойство read only, содержащее в себе количество созданных дочерних окошек. Если не создано ни одно

дочернее окно, то это свойство установлено в 0. Нам придётся частенько использовать MdiChildCount наряду с массивом MDIChildren. Массив MDIChildren содержит ссылки на объекты TForm всех дочерних окошек.

Обратите внимание, что MDIChildCount первого созданного дочернего окна равен 1.

Меню Window

Delphi обеспечивает большинство команд, которые можно поместить внутри пункта меню Window. Далее приведён пример вызова трёх основных методов для команд, которые мы поместили в наше приложение:

Code:

```
procedure TfrMain.Cascade1Click(Sender: TObject);
begin
Cascade;
end;

procedure TfrMain.Tile1Click(Sender: TObject);
begin
Tile;
end;
 procedure TfrMain.ArrangeAll1Click(Sender: TObject);
begin
ArrangeIcons;
end;
```

Главное меню — компонент MainMenu

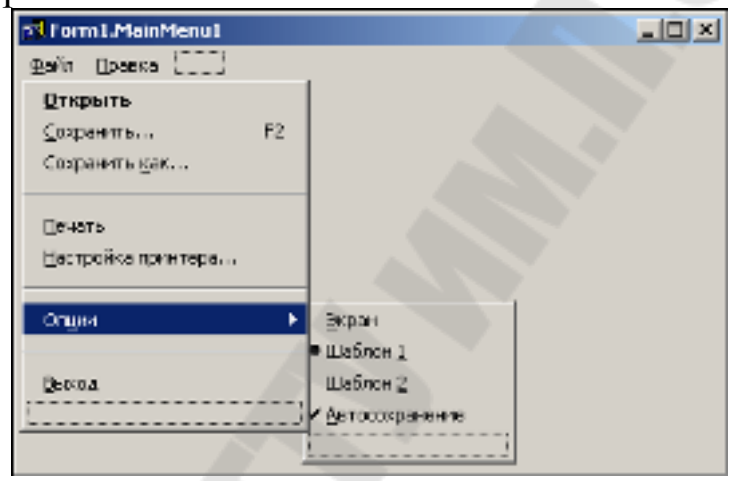
В Delphi имеется два компонента, представляющие меню: MainMenu — главное меню, и PopupMenu — всплывающее меню. Оба компонента расположены на странице "Standard". Эти компоненты имеют много общего. Начнем рассмотрение с компонента MainMenu.

Это невизуальный компонент, т.е. место его размещения на форме в процессе проектирования не имеет никакого значения для пользователя — он все равно увидит не сам компонент, а только меню, сгенерированное им.

Обычно на форму помещается один компонент MainMenu. В этом случае его имя автоматически заносится в свойство формы Menu. Но можно поместить на форму и несколько компонентов

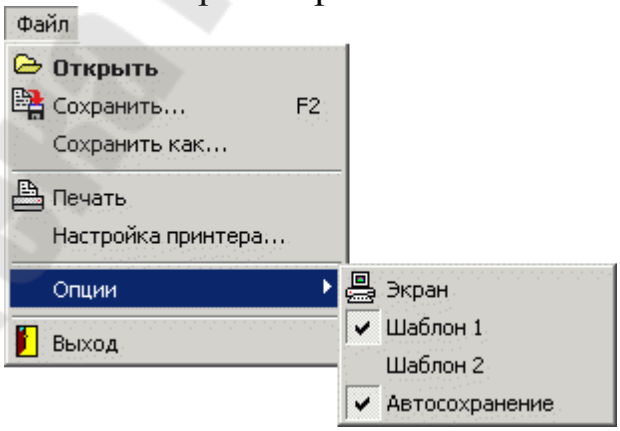
MainMenu с разными наборами разделов, соответствующими различным режимам работы приложения. В этом случае во время проектирования свойству Menu формы присваивается ссылка на один из этих компонентов. А в процессе выполнения в нужные моменты это свойство можно изменять, меняя соответственно состав главного меню приложения.

Основное свойство компонента — Items. Его заполнение производится с помощью Конструктора Меню, вызываемого двойным щелчком на компоненте MainMenu или нажатием кнопки с многоточием рядом со свойством Items в окне Инспектора Объектов. В результате откроется окно.



Окно Конструктора Меню

В этом окне вы можете спроектировать все меню.



Результат конструирования меню

При работе в конструкторе меню новые разделы можно вводить, помещая курсор в рамку из точек, обозначающую место расположения нового раздела. Если при этом раздел ввелся не на нужном вам месте, вы можете отбуксировать его мышью туда, куда вам надо. Другой путь ввода нового раздела — использование

контекстного меню, всплывающего при щелчке правой кнопкой мыши. Если вы предварительно выделите какой-то раздел меню и выберите из контекстного меню команду "Insert", то рамка нового раздела вставится перед ранее выделенным. Из контекстного меню вы можете также выполнить команду "Create Submenu", позволяющую ввести подменю в выделенный раздел (см. подменю раздела "Опции").

При выборе нового раздела вы увидите в Инспекторе Объектов множество свойств данного раздела. Дело в том, что каждый раздел меню, т.е. каждый элемент свойства Items, является объектом типа TMenuItem, обладающим своими свойствами, методами, событиями.

Свойство **Caption** обозначает надпись раздела. Заполнение этого свойства подчиняется тем же правилам, что и заполнение аналогичного свойства в кнопках, включая использование символа амперсанта для обозначения клавиш быстрого доступа. Если вы в качестве значения Caption очередного раздела введете символ минус «-», то вместо раздела в меню появится разделитель.

Разделители после разделов "Сохранить как", "Настройка принтера" и "Опции").

Свойство **Name** задает имя объекта, соответствующего разделу меню. Очень полезно давать этим объектам осмысленные имена, так как иначе вы скоро запутаетесь в ничего не говорящих именах типа "N21". Куда понятнее имена типа "MFile", "MOpen", "MSave" и т.п.

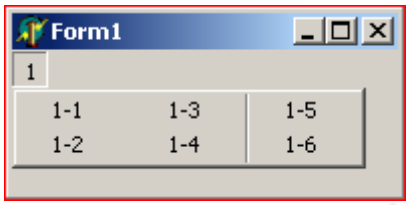
Свойство **Shortcut** определяет клавиши быстрого доступа к разделу меню — «горячие» клавиши, с помощью которых пользователь, даже не заходя в меню, может в любой момент вызвать выполнение процедуры, связанной с данным разделом. Чтобы определить клавиши быстрого доступа, надо открыть выпадающий список свойства Shortcut в окне Инспектора Объектов и выбрать из него нужную комбинацию клавиш. Эта комбинация появится в строке раздела меню.

Свойство **Default** определяет, является ли данный раздел разделом по умолчанию своего подменю, т.е. разделом, выполняемым при двойном щелчке пользователя на родительском разделе. Подменю может содержать только один раздел по умолчанию, выделяемый жирным шрифтом (см. раздел "Открыть").

Свойство **Break** используется в длинных меню, чтобы разбить список разделов н а несколько столбцов. Возможные значение **Break**:

mbNone — отсутствие разбиения меню (это значение принято по умолчанию), mbBarBreak и mbBreak — в меню вводится новый столбец разделов, отделенный от предыдущего полосой (mbBarBreak) или пробелами (mbBreak).

Пример, в котором в разделе 1-3 установлено значение Break = mbBreak, а в разделе 1-5 — Break = mbBarBreak.



Свойство **Checked**, установленное в true, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран (раздел «"Автосохранение"»). Правда, сам по себе этот маркер не изменяется и в обработчик события OnClick такого раздела надо вставлять оператор типа

MAutoSave.Checked := not MAutoSave.Checked;

(в приведенном операторе подразумевается, что раздел меню назван MAutoSave).

Еще одним свойством, позволяющим вводить маркеры в разделы меню, является **RadioItem**. Это свойство, установленное в true, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства **GroupIndex**.

По умолчанию значение **GroupIndex** равно 0. Но можно задать его большим нуля и тогда, если имеется несколько разделов с одинаковым значением **GroupIndex** и с **RadioItem** = **true**, то в них могут появляться маркеры флажков, причем только в одном из них (свойство **RadioItem** установлено в **true** в разделах "Шаблон 1" и "Шаблон 2", имеющих одинаковое значение **GroupIndex**). Если вы зададите программно в одном из этих разделов Checked = true, то в остальных разделах **Checked** автоматически сбросится в false. Впрочем, установка **Checked** = **true** лежит на программе; эта установка может выполняться аналогично приведенному выше оператору.

Описанные маркеры флажков в режиме радиокнопок и в обычном режиме используются для разделов меню, представляющих собой различные опции, взаимоисключающие или совместимые.

Для каждого раздела могут быть установлены во время проектирования или программно во время выполнения свойства **Enabled** (доступен) и **Visible** (видимый).

Если установить **Enabled** = **false**, то раздел будет изображаться серой надписью и не будет реагировать на щелчок пользователя. Если же задать **Visible** = **false**, то раздел вообще не будет виден, а остальные разделы сомкнутся, заняв место невидимого. Свойства **Enabled** и **Visible** используются для того, чтобы изменять состав доступных пользователю разделов в зависимости от режима работы приложения.

Начиная с Delphi 4 предусмотрена возможность ввода в разделы меню изображений. За это ответственны свойства разделов **Bitmap** и **ImageIndex**. Первое из них позволяет непосредственно ввести изображение в раздел, выбрав его из указанного вами файла. Второе позволяет указать индекс изображения, хранящегося во внешнем компоненте **ImageList**. Указание на этот компонент вы можете задать в свойстве Images компонента **MainMenu**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), изображения не будет.

Мы рассмотрели все основные свойства объектов, соответствующих разделам меню. Основное событие раздела — OnClick, возникающее при щелчке пользователя на разделе или при нажатии «горячих» клавиш быстрого доступа.

Рассмотрим теперь вопросы объединения главных меню вторичных форм с меню главной формы. Речь идет о приложениях с несколькими формами, в которых и главная, в вспомогательные формы имеют свои главные меню — компоненты MainMenu. Конечно, пользователю неудобно работать одновременно с несколькими окнами, каждое из которых имеет свое меню. Обычно надо, чтобы эти меню сливались в одно меню главной формы.

Приложения с несколькими формами могут быть двух видов: приложения с интерфейсом множества документов — так называемые MDI приложения, и обычные приложения с главной и вспомогательными формами. Типичными примерами приложений MDI являются программы Word и Excel. Рассмотрение особенностей этих видов приложений выходит за рамки данной книги. Сейчас нас интересует только один вопрос: как объединяются меню различных форм. В MDI приложениях меню дочерних форм всегда объединяются с меню родительской формы. А в приложениях с

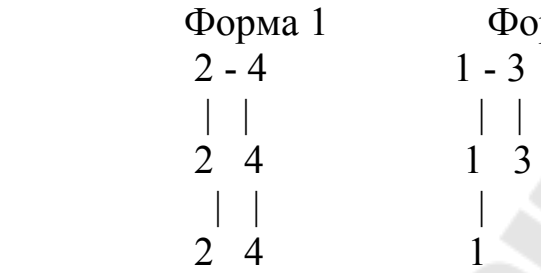
несколькими формами наличие или отсутствие объединения определяется свойством **AutoMerge** компонентов **TMainMenu**. Если требуется, чтобы меню вторичных форм объединялись с меню главной формы, то в каждой такой вторичной форме надо установить **AutoMerge** в **true**. При этом свойство **AutoMerge** главной формы должно оставаться в **false**.

Способ объединения меню определяется свойством разделов **GroupIndex**. По умолчанию все разделы меню имеют одинаковое значение **GroupIndex**, равное нулю.

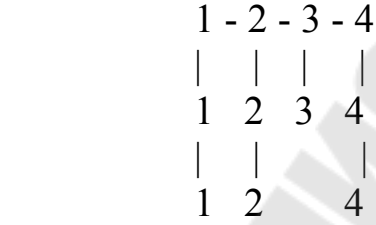
Если требуется объединение меню, то разделам надо задать неубывающие номера свойств **GroupIndex**. Тогда, если разделы встраиваемого меню имеют те же значения **GroupIndex**, что и какие-то разделы меню основной формы, то эти разделы заменяют соответствующие разделы основного меню. В противном случае разделы вспомогательного меню встраиваются между элементами основного меню в соответствии с номерами **GroupIndex**. Если встраиваемый раздел имеет **GroupIndex** меньший, чем любой из разделов основного меню, то разделы встраиваются в начало.

Тогда в момент, когда активизируется вторая форма, в первой появляется меню со структурой:

Пусть, например, в основной и вторичной формах структуры меню имеет следующие значения **GroupIndex**:



Тогда в момент, когда активизируется вторая форма, в первой появляется меню со структурой:



В этом примере отсутствовали разделы, имеющие в обеих формах одинаковые значения **GroupIndex**. Если бы такие были, то при активизации второй формы соответствующие разделы ее меню заменили бы аналогичные разделы первой формы.

Если в меню имеются разделы, работающие как радиокнопки, то нельзя забывать, что их взаимодействие также определяется свойствами **GroupIndex**.

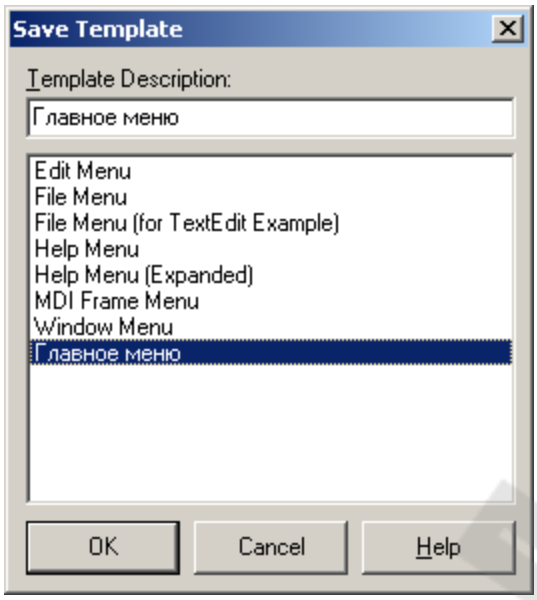
Теперь остановимся на одном из вопросов, связанных с меню в упоминавшийся выше приложениях MDI. В них пользователь может открывать сколько ему требуется окон документов. Обычно в подобных приложениях имеется меню "Окно", которое содержит такие разделы, как "Новое", "Упорядочить" и т.п. Последним идет обычно список открытых окон документов, в который заносятся названия открытых пользователем окон. Выбирая в этом списке, пользователь может переключаться между окнами документов.

Меню «"Окно"» в приложении MDI
со списком открытых документов

Для включения в меню раздела списка открытых окон, надо в свойстве "WindowMenu" главной формы приложения MDI указать имя меню, в конец которого должен помещаться список. Указывается именно имя меню, а не разделов выпадающего списка. Для примера должно быть указано имя элемента меню, соответствующего команде "Окно".

Одним из безусловных требований, предъявляемых к меню приложений для Windows, является стандартизация меню и их разделов. Этому помогает команда "Save As Template..." в контекстном меню, всплывающем при щелчке правой кнопкой мыши в окне Конструктора Меню.

Эта команда вызывает диалог, представленный ниже. В этом диалоге вы можете в верхнем окне указать описание (заголовок), под которым хотите сохранить ваше меню. Впоследствии в любом вашем новом приложении вы можете загрузить этот шаблон в меню, выбирая из всплывающего меню в окне Конструктора Меню команду "Insert From Template...".



Окно сохранения шаблона разработанного меню

Контекстное всплывающее меню — компонент PopupMenu

Контекстное меню привязано к конкретным компонентам. Оно всплывает, если во время, когда данный компонент в фокусе, пользователь щелкнет правой кнопкой мыши. Обычно в контекстное меню включают те команды главного меню, которые в первую очередь могут потребоваться при работе с данным компонентом.

Контекстному меню соответствует компонент PopupMenu. Поскольку в приложении может быть несколько контекстных меню, то и компонентов PopupMenu может быть несколько. Оконные компоненты: панели, окна редактирования, а также метки и др. имеют свойство PopupMenu, которое по умолчанию пусто, но куда можно поместить имя того компонента PopupMenu, с которым будет связан данный компонент.

Формирование контекстного всплывающего меню производится с помощью Конструктора Меню, вызываемого двойным щелчком на PopupMenu, точно так же, как это делалось для главного меню. Обратим только внимание на возможность упрощения этой работы. Поскольку разделы контекстного меню обычно повторяют некоторые разделы уже сформированного главного меню, то можно обойтись копированием соответствующих разделов.

Для этого, войдя в Конструктор Меню из компонента PopupMenu, щелкните правой кнопкой мыши и из всплывшего меню выберите команду "Select Menu" (выбрать меню). Вам будет предложено диалоговое окно, в котором вы можете перейти в главное меню. В нем вы можете выделить нужный вам раздел или разделы

(при нажатой клавише Shift выделяются разделы в заданном диапазоне, при нажатой клавише Ctrl можно выделить совокупность разделов, не являющихся соседними). Затем выполните копирование их в буфер обмена, нажав клавиши Ctrl-C. После этого опять щелкните правой кнопкой мыши, выберите команду "Select Menu" и вернитесь в контекстное меню. Укажите курсором место, в которое хотите вставить скопированные разделы, и нажмите клавиши чтения из буфера обмена — Ctrl-V. Разделы меню вместе со всеми их свойствами будут скопированы в создаваемое вами контекстное меню.

В остальном работа с PopupMenu не отличается от работы с MainMenu. Только не возникает вопросов объединения меню разных форм: контекстные меню не объединяются.

Многостраничные панели

Многостраничные панели позволяют экономить пространство окна приложения, размещая на одном и том же месте страницы разного содержания. На рисунке показаны различные формы отображения многостраничного компонента PageControl. Начнем рассмотрение многостраничных панелей именно с этого компонента.

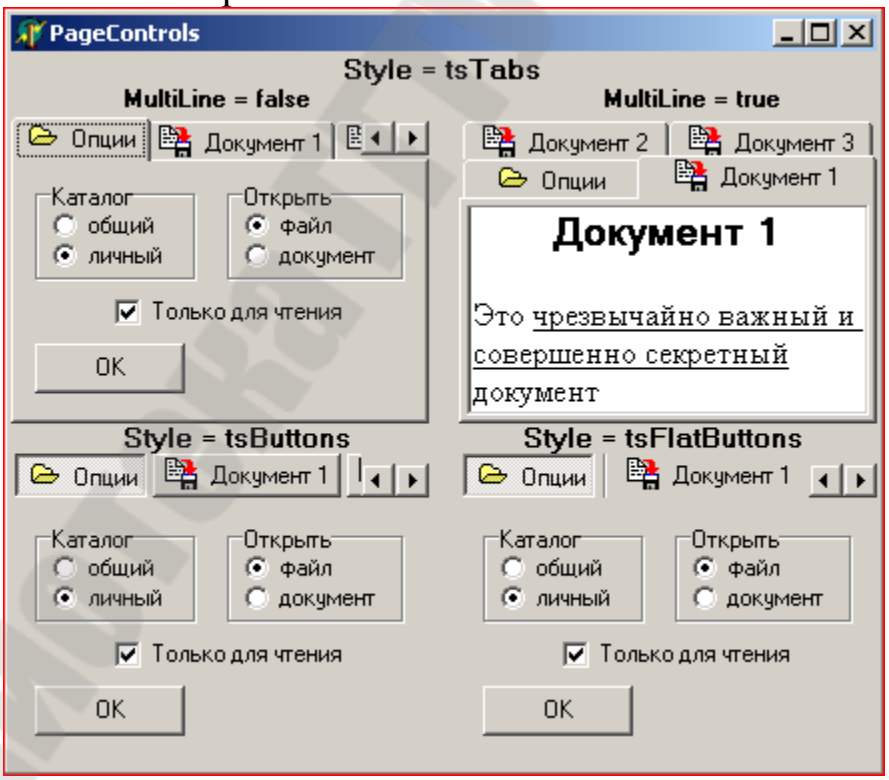


Иллюстрация различных вариантов панели PageControl

Перенесите компонент **PageControl** на форму. Чтобы задавать и редактировать страницы этого компонента, надо щелкнуть на нем правой кнопкой мыши. Во всплывшем меню вы можете видеть

команды: "New Page" — создать новую страницу, "Next Page" — переключиться на следующую страницу, "Previous Page" — переключиться на предыдущую страницу.

Каждая создаваемая вами страница является объектом типа **TTabSheet**. Это панель, на которой можно размещать любые управляющие компоненты, окна редактирования и т.п. После того, как вы создадите несколько страниц, выделите одну из них, щелкнув в ее середине, и посмотрите ее свойства в Инспекторе Объектов.

Страница имеет следующие основные свойства:

Name Имя, по которому можно ссылаться на страницу

Caption Надпись, которая появляется на ярлычке закладки

PageIndex Индекс страницы, по которому можно ссылаться на страницу

ImageIndex Индекс изображения, которое может появляться на ярлычке закладки

Из общих свойств компонента **PageControl** можно отметить:

Style Определяет стиль отображения компонента:

tsTabs — закладки (верхние компоненты),

tsButtons — кнопки (левый нижний компонент),

tsFlatButtons — плоские кнопки (правый нижний компонент)

MultiLine Определяет, будут ли закладки размещаться в несколько рядов, если все они не помещаются в один ряд (два одинаковых компонента, но в левом MultiLine = false, а в правом — true; примером компонента с MultiLine = false является также знакомая вам палитра компонентов в Delphi)

TabPosition Определяет место расположения ярлычков закладок: tpBottom — внизу, tpLeft — слева, tpRight — справа и tpTop — вверху компонента (это значение по умолчанию и именно оно задано)

TabHeight и **TabWidth** Высота и ширина ярлычков закладок в пикселях. Если значения этих параметров заданы равными 0, то размеры ярлычков определяются автоматически по размерам надписей на них

Images Ссылка на компонент ImageList (см. раздел 9.3), который содержит список изображений на ярлычках. Свойства ImageIndex страниц содержат индексы, соответствующие именно этому списку

ScrollOpposite Определят способ перемещения закладок при размещении их в несколько рядов (опробуйте экспериментально, как это свойство влияет на поведение компонента)

ActivePage Имя активной страницы

Pages[Index: Integer] Доступ к странице по индексу (первая страница имеет индекс 0). Свойство только для чтения

PageCount Количество страниц. Свойство только для чтения

В компоненте имеется ряд методов, позволяющих оперировать страницами, создавать их, уничтожать, переключать. Посмотрите их во встроенной справке Delphi.

Основные события компонента — **OnChanging** и **OnChange**. Первое из них происходит непосредственно перед переключением на другую страницу после щелчка пользователя на новой закладке. При этом в обработчик события передается по ссылке параметр **AllowChange** — разрешение переключения. Если в обработчике задать **AllowChange = false**, то переключение не произойдет. Событие **OnChange** присходит сразу после переключения.

Тема 2. Работа с записями и файлами в Delphi

Работа с записями

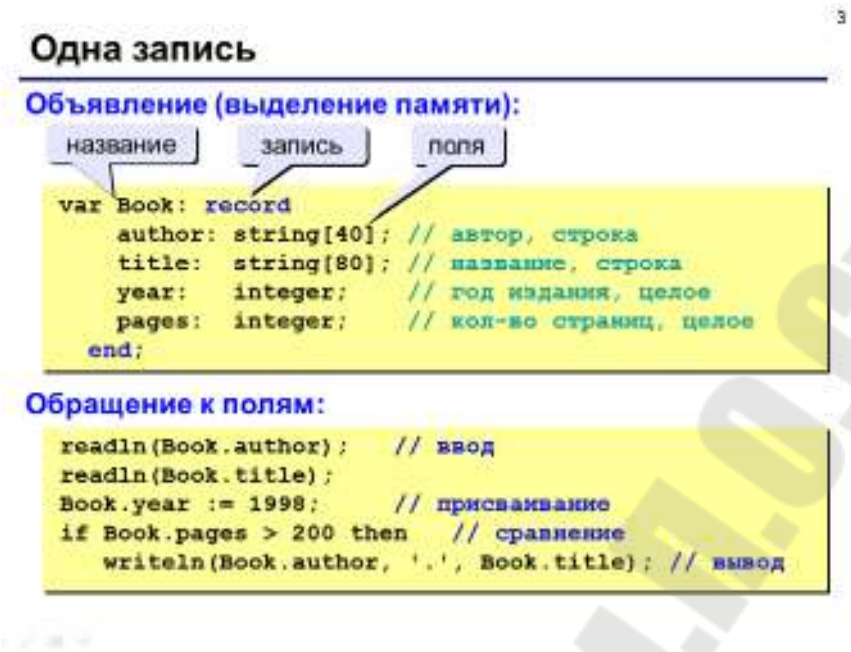
Обращение к записи в целом допускается только в операторах присваивания, где слева и справа от знака присваивания используются имена записей одинакового типа.

Во всех остальных случаях оперируют отдельными полями записей.

Чтобы обратиться к отдельной компоненте записи, необходимо задать имя записи и через точку указать имя нужного поля, например:

str.fio, str.tel

Такое ***имя*** называется ***составным***.



Запись может входить в состав данных более сложной структуры.

Можно говорить, например, о массивах и файлах, состоящих из записей.

Запись может быть полем другой записи.

Оператор присоединения

Обращение к компонентам записей можно упростить, если воспользоваться оператором присоединения With.

Он позволяет заменить составные имена, характеризующие каждое поле, просто на имена полей, а имя записи определить в операторе присоединения.

```
With   <переменная-запись>
     {, <переменная-запись>}  do
                  <оператор> ;
```

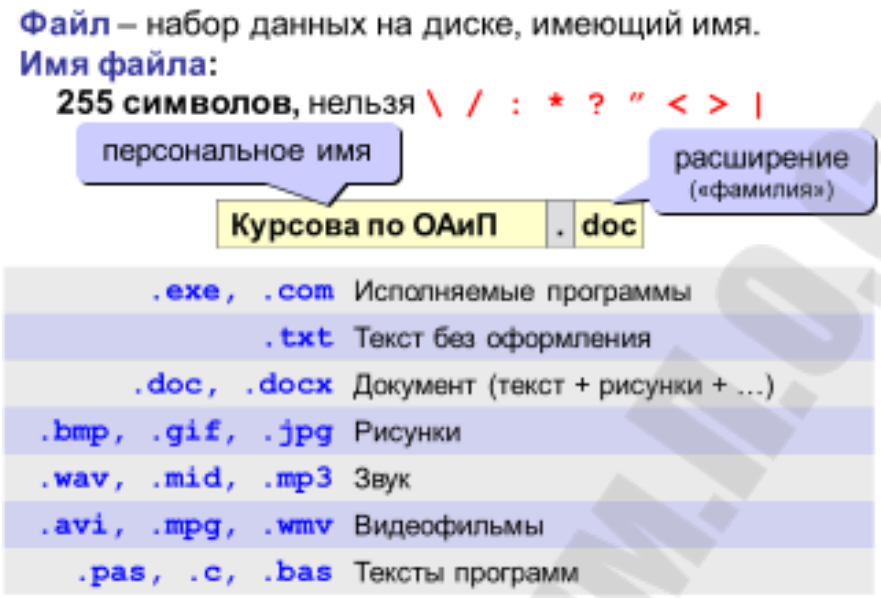
Общие сведения о файлах и файловой системе

Файлом называют именованную последовательность элементов данных (компонент файла), расположенных, как правило, во внешней памяти:

- на дискетах,
- винчестере,
- CD или других устройствах хранения информации,
- также устройствах ввода-вывода.

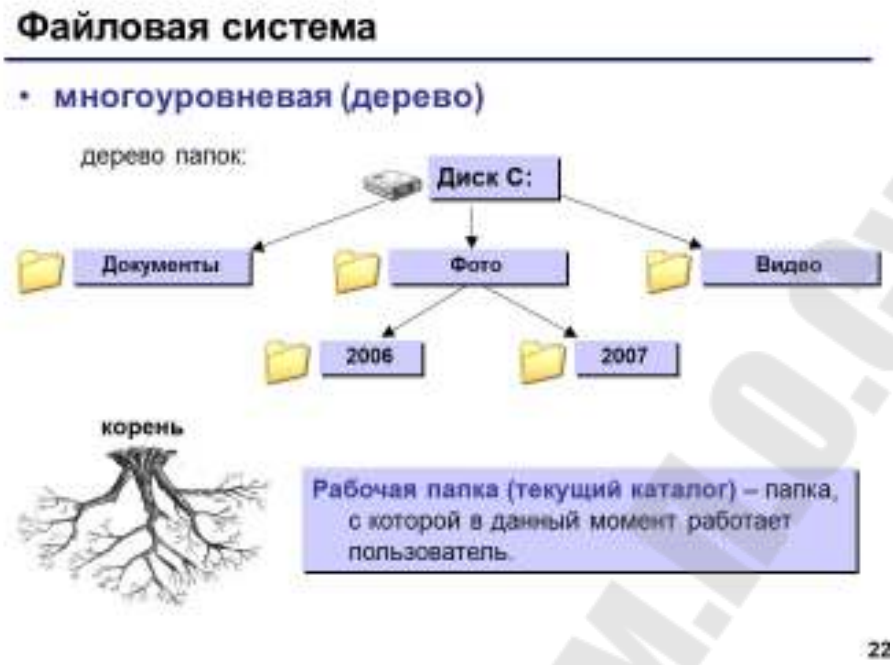
В файле может храниться текст, программа, числовые данные, графическое изображение и т.д.

Для организации работы с файлами программа на взаимодействует с операционной системой.



Для того чтобы операционная система (ОС) могла размещать файлы на дисках, последние должны быть специальным образом размечены (форматированы). Разметка осуществляется средствами используемой ОС.

Как правило, диски хранят большое количество файлов (количество их на жестких дисках обычно исчисляется тысячами). Для удобства и ускорения работы с таким количеством файлов применяется та или иная файловая система.



В Delphi файл определяется как последовательность компонентов, относящихся к одному типу:

файл записей,

файл целых чисел,

файл строк

и т. п.

Особенностью файлов по сравнению с другими структурными типами данных является то, что в любой момент доступен только один компонент.

Количество компонентов файла заранее не определяется.

Максимальный размер файла, размещенного во внешней памяти, ограничивается лишь техническими возможностями вычислительной системы.

Когда нужно использовать файлы

- Файлы полезны, если объем входных данных превосходит посильный при ручном вводе. (Крайним является случай, когда входные или выходные данные заведомо не могут поместиться в оперативной памяти.)
- Файлы нужны, если приходится многократно вводить одну и ту же информацию, с минимальными изменениями или вовсе без изменений (например, при отладке программы).
- Файлы необходимы, если нужно сохранять информацию о результатах работы программы, полученных при вводе различных входных данных (то есть: при поиске ошибок в программе).

Разновидности файлов

Бинарные файлы бывают двух видов

- типизированные,
- нетипизированные.

К **типизированным** относятся файлы, содержащие данные строго определенного типа. Обычно такие файлы представляются собой наборы записей.

К **нетипизированным** относятся двоичные файлы, которые могут содержать любые совокупности байтов данных без привязки к какому-нибудь одному типу.

Тема 3. Основные понятия БД. Работа с BDE

Архитектура и функции BDE

Мощность и гибкость Delphi при работе с базами данных основана на низкоуровневом ядре - процессоре баз данных Borland Database Engine (BDE).

BDE позволяет осуществлять доступ к данным как с использованием традиционного record-ориентированного (навигационного) подхода, так и с использованием set-ориентированного подхода, используемого в SQL-серверах баз данных.

Все инструментальные средства баз данных Borland - Paradox, dBase, Database Desktop - используют BDE. Все особенности, имеющиеся в Paradox или dBase, “наследуются” BDE, и поэтому этими же особенностями обладает и Delphi.

Алиасы

Таблицы сохраняются в базе данных. Некоторые СУБД сохраняют базу данных в виде нескольких отдельных файлов, представляющих собой таблицы (в основном, все локальные СУБД), в то время как другие состоят из одного файла, который содержит в себе все таблицы и индексы (InterBase).

Например, таблицы dBase и Paradox всегда сохраняются в отдельных файлах на диске. Каталог, содержащий dBase *.DBF* файлы или Paradox *.DB* файлы, рассматривается как база данных. Другими словами, любой каталог, содержащий файлы в формате Paradox или dBase, рассматривается Delphi как единая база данных. Для

переключения на другую базу данных нужно просто переключиться на другой каталог.

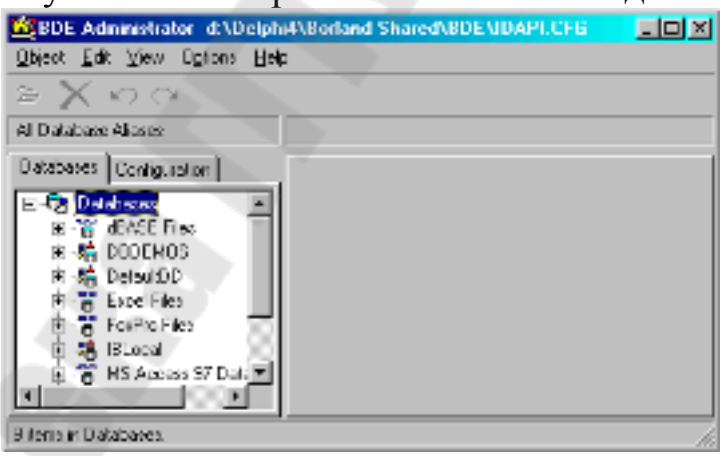
Удобно не просто указывать путь доступа к таблицам базы данных, а использовать для этого некий заменитель - псевдоним, называемый *алиасом*. Он сохраняется в отдельном конфигурационном файле в произвольном месте на диске и позволяет исключить из программы прямое указание пути доступа к базе данных.

Такой подход дает возможность располагать данные в любом месте, не перекомпилируя при этом программу.

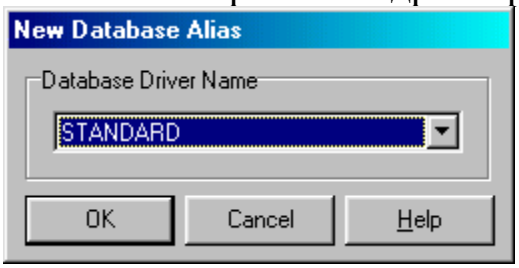
Кроме пути доступа, в алиасе указываются тип базы данных, языковый драйвер и много другой управляющей информации. Поэтому использование алиасов позволяет легко переходить от локальных баз данных к SQL-серверным базам (естественно, при выполнении требований разделения приложения на клиентскую и серверную части).

Для создания алиаса запустите утилиту конфигурации BDE (программу BDEADMIN.EXE), находящуюся в каталоге, в котором располагаются динамические библиотеки BDE.

Главное окно утилиты настройки BDE имеет вид



Для создания алиаса выберите в меню “Object” пункт “New”. В появившемся диалоговом окне выберите имя драйвера базы данных.



Тип алиаса может быть стандартным (STANDARD) для работы с локальными базами в формате dBase или Paradox или соответствовать наименованию SQL-сервера (InterBase, Sybase, Informix, Oracle и т.д.).

После создания нового алиаса следует дать ему имя. Это можно сделать с помощью подпункта “Rename” меню “Object”. Однако просто создать алиас не достаточно. Вам нужно указать дополнительную информацию, содержание которой зависит от типа выбранной базы данных.

Например, для баз данных Paradox и dBase (STANDARD) требуется указать лишь путь доступа к данным, имя драйвера и флаг ENABLE BCD, который определяет, транслирует ли BDE числа в двоично-десятичном формате (значения двоично-десятичного кода устраняют ошибки округления):

TYPE	STANDARD
DEFAULT DRIVER	PARADOX
ENABLE BCD	FALSE
PATH	c:\users\data

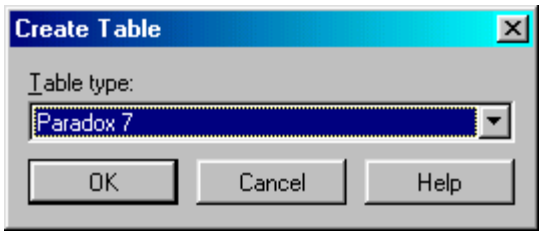
SQL-сервер InterBase и другие типы баз данных требуют задания большого количества параметров, многие из которых можно оставить установленными по умолчанию.

Тема 4. Работа с Data Base DeskTop

Утилита Database Desktop

Database Desktop - это утилита, во многом похожая на Paradox, которая поставляется вместе с Delphi для интерактивной работы с таблицами различных форматов локальных баз данных - Paradox и dBase, а также SQL-серверных баз данных InterBase, Oracle, Informix, Sybase (с использованием SQL Links). Исполняемый файл утилиты называется DBD32.EXE. Для запуска Database Desktop просто дважды щелкните по ее иконке.

После старта Database Desktop выберите команду меню File|New|Table для создания новой таблицы. Перед Вами появится диалоговое окно выбора типа таблицы.



Вы можете выбрать любой формат из предложенного, включая различные версии одного и того же формата.

После выбора типа таблицы Database Desktop представит Вам диалоговое окно, специфичное для каждого формата, в котором Вы сможете определить поля таблицы и их тип.

Следующий (после выбора имени поля) шаг состоит в задании типа поля. Типы полей очень сильно различаются друг от друга, в зависимости от формата таблицы. Для получения списка типов полей перейдите к столбцу “Type”, а затем нажмите пробел или щелкните правой кнопкой мышки.

Создание таблиц с помощью SQL

Database Desktop не обладает всеми возможностями по управлению SQL-серверными базами данных. Поэтому с помощью Database Desktop удобно создавать или локальные базы данных или только простейшие SQL-серверные базы данных, состоящие из небольшого числа таблиц, не очень сильно связанных друг с другом.

Если же необходимо создать базу данных, состоящую из большого числа таблиц, имеющих сложные взаимосвязи, можно воспользоваться языком SQL. При этом можно воспользоваться SQL Explorer в Delphi, каждый раз посылая по одному SQL-запросу, а можно записать всю последовательность SQL-предложений в один так называемый скрипт и послать его на выполнение.

Тема 5. Класс TDataSet

Компоненты Delphi для работы с БД

Имеются несколько основных компонент (объектов), которые Вы будете использовать постоянно для доступа к БД. Эти объекты могут быть разделены на три группы:

- невизуальные: TTable, TQuery, TDataSet, TField
- визуальные: TDBGrid, TDBEdit
- связующие: TDataSource

Первая группа включает невизуальные классы, которые используются для управления таблицами и запросами. Эта группа сосредотачивается вокруг компонент типа TTable, TQuery, TDataSet и TField. В Палитре Компонент эти объекты расположены на странице BDE(Delphi6-…).

Вторая важная группа классов - визуальные, которые показывают данные пользователю, и позволяют ему просматривать и модифицировать их. Эта группа классов включает компоненты типа TDBGrid, TDBEdit, TDBImage и TDBComboBox. В Палитре Компонент эти объекты расположены на странице Data Controls.

Имеется и третий тип, который используется для того, чтобы связать предыдущие два типа объектов. К третьему типу относится только невизуальный компонент TDataSource.

Класс TDataSet

TDataSet класс - один из наиболее важных объектов БД. Чтобы начать работать с ним, Вы должны взглянуть на следующую иерархию:

```
TDataSet
    |
  TDBDataSet
    |
    |-- TTable
    |-- TQuery
    |-- TStoredProc
```

TDataSet содержит абстрактные методы там, где должно быть непосредственное управление данными.

TDBDataSet знает, как обращаться с паролями и то, что нужно сделать, чтобы присоединить Вас к определенной таблице.

TTable знает (т.е. уже все абстрактные методы переписаны), как обращаться с таблицей, ее индексами и т.д.

Как Вы увидите в далее, TQuery имеет определенные методы для обработки SQL запросов.

TDataSet - инструмент, который Вы будете использовать чтобы открыть таблицу, и перемещаться по ней. Конечно, Вы никогда не будете непосредственно создавать объект типа TDataSet. Вместо этого, Вы будете использовать TTable, TQuery или других потомков TDataSet (например, TQBE).

На наиболее фундаментальном уровне, Dataset это просто набор записей.

В большинстве случаев dataset будет иметь прямое, один к одному, соответствие с физической таблицей, которая существует на диске. Однако, в других случаях Вы можете исполнять запрос или другое действие, возвращающие dataset, который содержит либо любое подмножество записей одной таблицы, либо объединение (join) между несколькими таблицами.

Открытие и закрытие DataSet

Если Вы используете TTable для доступа к таблице, то при открытии данной таблицы заполняются некоторые свойства TTable (количество записей RecordCount, описание структуры таблицы и т.д.).

Прежде всего, Вы должны поместить во время дизайна на форму объект TTable и указать, с какой таблицей хотите работать. Для этого нужно заполнить в Инспекторе объектов свойства DatabaseName и TableName. В DatabaseName можно либо указать

директорию, в которой лежат таблицы в формате dBase или Paradox (например, C:\DELPHI\DEMOS\DATA), либо выбрать из списка псевдоним базы данных (DBDEMOS).

Теперь, если свойство Active установить в True, то при запуске приложения таблица будет открываться автоматически.

Имеются два различных способа открыть таблицу во время выполнения программы. Вы можете написать следующую строку кода:

```
Table1.Open;
```

Или, если Вы предпочитаете, то можете установить свойство Active равное True:

```
Table1.Active := True;
```

Нет никакого различия между результатом производимым этими двумя операциями. Метод Open, однако, сам заканчивается установкой свойства Active в True, так что может быть даже чуть более эффективно использовать свойство Active напрямую.

Также, как имеются два способа открыть а таблицу, так и есть два способа закрыть ее. Самый простой способ просто вызывать Close:

```
Table1.Close;
```

Или, если Вы желаете, Вы можете написать:

```
Table1.Active := False;
```

Еще раз повторим, что нет никакой существенной разницы между двумя этими способами. Вы должны только помнить, что Open и Close это методы (процедуры), а Active - свойство.

Навигация (Перемещение по записям)

После открытия таблицы, следующим шагом Вы должны узнать как перемещаться по записям внутри него.

Следующий обширный набор методов и свойства TDataSet обеспечивает все, что Вам нужно для доступа к любой конкретной записи внутри таблицы:

```
procedure First;
procedure Last;
procedure Next;
procedure Prior;
property BOF: Boolean read FBOF;
property EOF: Boolean read FEOF;
procedure MoveBy(Distance: Integer);
```

Дадим краткий обзор их функциональных возможностей:

- Вызов Table1.First перемещает Вас к первой записи в таблице.
- Table1.Last перемещает Вас к последней записи.
- Table1.Next перемещает Вас на одну запись вперед.
- Table1.Prior перемещает Вас на одну запись Назад.
- Вы можете проверять свойства BOF или EOF, чтобы понять, находитесь ли Вы в начале или в конце таблицы.
- Процедура MoveBy перемещает Вас на N записей вперед или назад в таблице. Нет никакого функционального различия между запросом Table1.Next и вызовом Table1.MoveBy(1). Аналогично, вызов Table1.Prior имеет тот же самый результат, что и вызов Table1.MoveBy(-1).

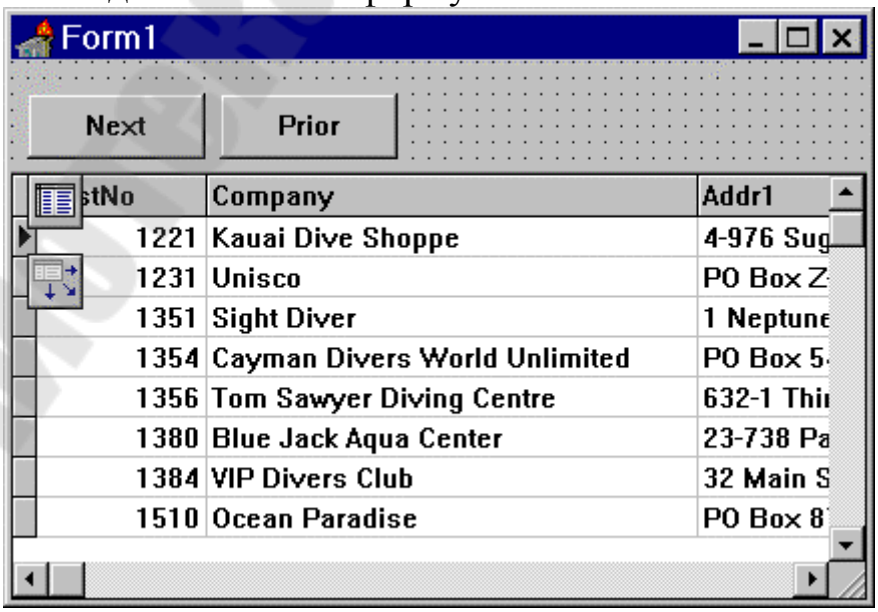
Чтобы начать использовать эти навигационные методы, Вы должны поместить TTable, TDataSource и TDBGrid на форму. Присоедините DBGrid1 к DataSource1, и DataSource1 к Table1. Затем установите свойства таблицы:

- в DatabaseName имя подкаталога, где находятся демонстрационные таблицы (или псевдоним DBDEMOS);
- в TableName установите имя таблицы CUSTOMER.

Если Вы запустили программу, которая содержит видимый элемент TDBGrid, то увидите, что можно перемещаться по записям таблицы с помощью полос прокрутки (scrollbar) на нижней и правой сторонах DBGrid.

Однако, иногда нужно перемещаться по таблице “программным путем”, без использования возможностей, встроенных в визуальные компоненты.

Поместите две кнопки на форму и назовите их Next и Prior.



Дважды щелкните на кнопке Next - появится заготовка обработчика события:

```
procedure TForm1.NextClick(Sender: TObject);
begin
end;
```

Теперь добавьте одну строчку кода так, чтобы процедура выглядела так:

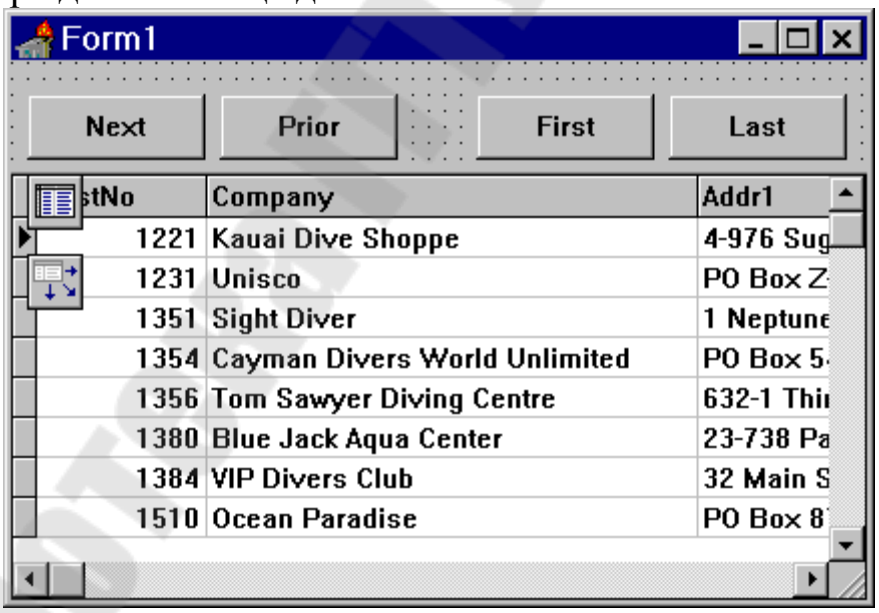
```
procedure TForm1.NextClick(Sender: TObject);
begin
  Table1.Next;
end;
```

Повторите те же самые действия с кнопкой Prior, так, чтобы функция связанная с ней выглядела так:

```
procedure TForm1.PriorClick(Sender: TObject);
begin
  Table1.Prior;
end;
```

Теперь запустите программу, и нажмите на кнопки. Вы увидите, что они легко позволяют Вам перемещаться по записям в таблице.

Теперь добавьте еще две кнопки и назовите их First и Last.



Сделайте то же самое для новых кнопок.

```
procedure TForm1.FirstClick(Sender: TObject);
begin
  Table1.First;
end;
procedure TForm1.LastClick(Sender: TObject);
```

```
begin
  Table1.Last;
end;
```

TDataSet.BOF - read-only Boolean свойство, используется для проверки, находитесь ли Вы в начале таблицы. Свойства BOF возвращает true в трех случаях:

- После того, как Вы открыли файл;
- После того, как Вы вызывали TDataSet.First;
- После того, как вызов TDataSet.Prior не выполняется.

Первые два пункта - очевидны. Когда Вы открываете таблицу, Delphi помещает Вас на первую запись; когда Вы вызываете метод First, Delphi также перемещает Вас в начало таблицы. Третий пункт, однако, требует небольшого пояснения: после того, как Вы вызывали метод Prior много раз, Вы могли добраться до начала таблицы, и следующий вызов Prior будет неудачным - после этого BOF и будет возвращать True.

Следующий код показывает самый общий пример использования Prior, когда Вы попадаете к началу файла:

```
while not Table.Bof do begin
  DoSomething;
  Table1.Prior;
end;
```

Все сказанное относительно BOF также применимо и к EOF. Другими словами, код, приведенный ниже показывает простой способ пробежать по всем записям в a dataset:

```
Table1.First;
while not Table1.EOF do begin
  DoSomething;
  Table1.Next;
end;
```

Классическая ошибка в случаях, подобных этому: Вы входите в цикл while или repeat, но забываете вызывать Table1.Next:

```
Table1.First;
repeat
  DoSomething;
until Table1.EOF;
```

Если Вы случайно написали такой код, то ваша машина зависнет. Также, этот код мог бы вызвать проблемы, если Вы открыли

пустую таблицу. Так как здесь используется цикл **repeat**, DoSomething был бы вызван один раз, даже если бы нечего было обрабатывать. Поэтому, лучше использовать цикл **while** вместо **repeat** в ситуациях подобных этой.

EOF возвращает True в следующих трех случаях:

- После того, как Вы открыли пустой файл;
- После того, как Вы вызывали TDataSet.Last;
- После того, как вызов TDataSet.Next не выполняется.

Единственная навигационная процедура, которая еще не упоминалась - MoveBy, которая позволяет Вам переместиться на N записей вперед или назад в таблице. Если Вы хотите переместиться на две записи вперед, то напишите:

MoveBy(2);

И если Вы хотите переместиться на две записи назад, то:

MoveBy(-2);

При использовании этой функции Вы должны всегда помнить, что DataSet - это изменяющиеся объекты, и запись, которая была пятой по счету в предыдущий момент, теперь может быть четвертой или шестой или вообще может быть удалена...

Prior и Next - это простые функции, которые вызывают MoveBy.

Тема 6. Работа с полями TFIelds

Значения полей

Получить доступ к значениям полей в Delphi очень просто. Компонент TDataSet по умолчанию предлагает массив свойств с именем FieldValues [ ], который возвращает значение определенного поля как значение типа Variant.

Поскольку массив FieldValues [ ] - это массив свойств по умолчанию, вам не нужно определять имя свойства для доступа к массиву.

Например, в следующем фрагменте кода значение поля CustName таблицы Tablel присваивается переменной S типа String:

```
S := Tablel['CustName'];
```

Так же просто можно присвоить значение поля CustNo целого типа переменной I:

```
I := Tablel['CustNo'];
```

Мощным следствием из сказанного выше является возможность сохранения значений нескольких полей в массиве типа Variant.

Единственным осложнением является то, что индекс массива типа Variant должен начинаться с нуля, а его содержимое представляет собой переменные var Variant.

```
const
    AStr = 'The %s is of the %s category and its length is %f in.';
var
    VaArr: Variant;
    F: Double;
begin
    VarArr := VarArrayCreate([0, 2], varVariant);
    { Предполагаем, что объект Tablel связан с таблицей Biolife }
    VarArr := Tablel['Common_Name;Category;Length_In'];
    F := VarArr[2];
    ShowMessage(Format(AStr, [VarArr[0], VarArr[l], F]));
 end;
```

Работа с массивом свойств FieldValues [ ] намного проще, чем со средствами доступа к значениям полей. По другой технологии для доступа к отдельному объекту TField, ассоциированному с набором данных, используется массив свойств Fields [ ] объекта TDataSet или вызывалась функция FieldsByName (). Компонент TField содержал информацию о конкретном поле.

Массив Fields [ ] - это массив (с индексами, начинающимися с нуля) объектов класса TField. Так, элемент Fields[0] возвращает объект TField, представляющий первое логическое поле записи.

Функции FieldsByName () передается строковый параметр, который соответствует некоторому имени поля в таблице. Таким образом, при вызове этой функции в виде FieldsByName ('OrderNo') она должна возвратить компонент TField, представляющий поле OrderNo в текущей записи набора данных.

Получив объект TField, можно ссчитать или присвоить значение полю с помощью одного из свойств этого объекта, приведенных в следующей таблице

Свойства объектов TFields

Свойство	Возвращаемый тип
AsDateTime	TDateTime
Value	Variant
AsBoolean	Boolean
AsFloat	Double
As Integer	Longint
AsString	String

Если первое поле в текущем наборе данных - строковое, сохранить его значение в переменной S типа String можно следующим образом:

S := Tablel.Fields[0].AsString;

В следующем фрагменте кода целой переменной I присваивается значение поля ' OrderNo' в текущей записи таблицы:

I := Tablel.FieldsByName('OrderNo').Aslnteger;

Типы полей данных

Если вы хотите узнать тип поля, используйте свойство DataType объекта TField, которое отображает тип данных таблицы базы данных (независимо от типа Object Pascal).

Свойство DataType имеет тип TFieldType, который определяется следующим образом:

type

TFieldType = (ftUnknown, ftString, ftSmallint, ftlnteger, ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob, ftVariant, ftlnterface, ftDispatch, ftGuid);

Существуют классы, производные от класса TField и предназначенные специально для работы с упоминаемыми выше типами данных.

Имена и номера полей

Для поиска имени определенного поля используйте свойство FieldName класса TField. Например, в следующем фрагменте кода имя первого поля в текущей таблице записывается в переменную S типа String:

```
var
    S: String;
begin
    S := Tablel.Fields[0].FieldName;
end;
```

Таким же образом, зная имя поля, можно получить его номер, используя свойство FieldNo. В следующем фрагменте кода номер поля OrderNo помещается в переменную I типа Integer:

```
var
    I: integer;
begin
    I := Tablel.FieldsByName('OrderNo1).FieldNo;
end;
```

Манипулирование полем данных

Ниже приведена последовательность действий, которые необходимо выполнить для редактирования одного или нескольких полей в текущей записи.

1. Вызовите метод Edit() набора данных для переключения в режим редактирования (Edit).

2. Назначьте новые значения требуемым полям.

3. Внесите изменения в набор данных посредством вызова метода Post() или перемещения на новую запись, в результате чего изменения будут внесены автоматически.

Например, типичная процедура изменения записи выглядит следующим образом:

```
TableI.Edit;
Table1[ 'Age'] := 23;
Tablel.Post;
```

Как и в случае редактирования, вставлять или добавлять записи в конец набора данных можно следующим образом.

1. Вызовите метод Insert () или Append () набора данных для переключения в режим вставки (Insert) или добавления (Append).

2. Присвойте значения полям новой записи набора данных.

3. Внесите новую запись в набор данных посредством вызова метода Post() или перемещения на новую запись, в результате чего изменения будут внесены автоматически.

Если потребуется отменить внесенные в запись, но еще не зафиксированные в наборе данных изменения, это можно сделать посредством вызова метода Cancel (). Например, следующий фрагмент кода отменяет выполненное редактирование до внесения изменений в таблицу:

```
TableI.Edit;
Table1[ 'Age'] := 23;
Table1.Cancel;
```

Метод Cancel () отменяет внесенные, но не зафиксированные изменения и возвращает набор данных из режима Edit, Append или Insert в режим Browse.

Завершим описание методов, манипулирующих записью компонента TDataSet, методом Delete(), который удаляет текущую запись из набора данных. Например, в следующем фрагменте кода удаляется последняя запись в таблице:

```
Tablel.Last; Table1.Delete;
```

Ограничения вводимых значений

Несколько возможностей ограничения предоставляют свойства полей, которые можно увидеть, сделав двойной щелчок на компоненте Table и выделив в окне Редактора Полей требуемое поле.

Для числовых полей имеются свойства. Value и MaxValue, устанавливающие допустимые пределы вводимых в поле значений. При нарушении этих пределов будет генерироваться исключение EDatabaseError, которое лучше перехватывать в приложении, чтобы выдать пользователю понятное пояснение на русском языке.

Другая возможность ограничения — использование свойств CustomConstraint и ConstraintErrorMessage.

Свойство CustomConstraint позволяет написать ограничение на значение поля в виде строки SQL

Если вы задали свойство CustomConstraint, то необходимо задать и своё. ConstraintErrorMessage. Оно содержит строку текста, который будет показано пользователю в случае, если он вводит данные, не удовлетворяющие поставлен ограничениям.

Еще одна возможность проверять данные на уровне поля — использовать обработку события поля OnValidate. Это событие возникает перед записью введенного значения поля в буфер текущей записи. Тут можно предусмотреть любые проверки, при появлении недопустимых данных выдать пользователю сообщение и пример, сгенерировать исключение EAbort функцией Abort.

Если все нормально, то после события OnValidate возникает еще событие OnChange, в обработчике которого тоже еще не поздно сгенерировать исключение.

Описанные выше способы проверки данных относятся к конкретному полю. Имеется также возможность осуществлять проверку на уровне записи, анализирую; различные ее поля.

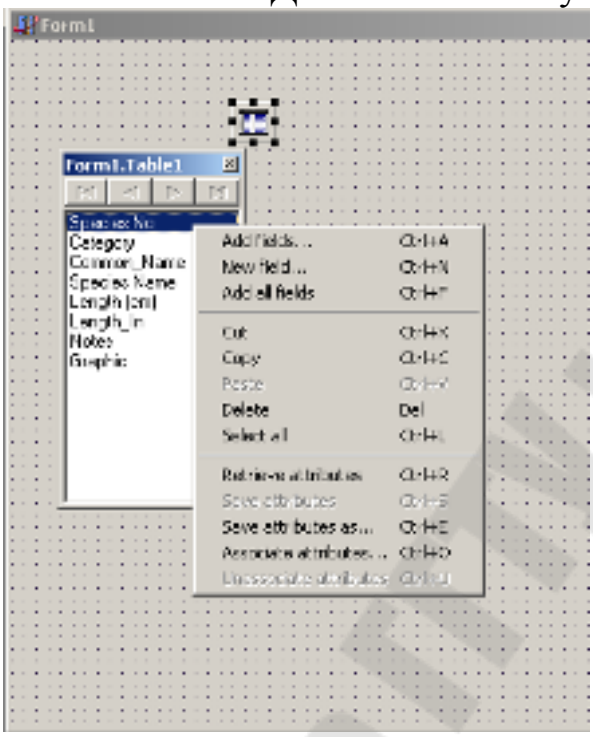
Для этого служит свойство Constraints компонента Table, нажмите кнопку в этом свойстве, и перед вами откроется окно. Щелкая в нем на кнопке Add, вы можете занести в свойство Constraints набор ограничений. Каждое из них представляет собой самостоятельный объект.

Выделив одно из ограничений, вы увидите в Инспекторе Объектов его свойства, свойство CustomConstraint представляет собой строку SQL, определяющую допустимые значения. Свойство ErrorMessage определяет строку текста, которая будет объявлена пользователю в случае нарушения ограничений.

РЕДАКТОР ПОЛЕЙ

В Delphi существует утилита Fields Editor (Редактор полей), предоставляющая возможность более гибкого управления полями набора данных. Этот инструмент можно использовать для просмотра отдельного набора данных в окне конструктора форм, для чего следует дважды щелкнуть на компоненте TTable, TQuery или TStoredProc либо выбрать команду Fields Editor в контекстном меню для набора данных.

В окне редактора полей можно выбрать поля набора данных, с которыми требуется работать, или же создать новые вычисляемые либо подстановочные поля. Для этого используйте контекстное меню редактора.



Для демонстрации методов использования редактора полей откройте новый проект и поместите в главную форму компонент TTable.

Установите свойство DatabaseName объекта Table1 равным DBDEMOS (это псевдоним, который заставляет Delphi использовать таблицы примеров), а свойство TableName равным ORDERS.DB.

Поместите в форму компоненты TDataSource и TDBGrid.

Свяжите объект DataSource1 с объектом Table1, а объект DBGrid1 — с объектом DataSource1. Теперь установите свойство Active объекта Table1 в True, и вы увидите данные объекта Table1.

Добавление полей

Откройте окно редактора полей дважды щелкнув на объекте Table1. Предположим, что требуется ограничить представление таблицы несколькими полями. В контекстном меню окна редактора полей выберите

команду Add Fields (Добавить поля). На экране раскроется диалоговое окно Add Fields. Выделите в списке Available fields поля OrderNo, CustNo и ItemsTotal, а затем щелкните на кнопке ОК. Эти три выделенных поля появятся в окне редактора полей и в сетке с данными.

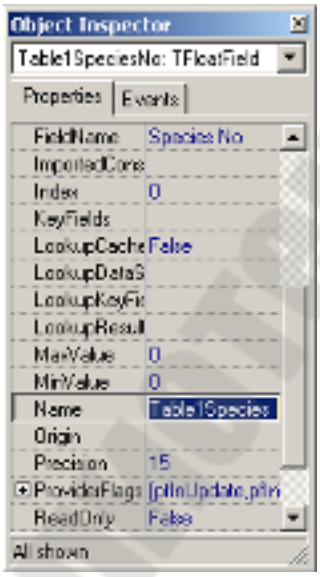
Для представления полей набора данных, выделенных в окне редактора полей, Delphi создает объекты, производные от объекта TField. Например, для выбранных только что трех полей таблицы ORDERS. DB Delphi помещает в исходный текст формы следующие объявления:

```
Table1OrderNo: TFloatField;
Table1CustNo: TFloatField;
Table1ItemsTotal: TCurrencyField;
```

Заметьте, что имя объекта поля — это соединение имени TTable и имени поля. Поскольку эти объекты полей создаются программно, можно получить доступ ко всем унаследованным ими от класса TField свойствам и методам непосредственно во время выполнения программы, а не только во время проектирования.

Поля и инспектор объектов

Если выделить поле в окне редактора полей, в окне инспектора объектов (Object Inspector) можно будет получить доступ к свойствам и событиям, ассоциированным с данным потомком объекта TField. Это позволяет модифицировать свойства полей (например, определять минимальное и максимальное значение, формат отображения, а также делать их доступными только для чтения). Назначение одних свойств (таких как Readonly — только чтение) очевидно из их названия, а назначение других может быть не совсем понятно.



Открыв в окне инспектора объектов вкладку Events, можно увидеть, что с объектами поля ассоциированы и некоторые события. События OnChange, OnGetText, OnSetText и OnValidate подробно описаны в интерактивной справочной системе.

Для получения справки по событию достаточно щелкнуть слева от его имени, а затем нажать клавишу <F1>.

Из всех событий чаще всего используется, пожалуй, OnChange. Оно позволяет выполнять некоторые действия при каждом изменении

содержимого поля (например, переходить на другую запись или добавлять новую запись).

Вычисляемые поля

Помимо прочего, в окне редактора полей к набору данных можно добавить вычисляемые поля. Например, допустим, что в набор данных необходимо добавить поле, отображающее для каждой строки в таблице ORDERS объем оптовой продажи, составляющий 32% от общего объема.

Выберите в контекстном меню окна редактора полей команду New Field. На экране раскроется диалоговое окно New Field. В поле Name этого окна введите имя нового поля— WholesaleTotal.

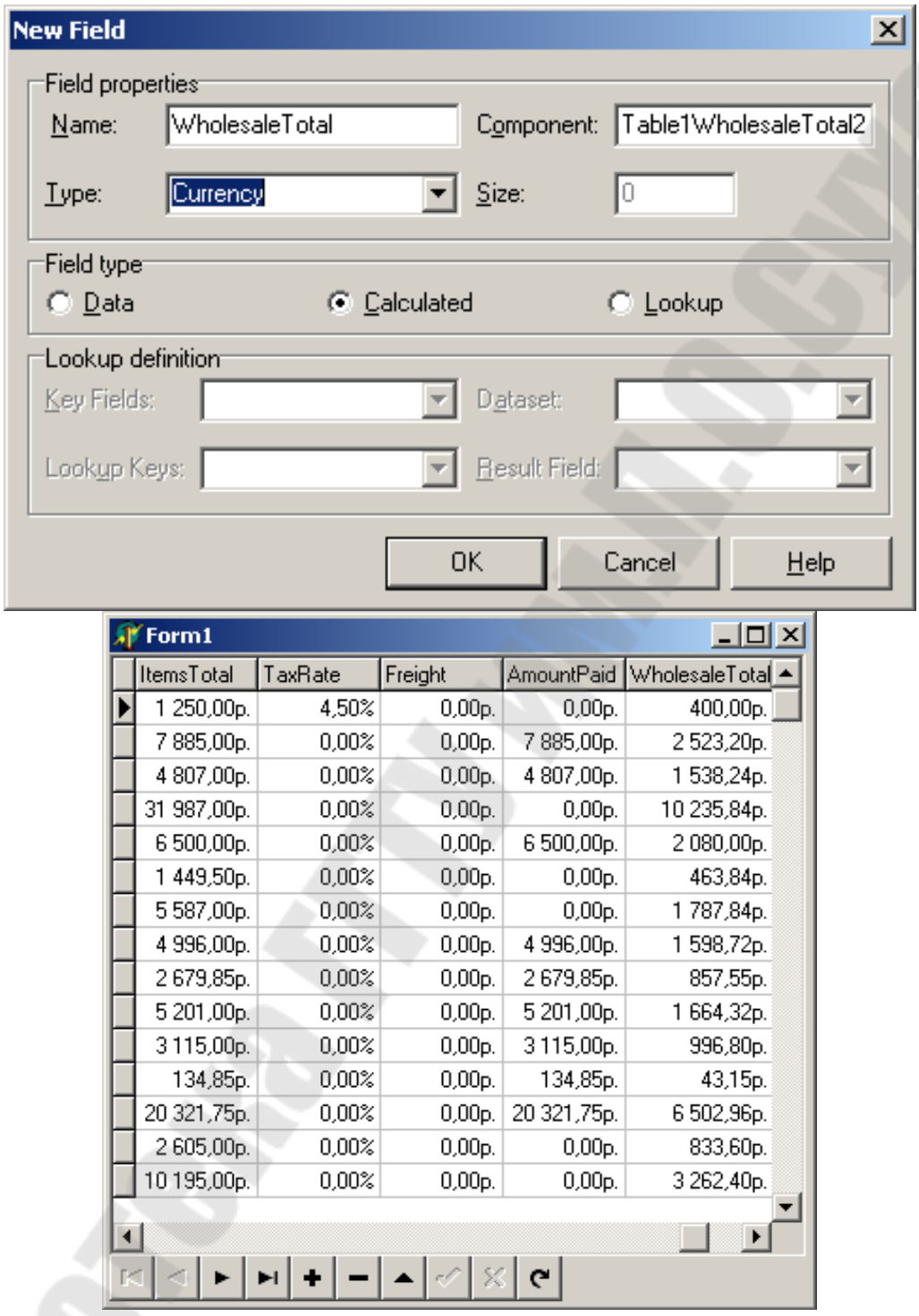
Тип этого поля— Currency, поэтому в раскрывающемся списке Type выберите именно это значение.

В группе Field Type установите переключатель в положение Calculated и щелкните на кнопке ОК. Новое поле появится в сетке, но пока не будет содержать никаких данных.

Чтобы заполнить новое поле данными, необходимо назначить требуемый метод событию OnCalcFields объекта Tablel. В тексте обработчика этого события полю WholesaleTotal следует просто присвоить значение, равное 32% от существующего значения поля SalesTotal.

Соответствующий текст метода обработки события Tablel.OnCalcFields показан ниже:

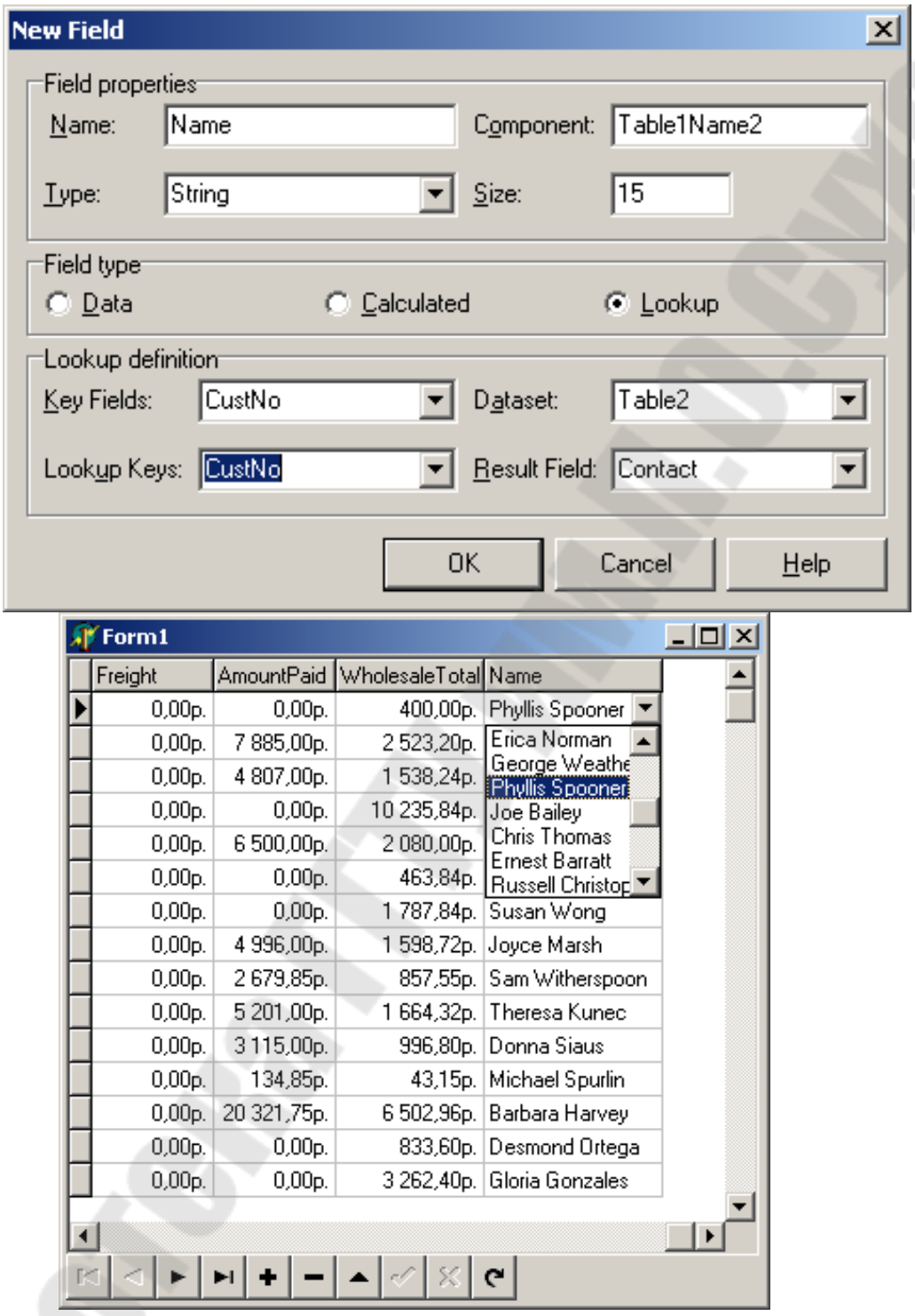
```
procedure TForml.TablelCalcFields(DataSet: TDataSet);
begin
    DataSet['WholesaleTotal']  := DataSet['ItemsTotal'] * 0.32;
end;
```



Подстановочные поля

Подстановочные (lookup) поля позволяют создавать в наборе данных такие поля, значения которых будут выбираться из другого набора данных.

Для иллюстрации сказанного добавим такое поле к текущему проекту. Вряд ли по номеру клиента (поле CustNo таблицы ORDERS) можно будет вспомнить его имя. Поэтому целесообразно добавить к таблице Tablel подстановочное поле, связанное с таблицей CUSTOMER, из которой по номеру клиента будет выбираться его имя.



Вначале поместите в форму второй объект TTable и присвойте его свойству DatabaseName значение DBDEMOS, а свойству TableName значение CUSTOMER.DB. Это будет объект Table2.

Затем вновь откройте окно New Field, для чего выберите команду New Field в контекстном меню окна редактора поля. Присвойте новому полю имя CustName и тип String.

Размер поля установите равным 15 символам. Не забудьте установить переключатель в группе Field Type в положение Lookup.

В списке Dataset этого диалогового окна выберите значение Table2 — именно этот набор данных необходимо просматривать.

В обоих списках Key Fields и Lookup Keys этого диалогового окна выберите значение CustNo — это то общее поле, по значению которого будет выполняться поиск.

И, наконец, в списке Result выберите значение Contact — именно это поле необходимо отображать в нашем наборе данных.

Обновление набора данных

При разработке приложения необходимо учитывать, что набор данных, с которым вы работаете, находится в постоянном движении. Его записи постоянно добавляются, удаляются или модифицируются, особенно при работе в локальной сети. Поэтому изредка необходимо перечитывать информацию о наборе данных с диска или из памяти для обновления содержимого набора данных, отображаемого в формах.

Обновить набор данных можно с помощью метода **Refresh()** компонента TDataSet.

Функционально он напоминает последовательный вызов для набора данных методов Close () и Open(), но выполняется несколько быстрее.

Метод Refresh () может успешно применяться ко всем локальным таблицам, однако при работе с базой данных на сервере SQL все же имеются некоторые ограничения.

Компонент TTable, подключенный к базе данных SQL, должен использовать уникальный индекс, и только в этом случае BDE может предпринять попытку выполнить операцию Refresh ().

Суть в том, что метод Refresh() всегда пытается сохранить текущую запись (если это возможно). Это означает, что для перемещения к текущей (на данный момент) записи BDE вынужден использовать метод Seek(). Однако это возможно лишь в том случае, если для набора данных SQL существует уникальный индекс.

Метод Refresh () нельзя использовать для компонентов TQuery, связанных с базой данных SQL.

Изменение состояния набора данных

Иногда бывает необходимо уточнить, в каком режиме находится таблица (Edit или Append) и активна ли она вообще. Получить эту информацию можно с помощью свойства State компонента TDataSet.

Свойство State имеет тип TDataSetState, а его значения приведены в таблице.

Значение	Описание
dsBrowse	Набор данных находится в режиме Browse (обычный режим просмотра данных)
dsCalcFields	Вызван обработчик события OnCalcFields, и значение полей записи в настоящий момент пересчитывается
dsEdit	Набор данных находится в режиме Edit (Редактирование). Это означает, что был вызван метод Edit( ) , но отредактированная запись еще не была внесена в таблицу
dsIinactive	Набор данных закрыт
dslnsert	Набор данных находится в режиме Insert (Вставка). Обычно это означает, что был вызван метод insert ( ), но вставляемая запись еще не была внесена в таблицу
dsSetKey	Набор данных находится в режиме SetKey (Задание ключа). Был вызван метод SetKey ( ) , но метод GotoKey ( ) еще не вызывался
dsNewValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству NewValue
dsOldValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству OldValue
dsCurValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству CurValue
dsFilter	В наборе данных в настоящее время выполняется фильтрация записей, выбор подстановочных значений или другая операция с использованием фильтра
dsBlockRead	Набор данных буферизируется, поэтому при установке этого значения перемещение курсора не приводит к обновлению данных в элементах управления и генерации событий
dslnternalCalc	В настоящее время вычисляется значение поля, у которого свойство FieldKind имеет значение f klnternalCalc
dsOpening	Набор данных находится в состоянии открытия, которое в настоящей момент еще не завершено. Это состояние устанавливается только тогда, когда набор данных открывается для асинхронной выборки данных

Использование TDataSource для проверки состояния БД:

TDataSource имеет три ключевых события, связанных с состоянием БД

OnDataChange

OnStateChange

OnUpdateData

OnDataChange происходит всякий раз, когда Вы переходите на новую запись, или состояние DataSet сменилось с dsInactive на другое, или начато редактирование. Другими словами, если Вы вызываете Next, Previous, Insert, или любой другой запрос, который должен привести к изменению данных, связанных с текущей записью, то произойдет событие OnDataChange. Если в программе нужно определить момент, когда происходит переход на другую запись, то это можно сделать в обработчике события OnDataChange:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:
TField);
begin
  if DataSource1.DataSet.State = dsBrowse then begin
    DoSomething;
  end;
end;
```

Событие *OnStateChange* событие происходит всякий раз, когда изменяется текущее состояние DataSet. DataSet всегда знает, в каком состоянии он находится. Если Вы вызываете Edit, Append или Insert, то TTable знает, что он теперь находится в режиме редактирования (dsEdit или dsInsert). Аналогично, после того, как Вы делаете Post, то TTable знает что данные больше не редактируется, и переключается обратно в режим просмотра (dsBrowse).

Dataset имеет шесть различных возможных состояний, каждое из которых включено в следующем перечисляемом типе:

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,
                 dsSetKey, dsCalcFields);
```

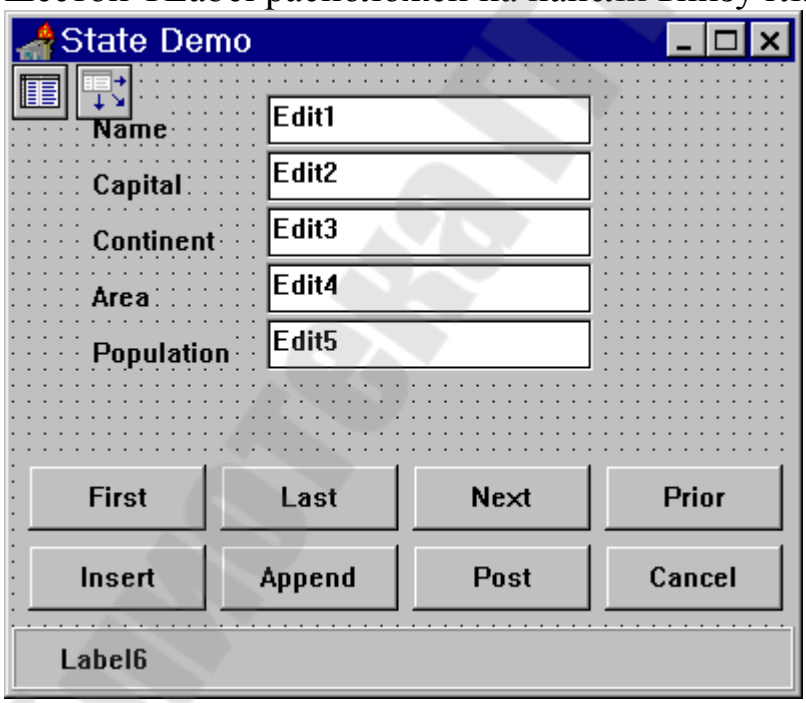
В течение обычного сеанса работы, БД часто меняет свое состояние между Browse, Edit, Insert и другими режимами. Если Вы хотите отслеживать эти изменения, то Вы можете реагировать на них написав примерно такой код:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  S: String;
```

```
begin
  case Table1.State of
    dsInactive: S := 'Inactive';
    dsBrowse: S := 'Browse';
    dsEdit: S := 'Edit';
    dsInsert: S := 'Insert';
    dsSetKey: S := 'SetKey';
    dsCalcFields: S := 'CalcFields';
  end;
  Label1.Caption := S;
end;
```

OnUpdateData событие происходит перед тем, как данные в текущей записи будут обновлены. Например, OnUpdateEvent будет происходить между вызовом Post и фактическим обновлением информации на диске.

События, генерируемые TDataSource могут быть очень полезны. Иллюстрацией этого служит следующий пример. Эта программа работает с таблицей COUNTRY, и включает TTable, TDataSource, пять TEdit, шесть TLlabel, восемь кнопок и панель. Действительное расположение элементов показано на рис.11. Обратите внимание, что шестой TLabel расположен на панели внизу главной формы.



Программа STATE показывает, как отслеживать текущее состояние таблицы.

Для всех кнопок напишите обработчики, вроде:

```
procedure TForm1.FirstClick(Sender: TObject);
begin
  Table1.First;
end;
```

В данной программе есть одна маленькая хитрость, которую Вы должны понять, если хотите узнать, как работает программа. Так как есть пять отдельных редакторов TEdit на главной форме, то хотелось бы иметь некоторый способ обращаться к ним быстро и легко. Один простой способ состоит в том, чтобы объявить массив редакторов:

```
Edits: array[1..5] of TEdit;
```

Чтобы заполнить массив, Вы можете в событии OnCreate главной формы написать:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to 5 do
    Edits[i] := TEdit(FindComponent('Edit' + IntToStr(i)));
  Table1.Open;
end;
```

Код показанный здесь предполагает, что первый редактор, который Вы будете использовать назовем Edit1, второй Edit2, и т.д. Существование этого массива позволяет очень просто использовать событие OnDataChange, чтобы синхронизировать содержание объектов TEdit с содержимом текущей записи в DataSet:

```
procedure TForm1.DataSource1DataChange(Sender: TObject;
                                       Field: TField);
var
  i: Integer;
begin
  for i := 1 to 5 do
    Edits[i].Text := Table1.Fields[i - 1].AsString;
end;
```

Всякий раз, когда вызывается Table1.Next, или любой другой из навигационных методов, то будет вызвана процедура показанная выше. Это обеспечивает то, что все редакторы всегда содержат данные из текущей записи.

Всякий раз, когда вызывается Post, нужно выполнить противоположное действие, то есть взять информацию из редакторов

и поместить ее в текущую запись. Выполнить это действие, проще всего в обработчике события TDataSource.OnUpdateData, которое происходит всякий раз, когда вызывается Post:

```
procedure TForm1.DataSource1UpdateData(Sender: TObject);
var
  i: Integer;
begin
  for i := 1 to 5 do
    Table1.Fields[i - 1].AsString := Edits[i].Text;
end;
```

Программа будет автоматически переключатся в режим редактирования каждый раз, когда Вы вводите что-либо в одном из редакторов. Это делается в обработчике события OnKeyDown (укажите этот обработчик ко всем редакторам):

```
procedure TForm1.Edit1KeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if DataSource1.State <> dsEdit then
    Table1.Edit;
end;
```

Этот код показывает, как Вы можете использовать св-во *State* DataSource, чтобы определить текущий режим DataSet.

Обновление метки в статусной панели происходит при изменении состояния таблицы:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  s : String;
begin
  case DataSource1.State of
    dsInactive : s:='Inactive';
    dsBrowse   : s:='Browse';
    dsEdit     : s:='Edit';
    dsInsert   : s:='Insert';
    dsSetKey   : s:='SetKey';
    dsCalcFields : s:='CalcFields';
  end;
  Label6.Caption:=s;
end;
```

Данная программа является демонстрационной и ту же задачу можно решить гораздо проще, если использовать объекты TDBEdit.

Отслеживание состояния DataSet

В предыдущей части Вы узнали, как использовать TDataSource, чтобы узнать текущее состоянии TDataSet. Использование DataSource - это простой путь выполнения данной задачи. Однако, если Вы хотите отслеживать эти события без использования DataSource, то можете написать свои обработчики событий TTable и TQuery:

property OnOpen
property OnClose
property BeforeInsert
property AfterInsert
property BeforeEdit
property AfterEdit
property BeforePost
property AfterPost
property OnCancel
property OnDelete
property OnNewRecord

Большинство этих свойств очевидны. Событие BeforePost функционально подобно событию TDataSource.OnUpdateData, которое объяснено выше. Другими словами, программа STATE работала бы точно также, если бы Вы отвечали не на DataSource1.OnUpdateData а на Table1.BeforePost. Конечно, в первом случае Вы должен иметь TDataSource на форме, в то время, как во втором этого не требуется.

Дополнительные возможномти работы с полями

В большинстве случаев, когда Вы хотите получить доступ из программы к индивидуальные полям записи, Вы можете использовать одно из следующих свойств или методов, каждый из которых принадлежат TDataSet:

```
property Fields[Index: Integer];
function FieldByName(const
                                    FieldName:
                                   string): TField;
property FieldCount;
```

Свойство *FieldCount* возвращает число полей в текущей структуре записи. Если Вы хотите программным путем прочитать имена полей, то используйте свойство *Fields* для доступа к ним:

```
var
  S: String;
begin
  S := Fields[0].FieldName;
end;
```

Если Вы работали с записью в которой первое поле называется CustNo, тогда код показанный выше поместит строку “CustNo” в переменную S. Если Вы хотите получить доступ к имени второго поля в вышеупомянутом примере, тогда Вы могли бы написать:

```
S := Fields[1].FieldName;
```

Таким образом, индекс передаваемый в *Fields* (начинающийся с нуля), и определяет номер поля к которому Вы получите доступ, т.е. первое поле - ноль, второе один, и так далее.

Если Вы хотите прочитать текущее содержание конкретного поля конкретной записи, то Вы можете использовать свойство *Fields* или метод *FieldsByName*.

Для того, чтобы найти значение первого поля записи, прочитайте первый элемент массива *Fields*:

```
  S := Fields[0].AsString;
```

Предположим, что первое поле в записи содержит номер заказчика, тогда код, показанный выше, возвратил бы строку типа “1021”, “1031” или “2058”.

Если Вы хотели получить доступ к этой переменной, как к числовой величине, тогда Вы могли бы использовать AsInteger вместо AsString. Аналогично, свойство Fields включают AsBoolean, AsFloat и AsDate.

Если хотите, Вы можете использовать функцию *FieldsByName* вместо свойства *Fields*:

```
  S := FieldsByName(‘CustNo’).AsString;
```

Тема 7. Фильтрация набора данных

Один из наиболее общих механизмов фильтрации, используемых в Delphi, — ограничение представления набора данных до некоторых указанных записей.

Этот процесс выполняется следующим образом.

1. Назначьте процедуру событию OnFilterRecord набора данных. В эту процедуру следует поместить операторы, выполняющие отбор записей на основе значений одного или нескольких полей.

2. Установите свойство Filtered набора данных равным True.

На первом этапе создадим обработчик события OnFilterRecord для этой таблицы. В данном случае будут отбираться только те записи, в которых значение поле Company начинается с прописной буквы S.

Текст этой процедуры выглядит следующим образом:

```
procedure TForml.TablelFilterRecord(DataSet: TDataSet; var Accept: Boolean);
var
    FieldVal: String;
begin
    FieldVal:= DataSet['Company']; // Получение значения поля
                                                    //  Company
    Accept:= FieldVal[l] = 'S'; // Принять запись,
                                                    // если поле начинается с буквы S
end;
```

После выполнения второго этапа (установки свойства Filtered таблицы равным True) таблица примет вид

В сетке отображаются только те записи, которые удовлетворяют критерию фильтрации.

Тема 8. Организация поиска. Работа с индексами

Методы FindFirst/FindNext

Компонент TDataSet содержит методы

FindFirst(),
FindNext(),
FindPrior()
и FindLast(),

которые используют фильтр для поиска записей, отвечающих его критерию поиска.

Все эти функции работают на не отфильтрованных наборах данных и предусматривают вызов обработчика события OnFilterRecord набора данных. На основе критерия поиска в обработчике события эти функции должны найти соответственно первую, следующую, предыдущую или последнюю запись. Каждая из этих функций не имеет параметров и возвращает значение Boolean, которое отображает, существует ли искомая запись.

Поиск записи

Фильтры эффективны не только при определении просматриваемого подмножества записей некоторого набора данных, они также могут использоваться для поиска записей в наборе данных по значению одного или нескольких полей.

Для этой цели класс TDataSet предоставляет метод Locate().

Обратите внимание на то, что функция Locate() выполняет поиск с помощью средств фильтрации и работает независимо от любых созданных для набора данных индексов.

Метод Locate () определяется следующим образом:

function Locate(const KeyFields: string;
const KeyValues: Variant;
Options: TLocateOptions): Boolean;

Первый параметр, KeyFields, содержит имя поля (или полей), по которому проводится поиск.

Второй параметр, KeyValues, содержит значение (или значения) поля, которое отыскивается.

Третий параметр, Options, позволяет указать требуемый тип поиска.

Этот параметр (Options) имеет тип TLocateOptions, который представляет собой множество, определяемое следующим образом:

type

TLocateOption = (loCaselnsensitive, loPartialKey);

TLocateOptions = set of TLocateOption;

Если множество содержит член loCaselnsensitive, поиск будет выполняться без учета регистра.

Если набор включает элемент loPartialKey, значения, содержащиеся в KeyValues, будут считаться удовлетворяющими критерию, даже если они являются подстрокой искомого значения.

Метод Locate() возвращает значение True, если искомая запись найдена.

Например, для поиска первого появления значения 1356 в поле CustNo набора данных Tablel можно использовать следующий оператор:

Tablel.Locate('CustNo', 1356, []);

Поиск записей

Для выполнения поиска записей в таблице библиотека VCL предлагает несколько методов. При работе с таблицами dBASE или Paradox, Delphi предполагает, что используемые для поиска поля индексированы.

Для SQL-таблиц быстродействие поиска будет уменьшаться при проведении поиска по неиндексированным полям.

В качестве примера будем использовать некоторую таблицу, проиндексированную по первому полю с числовым типом и по второму полю с текстовым типом. В этом случае поиск записей можно будет выполнять одним из двух способов: с помощью метода FindKey() или с помощью пары методов SetKey(). .GotoKey().

Метод FindKey ()

Метод FindKey () класса TTable позволяет искать запись, используя одно или несколько ключевых полей при одном вызове функции.

В качестве параметра методу FindKey () передается массив типа array of const, содержащий критерии поиска. Если поиск прошел успешно, метод возвращает значение True.

Следующий оператор вызывает перемещение в наборе данных к записи, где первое поле в индексе имеет значение 123, а второе содержит строку Hello:

```
if not Tablel.FindKey([123,  'Hello']) then
                                        MessageBeep(O);
```

Если поле не найдено, метод FindKey() возвращает значение False и компьютер издает звуковой сигнал.

Методы SetKey()..GotoKey()

Вызов метода SetKey() класса TTable переводит таблицу в режим, при котором ее поля подготавливаются к заполнению значениями, представляющими собой критерии поиска.

Как только критерий поиска будет установлен, можно будет вызвать метод GotoKey() для выполнения поиска требуемой записи в направлении сверху вниз.

Описанный выше пример с парой методов SetKey(). .GotoKey() можно записать следующим образом:

```
with Tablel do
begin
        SetKey;
        Fields[O].AsInteger := 123;
        Fields[I].AsString := 'Hello';
        If not GotoKey then MessageBeep(O);
end;
```

Поиск ближайшего соответствия

Подобным же образом метод FindNearest() или пару методов SetKey (). .GotoNearest() можно использовать для поиска в таблице значения, которое наиболее близко соответствует критерию поиска.

Для поиска первой записи, где значение первого индексированного поля наиболее близко (больше или равно) 123, используйте следующий код:

```
Tablel.FindNearest([123]);
```

И опять-таки, в качестве параметра методу FindNearest() передается массив типа array of const, содержащий значения полей, по которым требуется выполнить поиск.

Методы SetKey (). .GotoNearest() можно использовать для поиска следующим образом:

```
with Tablel do
 begin
        SetKey;
        Fields[OJ.AsInteger := 123;
        GotoNearest;
 end;
```

Если поиск завершится успешно и свойство **KeyExclusive** объекта таблицы имеет значение False, указатель текущей записи будет установлен на первую из записей соответствующих критерию поиска.

Если свойство KeyExclusive имеет значение True, текущей записью будет сделана запись, следующая за последней из всех записей, соответствующих условию поиска.

Использование индексов

Во всех описанных выше методах поиска подразумевается, что поиск проводится с использованием первичного индекса таблицы.

Если требуется выполнить поиск, используя вторичный индекс, необходимо поместить имя этого индекса в параметр IndexName объекта таблицы.

Например, если в таблице существует вторичный индекс по полю Company с именем ByCompany, то поиск записи по компании Unisco можно выполнить следующим образом:

```
with Tablel do
begin
        IndexName := 'ByCompany';
        SetKey;
        FieldValues['Company'] := 'Unisco';
        GotoKey;
end
```

Диапазоны (range) позволяют отфильтровывать таблицу так, что остаются только записи со значениями полей, попадающими внутрь определенных границ. Диапазоны обрабатываются подобно поиску по ключу, и, как и при поиске, существует несколько способов применения диапазона к выбранной таблице.

Можно использовать метод SetRange () или группу методов SetRangeStart(), SetRangeEnd() и ApplyRange().

Создание индексов

Правильно созданные и используемые локальные индексы могут существенно ускорить выполнение операций с набором данных. В то же время их невозможно сохранить вместе с набором данных локально, их необходимо перестраивать при каждом новом открытии набора данных и его обновлении с сервера.

Для создания локального индекса используется метод

```
procedure Addlndex(const Name, Fields: string;
                   Options: TIndexOptions;
                   const DescFields: string = '';
                   const CaselnsFields: string = '';
                   const GroupingLevel: Integer = 0);
```

Параметр Name определяет имя нового индекса. Параметр Fields должен содержать имена полей, которые разработчик хочет включить в индекс. Имена полей должны разделяться точкой с запятой. Параметр options позволяет задать тип индекса:

TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaselnsensitive, ixExpression, ixNonMaintained);

TIndexOptions = set of TIndexOption;

ixPrimary — первичный индекс;

ixUnique — значения индекса уникальны;

ixDescending — индекс сортирует записи в обратном порядке;

ixCaselnsensitive — индекс сортирует записи без учета регистра символов;

ixExpression — в индексе используется выражение (для индексов dBASE);

ixNonMaintained — индекс не обновляется при открытии таблицы.

При этом можно задать поля, порядок сортировки которых будет обратным. Для этого их необходимо перечислить через точку с запятой в параметре DescFields. Параметр CaselnsFields аналогичным образом позволяет задать поля, на сортировку которых не влияет регистр символов.

Параметры DescFields и CaselnsFields используются вместо параметра Options.

Параметр GroupingLevel задает уровень группировки полей индекса.

Ниже приведен код обработчика кнопки OnClick, с помощью которого строится индекс:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bActive, bExclusive: Boolean;
begin
  bActive := Table1.Active;
  bExclusive := Table1.Exclusive;
  Table1.IndexDefs.Update;

with Table1 do
  begin
    Close;
    {таблица dBASE должна быть открыта  в монопольном (exclusive) режиме}
    Exclusive := TRUE;
    Open;
    if Table1.IndexDefs.IndexOf('FNAME') <> 0 then
```

```
    Table1.AddIndex('FNAME', 'FNAME', []);
   Close;
   Exclusive := bExclusive;
   Active := bActive;
  end;
end;
```

Если вы собираетесь запускать проект из Delphi, убедитесь в том, что свойство таблицы Active в режиме проектирования установлено в False.

Метод SetRange ()

Как и метод FindKey() или FindNearest(), метод SetRange () позволяет выполнить достаточно сложные действия над таблицей с помощью единственного вызова этой функции.

В качестве параметров методу SetRange () передаются два массива типа array of const.

Первый представляет значения полей начала диапазона, а второй — значения полей конца диапазона.

В качестве примера ниже показан оператор, выполняющий фильтрацию записей, у которых значение первого поля больше или равно 10, но меньше или равно 15:

```
Tablel.SetRange([10], [15]);
```

Метод ApplyRange ()

Использование метода установки диапазона ApplyRange() предполагает выполнение следующих действий.

1. Вызовите метод SetRangeStart (), а затем модифицируйте массив свойств Fields [ ] таблицы, установив в нем начальное значение ключевого поля (или полей).
2. Вызовите метод SetRangeEnd() и вновь модифицируйте массив свойств Fields [ ], установив конечное значение ключевого поля (или полей).
3. Вызовите метод ApplyRange () для установки нового фильтра диапазона.

Продемонстрируем описанные действия на примере.

```
with Tablel do
begin
    SetRangeStart;
    Fields[0].Aslnteger := 10;      // Установка
                                    //начального значения
    SetRangeEnd;
    Fields[0].Aslnteger := 15;      // Установка
                                    // конечного значения
```

```
        ApplyRange;
    end;
```

Для удаления установленного методом ApplyRange () или SetRange () фильтра и приведения таблицы в исходное состояние следует вызвать метод CancelRange() объекта TTable.

Тема 9. Закладки. Связанные курсоры

Закладки (Bookmarks)

Часто бывает полезно отметить текущее местоположение в таблице так, чтобы можно было быстро возвратиться к этому месту в дальнейшем. Delphi обеспечивает эту функциональную возможность посредством трех методов, которые используют понятие *закладки*.

```
function GetBookmark: TBookmark;
(устанавливает закладку в таблице)
procedure GotoBookmark(Bookmark: TBookmark);
(переходит на закладку)
procedure FreeBookmark(Bookmark: TBookmark);
(освобождает память)
```

Как Вы можете видеть, вызов *GetBookmark* возвращает переменную типа TBookmark. TBookmark содержит достаточное количество информации, чтобы Delphi мог найти местоположение, к которому относится этот TBookmark. Поэтому Вы можете просто передавать этот TBookmark функции GotoBookmark, и будете немедленно возвращены к местоположению, связанному с этой закладкой.

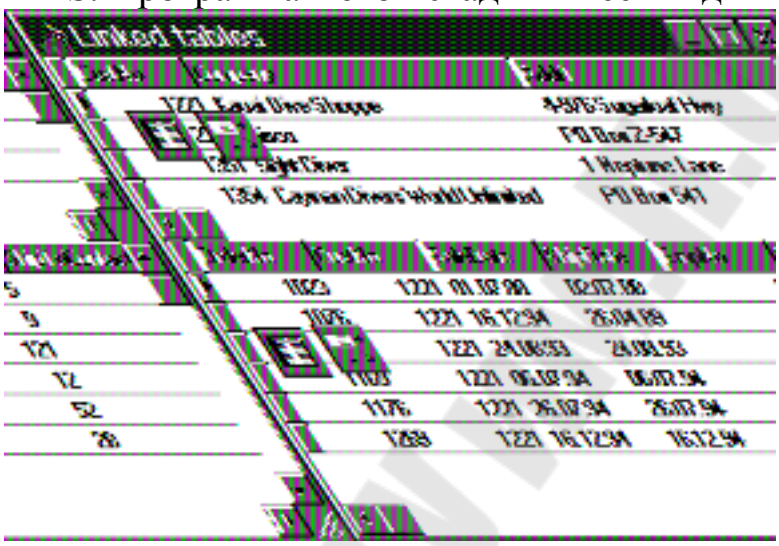
Обратите внимание, что вызов *GetBookmark* распределяет память для TBookmark, так что Вы должны вызывать *FreeBookmark* до окончания вашей программы, и перед каждой попыткой повторного использования Tbookmark (в GetBookMark).

Создание Связанных Курсоров (Linked cursors)

Связанные курсоры позволяют программистам определить отношение один ко многим (one-to-many relationship). Например, иногда полезно связать таблицы CUSTOMER и ORDERS так, чтобы каждый раз, когда пользователь выбирает имя заказчика, то он видит список заказов связанных с этим заказчиком. Иначе говоря, когда

пользователь выбирает запись о заказчике, то он может просматривать только заказы, сделанные этим заказчиком.

Программа LINKTBL демонстрирует, как создать программу которая использует связанные курсоры. Чтобы создать программу заново, поместите два TTable, два TDataSources и два TDBGrid на форму. Присоедините первый набор таблице CUSTOMER, а второй к таблице ORDERS. Программа в этой стадии имеет вид



Следующий шаг должен связать таблицу ORDERS с таблицей CUSTOMER так, чтобы Вы видели только те заказы, которые связанные с текущей записью в таблице заказчиков. В первой таблице заказчик однозначно идентифицируется своим номером - поле CustNo. Во второй таблице принадлежность заказа определяется также номером заказчика в поле CustNo. Следовательно, таблицы нужно связывать по полю CustNo в обоих таблицах (поля могут иметь различное название, но должны быть совместимы по типу).

Для этого, Вы должны сделать три шага, каждый из которых требует некоторого пояснения:

1. Установить свойство Table2.MasterSource = DataSource1
2. Установить свойство Table2.MasterField = CustNo
3. Установить свойство Table2.IndexName = CustNo

Если Вы теперь запустите программу, то увидите, что обе таблицы связаны вместе, и всякий раз, когда Вы перемещаетесь на новую запись в таблице CUSTOMER, Вы будете видеть только те записи в таблице ORDERS, которые принадлежат этому заказчику.

Свойство MasterSource в Table2 определяет DataSource, от которого Table2 может получить информацию. То есть, оно позволяет

таблице ORDERS знать, какая запись в настоящее время является текущей в таблице CUSTOMERS.

Но тогда возникает вопрос: Какая еще информация нужна Table2 для того, чтобы должным образом отфильтровать содержимое таблицы ORDERS? Ответ состоит из двух частей:

1. Требуется имя поля, по которому связанны две таблицы.
2. Требуется индекс по этому полю в таблице ORDERS (в таблице ‘многих записей’), которая будет связываться с таблицей CUSTOMER (таблице в которой выбирается ‘одна запись’).

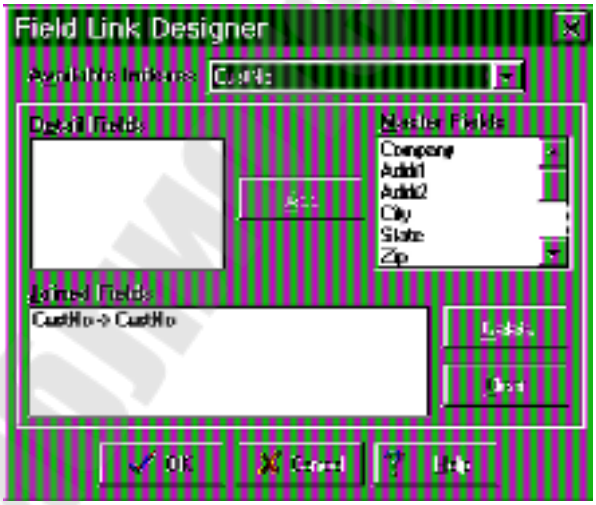
Чтобы правильно воспользоваться информацией описанной здесь, Вы должны сначала проверить, что таблица ORDERS имеет нужные индексы. Если этот индекс первичный, тогда не нужно дополнительно указывать его в поле IndexName, и поэтому Вы можете оставить это поле незаполненным в таблице TTable2 (ORDERS). Однако, если таблица связана с другой через вторичный индекс, то Вы должны явно определять этот индекс в поле IndexName связанной таблицы.

В примере, показанном здесь таблица ORDERS не имеет первичного индекса по полю CustNo, так что Вы должны явно задать в свойстве IndexName индекс CustNo.

Недостаточно, однако, просто упомянуть имя индекса, который Вы хотите использовать. Некоторые индексы могут содержать несколько полей, так что Вы должны явно задать имя поля, по которому Вы хотите связать две таблицы. Вы должны ввести имя ‘CustNo’ в свойство Table2.MasterFields.

Если Вы хотите связать две таблицы больше чем по одному полю, Вы должны внести в список все поля, помещая символ ‘|’ между каждым:

Table1.MasterFields := ‘CustNo | SaleData | ShipDate’;



В данном конкретном случае, выражение, показанное здесь, не имеет смысла, так как хотя поля SaleData и ShipDate также индексированы, но не дублируются в таблице CUSTOMER. Поэтому Вы должны ввести только поле CustNo в свойстве MasterFields. Вы можете определить это непосредственно в редакторе

свойств, или написать код подобно показанному выше. Кроме того, поле (или поля) связи можно получить, вызвав редактор связей - в Инспекторе Объектов дважды щелкните на свойство MasterFields.

Тема 10. TQuery

Основные понятия о TQuery

При использовании TTable, возможен доступ ко всему набору записей из одной таблицы. В отличие от TTable, TQuery позволяет произвольным образом (в рамках SQL) выбрать набор данных для работы с ним. Во многом, методика работы с объектом TQuery похожа на методику работы с TTable, однако есть свои особенности.

Вы может создать SQL запрос, используя компонент TQuery следующим способом:

1. Назначите Псевдоним (Alias) DatabaseName.
2. Используйте свойство SQL чтобы ввести SQL запрос типа
   “Select * from Country”.
3. Установите свойство Active в True

Если обращение идет к локальным данным, то вместо псевдонима можно указать полный путь к каталогу, где находятся таблицы.

Свойство SQL

Свойство SQL - вероятно, самая важная часть TQuery. Доступ к этому свойству происходит либо через Инспектор Объектов во время конструирования проекта (design time), или программно во время выполнения программы (run time).

Интересней, конечно, получить доступ к свойству SQL во время выполнения, чтобы динамически изменять запрос.

Например, если требуется выполнить три SQL запроса, то не надо размещать три компонента TQuery на форме. Вместо этого можно разместить один и просто изменять свойство SQL три раза.

Наиболее эффективный, простой и мощный способ - сделать это через параметризованные запросы.

Свойство SQL имеет тип TStrings, который означает что это ряд строк, сохраняемых в списке. Список действует также, как и массив, но, фактически, это специальный класс с собственными уникальными возможностями.

При программном использовании TQuery, рекомендуется сначала закрыть текущий запрос и очистить список строк в свойстве SQL:

Query1.Close;

Query1.SQL.Clear;

Обратите внимание, что всегда можно “безопасно” вызвать Close. Даже в том случае, если запрос уже закрыт, исключительная ситуация генерироваться не будет.

Следующий шаг - добавление новых строк в запрос:

Query1.SQL.Add(‘Select * from Country’);

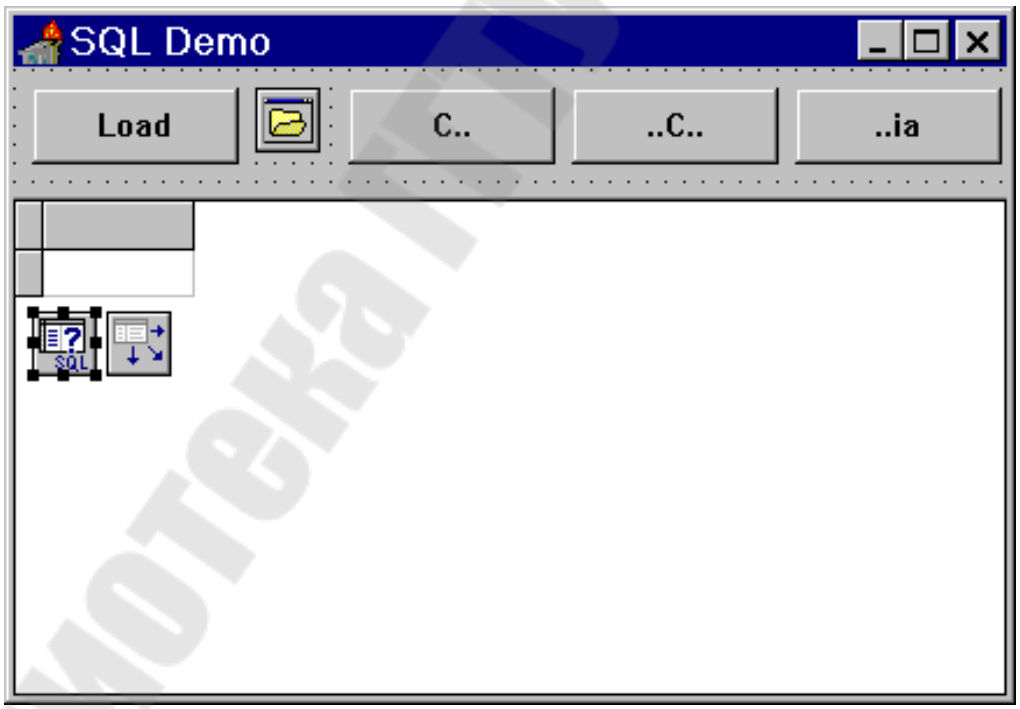
Query1.SQL.Add(‘where Name = ’’Argentina’’’);

Метод Add используется для добавления одной или нескольких строк к запросу SQL. Общий объем ограничен только количеством памяти на вашей машине.

Чтобы Delphi отработал запрос и возвратил курсор, содержащий результат в виде таблицы, можно вызвать метод:

Query1.Open;

Демонстрационная программа THREESQL показывает этот процесс.



Программа THREESQL использует особенность локального SQL, который позволяет использовать шаблоны поиска без учета регистра (case insensitive). Например, следующий SQL запрос:

Select * form Country where Name like ’C%’

возвращает DataSet, содержащий все записи, где поле Name начинается с буквы ‘C’.

Следующий запрос позволит увидеть все страны, в названии которых встречается буква ‘C’:

Select * from Country where Name like ‘%C%’;

Вот запрос, которое находит все страны, название которых заканчивается на ‘ia’:

Select * from Country where Name like ‘%ia’;

Одна из полезных особенностей свойства SQL - это способность читать файлы, содержащие текст запроса непосредственно с диска. Эта особенность показана в программе THREESQL.

Вот как это работает. В директории с примерами к данному уроку есть файл с расширением SQL. Он содержат текст SQL запроса. Программа THREESQL имеет кнопку с названием Load, которая позволяет Вам выбрать один из этих файлов и выполнять SQL запрос, сохраненный в этом файле.

Кнопка Load имеет следующий метод для события OnClick:

```
procedure TForm1.LoadClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with Query1 do begin
      Close;
          SQL.LoadFromFile(OpenDialog1.FileName);
      Open;
    end;
end;
```

Метод LoadClick сначала загружает компоненту OpenDialog и позволяет пользователю выбрать файл с расширением SQL. Если файл выбран, текущий запрос закрывается, выбраный файл загружается с диска в св-во SQL, запрос выполняется и результат показывается пользователю.

TQuery и Параметры

Delphi позволяет составить “гибкую” форму запроса, называемую параметризованным запросом. Такие запросы позволяют подставить значение переменной вместо отдельных слов в выражениях “where” или “insert”. Эта переменная может быть изменена практически в любое время.

Перед тем, как начать использовать параметризованные запросы, рассмотрим снова одно из простых вышеупомянутых предложений SQL:

Select * from Country where Name like ’C%’

Можно превратить это утверждение в параметризованный запрос заменив правую часть переменной NameStr:

select * from Country where Name like :NameStr

В этом предложении SQL, NameStr не является предопределенной константой и может изменяться либо во время дизайна, либо во время выполнения. SQL parser (программа, которая разбирает текст запроса) понимает, что он имеет дело с параметром, а не константой потому, что параметру предшествует двоеточие ":NameStr". Это двоеточие сообщает Delphi о необходимости заменить переменную NameStr некоторой величиной, которая будет известна позже.

Обратите внимание, слово NameStr было выбрано абсолютно случайно. Использовать можно любое допустимое имя переменной, точно также, как выбирается идентификатор переменной в программе.

Есть два пути присвоить значение переменной в параметризованном запросе SQL.

Один способ состоит в том, чтобы использовать свойство Params объекта TQuery.

Второй - использовать свойство DataSource для получения информации из другого DataSet. Вот ключевые свойства для достижения этих целей:

property Params[Index: Word];

function ParamByName(const Value: string);

property DataSource;

Если подставлять значение параметра в параметризованный запрос через свойство Params, то обычно нужно сделать четыре шага:

1. Закрыть TQuery
2. *Подготовить объект TQuery, вызвав метод Prepare*
3. Присвоить необходимые значения свойству Params
4. Открыть TQuery

Второй шаг выполняется в том случае, если данный текст запроса выполняется впервые, в дальнейшем его можно опустить.

Вот фрагмент кода, показывающий как это может быть выполнено практически:

```
Query1.Close;
Query1.Prepare;
Query1.Params[0].AsString := ‘Argentina’;
Query1.Open;
```

Этот код может показаться немного таинственным. Чтобы понять его, требуется внимательный построчный анализ. Проще всего начать с третьей строки, так как свойство Params является “сердцем” этого процесса.

Params - это индексированное свойство, которое имеет синтаксис как у свойства Fields для TDataSet. Например, можно получить доступ к первой переменной в SQL запросе, адресуя нулевой элемент в массиве Params:

```
Params[0].AsString := ‘”Argentina”’;
```

Если параметризованный SQL запрос выглядит так:

```
select * from Country where Name = :NameStr
```

то конечный результат (т.е. то, что выполнится на самом деле) - это следующее предложение SQL:

```
select * from Country where Name = “Argentina”
```

Все, что произошло, это переменной :NameStr было присвоено значение "Аргентина" через свойство Params. Таким образом, Вы закончили построение простого утверждения SQL.

Если в запросе содержится более одного параметра, то доступаться к ним можно изменяя индекс у свойства Params

```
Params[1].AsString := ‘SomeValue’;
```

либо используя доступ по имени параметра

```
ParamByName(‘NameStr’).AsString:=’”Argentina”’;
```

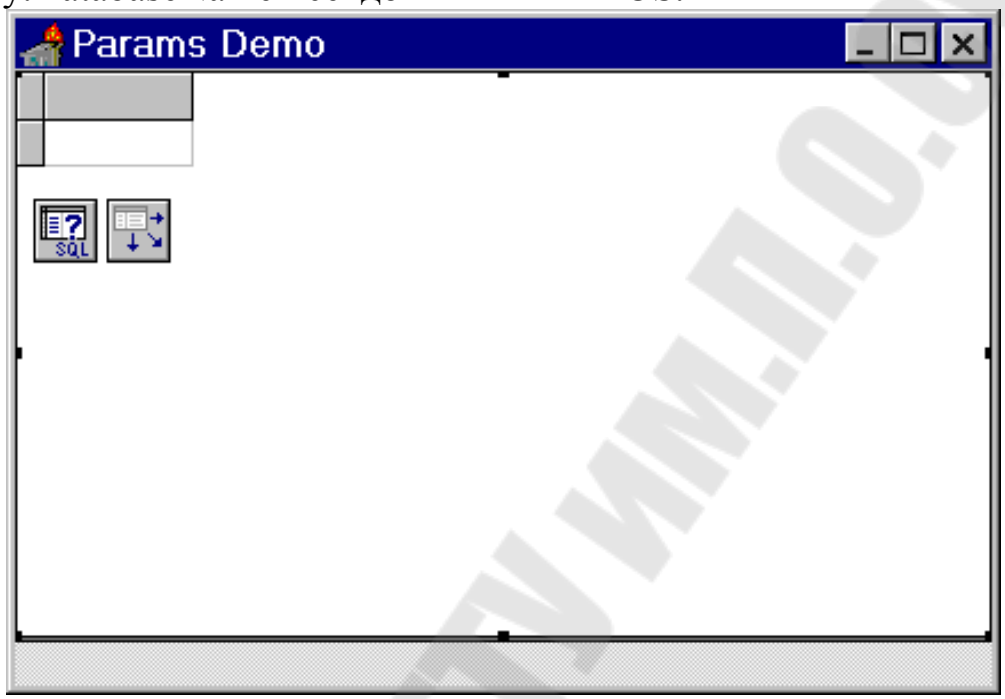
Итак, параметризованные SQL запросы используют переменные, которые всегда начинаются с двоеточия, определяя места, куда будут переданы значения параметров.

Прежде, чем использовать переменную Params, сначала можно вызвать Prepare. Этот вызов заставляет Delphi разобрать ваш SQL запрос и подготовить свойство Params так, чтобы оно "было готово принять” соответствующее количество переменных. Можно присвоить значение переменной Params без предварительного вызова Prepare, но это будет работать несколько медленнее.

После того, как Вы вызывали Prepare, и после того, как присвоили необходимые значения переменной Params, Вы должны вызвать Open, чтобы закончить привязку переменных и получить

желаемый DataSet. В нашем случае, DataSet должен включать записи, где в поле “Name” стоит “Argentina”.

Рассмотрим работу с параметрами на примере. Для создания программы, разместите на форме компоненты TQuery, TDataSource, TDBGrid и TTabSet. Соедините компоненты и установите в свойстве TQuery.DatabaseName псевдоним DBDEMOS.



Программа PARAMS во время дизайна.

В обработчике события для формы OnCreate напишем код, заполняющий закладки для TTabSet, кроме того, здесь подготавливается запрос:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i : Byte;
begin
  Query1.Prepare;
  for i:=0 to 25 do
    TabSet1.Tabs.Add(Chr(Byte('A')+i));
end;
```

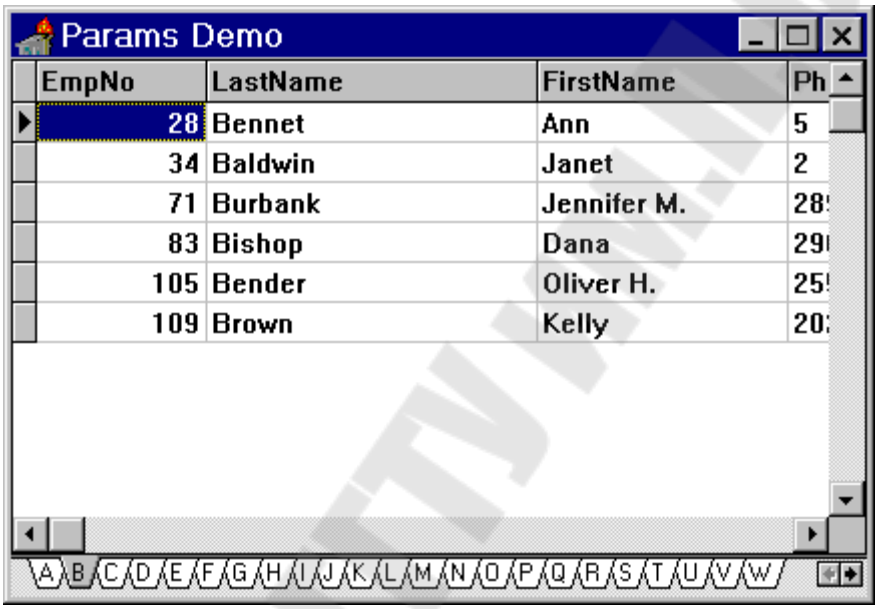
Текст SQL запроса в компоненте Query1:

```
select * from employee where LastName like :LastNameStr
```

Запрос выбирает записи из таблицы EMPLOYEE, в которых поле LastName похоже (like) на значение параметра :LastNameStr.

Параметр будет передаваться в момент переключения закладок:

```
procedure TForm1.TabSet1Change(Sender: TObject; NewTab: Integer; var
AllowChange: Boolean);
begin
  with Query1 do begin
    Close;
    Params[0].AsString:=
       ""+TabSet1.Tabs.Strings[NewTab]+'%"';
    Open;
  end;
end;
```



Программа PARAMS во время выполнения.
Передача параметров через TDataSource

Объект TQuery имеет свойство DataSource, которое может использоваться для того, чтобы создать связь с другим DataSet. Не имеет значения, является ли другой DataSet объектом TTable, TQuery, или некоторый другим потомком TDataSet. Все что нужно для установления соединения - это удостовериться, что у того DataSet есть связанный с ним DataSource.

Предположим, что Вы хотите создать связь между таблицами ORDERS и CUSTOMERS так, что каждый раз, когда Вы просматриваете конкретную запись о заказчике, будут видны только заказы, связанные с ним.

Рассмотрите следующий параметризованный запрос:

```
select * from Orders where CustNo = :CustNo
```

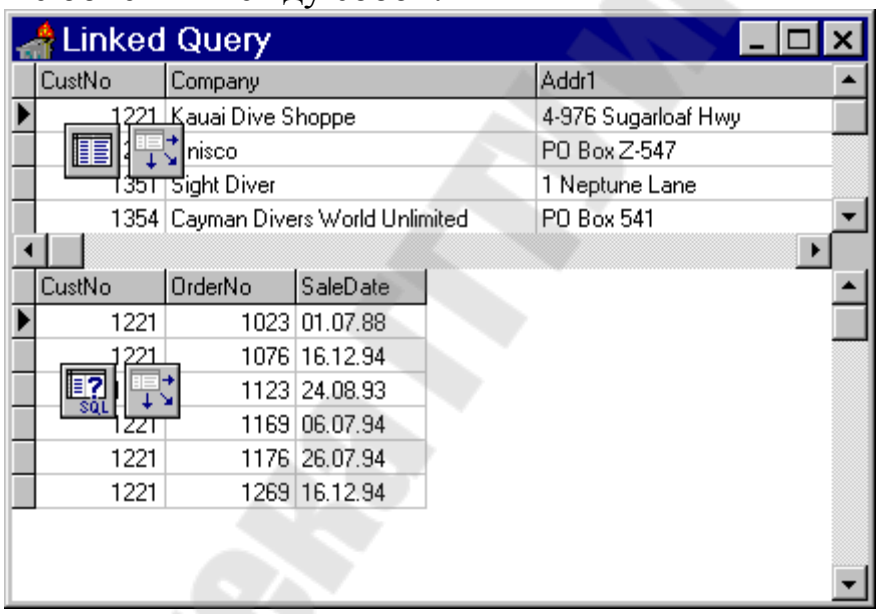
В этом запросе :CustNo - связывающая переменная, которой должно быть присвоено значение из некоторого источника. Delphi позволяет

использовать поле TQuery.DataSource чтобы указать другой DataSet, который предоставит эту информацию автоматически.

Другими словами, вместо того, чтобы использовать свойство Params и “вручную” присваивать значения переменной, эти значения переменной могут быть просто взяты автоматически из другой таблицы. Кроме того, Delphi всегда сначала пытается выполнить параметризованный запрос, используя свойство DataSource, и только потом (если не было найдено какое-то значение параметра) будет пытаться получить значение переменной из свойства Params.

При получении данных из DataSource считается, что после двоеточия стоит имя поля из DataSource. При изменении текущей записи в главном DataSet запрос будет автоматически пересчитываться.

Создайте новый проект, положите на форму один набор TTable, TDataSource и TDBGrid. Привяжите его к таблице CUSTOMER. Положите на форму второй набор - TQuery, TDataSource и TDBGrid и свяжите объекты между собой.



В свойстве SQL наберите текст запроса:

select * from Orders where CustNo = :CustNo

В свойстве DatabaseName для Query1 укажите DBDEMOS.

В свойстве DataSource для Query1 укажите DataSource1.

Поставьте Active = True и запустите программу.

Выполнение соединения нескольких таблиц.

Вы видели, что таблицы CUSTOMERS и ORDERS связаны в отношении один-ко-многим, основанному на поле CustNo. Таблицы ORDERS и ITEMS также связаны отношении один-ко-многим, только через поле OrderNo.

Более конкретно, каждый заказ, который существует в таблице ORDERS, будет иметь несколько записей в таблице ITEMS, связанных с этим заказом. Записи из таблицы ITEMS определяют тип и количество изделий, связанных с этим заказом.

Пример.

Некто Иванов Ф.П. 1 мая 1995г. заказал следующее:

1) Гайка 4х-угольная - 50 штук
2) Вентиль - 1 штука

А некто Сидорчук Ю.Г. 8 декабря 1994г. заказал:

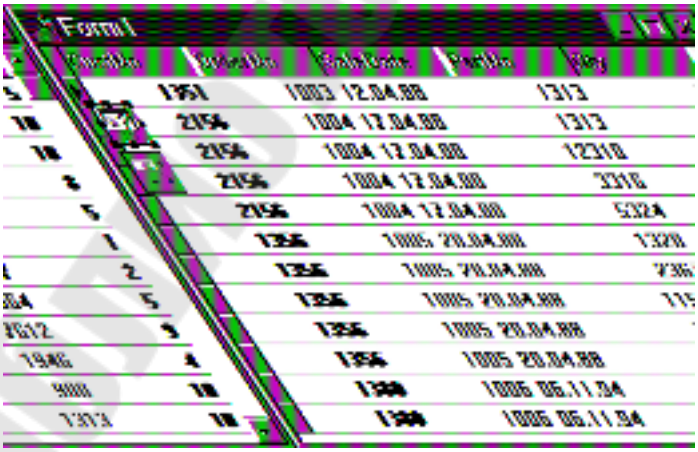
1) М/схема КР580 ИК80 - 10 штук
2) Транзистор КТ315 - 15 штук
3) Моток провода - 1 штука

В ситуации подобной этой, иногда проще всего "соединить" данные из таблиц ORDERS и ITEMS так, чтобы результирующий DataSet содержал информацию из обеих таблиц:

Иванов Ф.П.	1 мая 1995г	Гайка 4х-угольная	50 штук
Иванов Ф.П.	1 мая 1995г	Вентиль	1 штука
Сидорчук Ю.Г.	8 декабря 1994г	М/схема КР580 ИК80	10 штук
Сидорчук Ю.Г.	8 декабря 1994г	Транзистор КТ315	15 штук
Сидорчук Ю.Г.	8 декабря 1994г	Моток провода	1 штука

Слияние этих двух таблиц называется "соединение" и это одно из фундаментальных действий, которые Вы можете выполнить на наборе двух или больше таблиц.

Взяв таблицы ORDERS и ITEMS, их можно соединить их таким путем, что поля CustNo, OrderNo и SaleDate из таблицы ORDERS будут “слиты” с полями PartNo и Qty из таблицы ITEMS и сформируют новый DataSet, содержащий все пять полей. Grid, содержащий результирующий DataSet показан на рисунке.



Имеется существенное различие между связанными курсорами и соединенными таблицами. Однако они имеют две общие черты:

- И те, и другие используют две или более таблиц
- Каждый таблица связана

с другой по одному или более одинаковых полей.

Соединение таблиц ORDERS и ITEMS может быть выполнено единственным SQL запросом, который выглядит так:

```
select
  O.CustNo, O.OrderNo, O.SaleDate, I.PartNo, I.Qty
  from Orders O, Items I
  where O.OrderNo = I.OrderNo
```

Этот запрос состоит из четырех различных частей:

1. Выражение Select определяет, что Вы хотите получить - курсор, содержащий некоторую форму DataSet.
2. Затем идет список полей которые Вы хотите включить в dataset. Этот список включает поля CustNo, OrderNo, SaleDate, PartNo и Qty. Первые три поля из таблицы ORDERS, а два других - из таблицы ITEMS.
3. Выражение *from* объявляет, что Вы работаете с двумя таблицами, одна называется ORDERS, а другая ITEMS. Для краткости, в запросе используется особенность SQL, которая позволяет Вам ссылаться на таблицу ORDERS буквой O, а на таблицу ITEMS буквой I.
4. Выражение *where* жизненно важно потому, что оно определяет поля связи для двух таблиц. Некоторые серверы могут вернуть DataSet, даже если Вы не включите выражение *where* в запрос, но почти всегда результирующий набор записей будет не тем, что Вы хотели видеть. Чтобы получить нужный результат, убедитесь, что Вы включили выражение *where*.

Open или ExecSQL?

После того, как составлен SQL запрос, есть два различных способа выполнить его. Если Вы хотите получить курсор, то нужно вызывать Open. Если выражение SQL не подразумевает возвращение курсора, то нужно вызывать ExecSQL. Например, если происходит вставка, удаление или обновление данных (т.е. SQL запросы INSERT, DELETE, UPDATE), то нужно вызывать ExecSQL. Тоже самое можно сказать по-другому: Open вызывается при запросе типа SELECT, а ExecSQL - во всех остальных случаях.

Вот типичный SQL запрос, который используется для удаления записи из таблицы:

```
delete from Country where Name = ‘Argentina’;
```

Этот запрос удалил бы любую запись из таблицы COUNTRY, которая имеет значение "Argentina" в поле Имя.

Не трудно заметить, что это тот случай, когда удобно использовать параметризованный запрос.

Например, неплохо было бы менять имя страны, которую требуется удалить:

```
delete from Country where Name = :CountryName
```

В этом случае переменная :CountryName может быть изменена во время выполнения:

```
Query2.Prepare;
Query2.Params[0] := 'Argentina';
Query2.ExecSQL;
```

Код сначала вызывает Prepare, чтобы сообщить Delphi, что он должен разобрать SQL запрос и подготовить свойство Params. Следующим шагом присваивается значение свойству Params и затем выполняется подготовленный SQL запрос. Обратите внимание, что он выполняется через ExecSQL, а не Open.

Специальные свойства TQuery

Есть несколько свойств, принадлежащих TQuery, которые еще не упоминались:

```
property UniDirectional: Boolean;
property Handle: HDBICur;
property StmtHandle: HDBIStmt;
property DBHandle: HDBIDB;
```

Свойство UniDirectional используется для того, чтобы оптимизировать доступ к таблице. Если Вы установите UniDirectional в True, то Вы можете перемещаться по таблице более быстро, но Вы сможете двигаться только вперед.

Свойство StmtHandle связано со свойством Handle TDataSet. То есть, оно включено исключительно для того, что Вы могли делать вызовы Borland Database Engine напрямую.

При нормальных обстоятельствах, нет никакой необходимости использовать это свойство, так как компоненты Delphi могут удовлетворить потребностями большинства программистов. Однако, если Вы знакомы с Borland Database Engine, и если Вы знаете, что существуют некоторые возможностиб не поддерживаемые в VCL, то Вы можете использовать TQuery.StmtHandle, или TQuery. Handle, чтобы сделать вызов напрямую в engine.

Следующий фрагмент кода показывает два запроса к BDE:

```
var
  Name: array[0..100] of Char;
```

```
  Records: Integer;
begin
  dbiGetNetUserName(Name);
  dbiGetRecordCount(Query1.Handle, Records);
end;
```

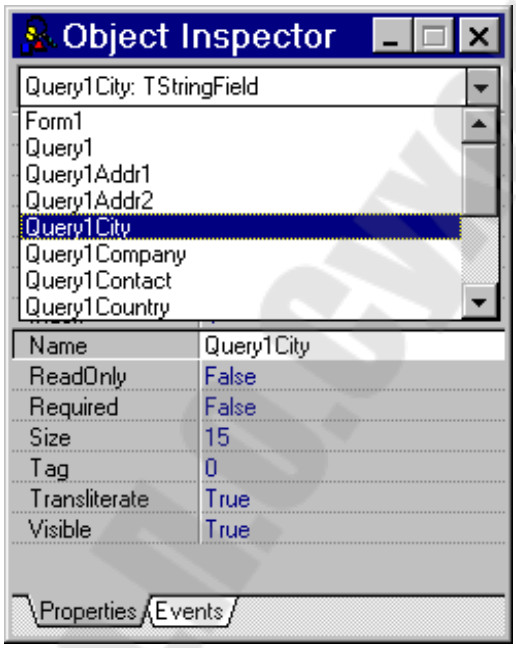
Редактор DataSet

Редактор DataSet может быть вызван с помощью объектов TTable или TQuery. Чтобы начать работать с ним, положите объект TQuery на форму, установите псевдоним DBDEMOS, введите SQL запрос "select * from customer" и активизируйте его (установив св-во Active в True).

Откройте комбобокс “Object Selector” вверху Инспектора Объектов - в настоящее время там имеется два компонента: TForm и TQuery.

Нажмите правую кнопку мыши на объекте TQuery и в контекстном меню выберите пункт “Fields Editor”. Нажмите кнопку Add - появиться диалог Add Fields, как показано на рисунке.

По-умолчанию, все поля в диалоге выбраны. Нажмите на кнопку OK, чтобы выбрать все поля, и закройте редактор. Снова загляните в “Object Selector”, теперь здесь появилось несколько новых объектов.



Эти новые объекты будут использоваться для визуального представления таблицы CUSTOMER пользователю.

Вот полный список объектов, которые только что созданы:
Query1CustNo: TFloatField;
Query1Company: TStringField;
Query1Addr1: TStringField;
Query1Addr2: TStringField;
Query1City: TStringField;
Query1State: TStringField;
Query1Zip: TStringField;
Query1Country: TStringField;
Query1Phone: TStringField;
Query1FAX: TStringField;
Query1TaxRate: TFloatField;
Query1Contact: TStringField;
Query1LastInvoiceDate: TDateTimeField;

Происхождение имен показанных здесь, должно быть достаточно очевидно. Часть "Query1" берется по-умолчанию от имени объекта TQuery, а вторая половина от имени поля в таблице Customer. Если бы мы сейчас переименовали объект Query1 в Customer, то получили бы такие имена:
CustomerCustNo
CustomerCompany

Это соглашение может быть очень полезно, когда Вы работаете с несколькими таблицами, и сразу хотите знать, на поле какой таблицы ссылается данная переменная.

Любой объект, созданный в редакторе DataSet, является наследником класса TField. Точный тип потомка зависит от типа данных в конкретном поле. Например, поле CustNo имеет тип TFloatField, а поле Query1City имеет тип TStringField. Это два типа полей, которые Вы будете встречать наиболее часто. Другие типы включают тип TDateTimeField, который представлен полем Query1LastInvoiceDate, и TIntegerField, который не встречается в этой таблице.

Чтобы понять, что можно делать с потомками TField, откройте Browser, выключите просмотр полей Private и Protected, и просмотрите свойства и методы Public и Published соответствующих классов.

Наиболее важное свойство называется Value. Вы можете получить доступ к нему так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  d: Double;
  S: string;
begin
  d := Query1CustNo.Value;
  S := Query1Company.Value;
  d:=d+1;
  S := 'Zoo';
  Query1CustNo.Value := d;
  Query1Company.Value := S;
end;
```

В коде, показанном здесь, сначала присваиваются значения переменным d и S. Следующие две строки изменяют эти значения, а последний две присваивают новые значения объектам. Не имеет большого смысла писать код, подобный этому, в программе, но этот код служит лишь для того, чтобы продемонстрировать синтаксис, используемый с потомками TField.

Свойство Value всегда соответствует типу поля, к которому оно относится. Например, у TStringFields - string, TCurrencyFields - double. Однако, если вы отображаете поле типа TCurrencyField с помощью компонент, “чувствительных к данным” (data-aware: TDBEdit, TDBGrid etc.), то оно будет представлена строкой типа: "$5.00".

Это могло бы заставить вас думать, что у Delphi внезапно отключился строгий контроль типов. Ведь TCurrencyField.Value объявлена как Double, и если Вы пробуете присвоить ему строку, Вы получите ошибку “type mismatch” (несоответствие типа).

Вышеупомянутый пример демонстрирует на самом деле свойства объектов визуализации данных, а не ослабление проверки типов.

Если нужно получить имена полей в текущем DataSet, то для этого используется свойство FieldName одним из двух способов, показанных ниже:

```
S := Query1.Fields[0].FieldName;
S := Query1CustNo.FieldName;
```

Если вы хотите получить имя объекта, связанного с полем, то вы должны использовать свойство Name:

```
S := Query1.Fields[0].Name;
S := Query1CustNo.Name;
```

Для таблицы CUSTOMER, первый пример вернет строку "CustNo", а любая из строк второго примера строку "Query1CustNo".

Вычисляемые Поля

Создание вычисляемых полей - одно из наиболее ценных свойств Редактора DataSet. Вы можете использовать эти поля для различных целей, но два случая выделяются особо:

- выполнение вычислений по двум или более полям в DataSet, и отображение результата вычислений в третьем поле.
- имитация соединения двух таблиц с возможностью редактировать результат соединения.

Создадим программу CALC_SUM, связывающую три таблицы в отношении один ко многим. В частности, ORDERS и ITEMS связаны по полю OrderNo, а ITEMS и PARTS связаны по полю PartNo. (В таблице ORDERS хранятся все заказы; в таблице ITEMS - предметы, указанные в заказах; PARTS - справочник предметов).

В программе можно перемещаться по таблице ORDERS и видеть связанный с текущим заказом список включенных в него предметов. Программа CALC_SUM достаточно сложная, но хорошо иллюстрирует мощность вычисляемых полей.

Последовательность создания проекта CALC_SUM:

- Создайте новый проект (File|New Project) и удалите из него форму (в Менеджере Проекта View|Project Manager)
- Выберите эксперта форм БД из меню Help.
- На первом экране, выберите "Create a master/detail form" и "Create a form using TQuery Objects".
- Нажмите кнопку Next и выберите таблицу ORDERS.DB из псевдонима БД DBDEMOS.
- Нажмите Next и выберите поля OrderNo, CustNo, SaleDate, ShipDate и ItemsTotal из таблицы ORDERS.DB.
- Нажмите Next и выберите "Horizontal" из расстановки компонентов dbEdit на форме.
- Нажмите Next и выберите таблицу ITEMS.DB.

- В двух следующих экранах выберите все поля из таблицы и поместите их в grid.
- Нажмите Next и выберите поле OrderNo из Master и Detail ListBoxes, и Нажмите кнопку Add.
- Нажмите Next и сгенерируйте форму.

Требуется много слов для того, чтобы описать процесс, показанный выше, но, фактически, выполнение команд в Эксперте форм БД легко и интуитивно.

Выделите первый из двух объектов TQuery и установят свойство Active в True. Для Query2 в свойстве SQL напишите текст запроса:

```
select * from Items I, Parts P
where (I.OrderNo =:OrderNo) and
(I.PartNo=P.PartNo)
```

Активизируйте объект Query2 (Active установите в True) и вызовите редактор DataSet (Fields Editor) для него. Вызовите диалог Add Fields и добавьте поля OrderNo, PartNo, Qty и ListPrice.

Нажмите Define и ведите слово Total в поле FieldName. Установите Field Type в CurrencyField. Проверьте, что Calculated CheckBox отмечен. Нажмите Ok и закройте редактор DataSet.

Простой процесс, описанный выше, показывает, как создать вычисляемое поле. Если посмотреть в DBGrid, то можно видеть, что там теперь есть еще одно пустое поле.

Для того, чтобы поместить значение в это поле, откройте в Инспекторе Объектов страницу событий для объекта Query2 и сделайте двойной щелчок на OnCalcFields. Заполните созданный метод так:

```
procedure TForm2.Query2CalcFields
                                    (DataSet: TDataSet);
begin
  Query2NewTotalInvoice.Value := 23.0;
end;
```

После запуска программы поле Total будет содержит строку $23.00.

Это показывает, насколько просто создать вычисляемое поле, которое показывает правильно сформатированные данные. На самом деле это поле должно показывать нечто другое - произведение полей Qty (количество) и ListPrice (цена).

Для этого вышеприведенный код для события OnCalcFields нужно изменить следующим образом:

```
procedure TForm1.Query2CalcFields
```

```
                                    (DataSet: TDataset);
begin
  Query2Total.Value:=Query2Qty.Value*
                                    Query2ListPrice.Value;
end;
```

Если теперь запустить программу, то поле Total будет содержать требуемое значение.

В обработчике события OnCalcFields можно выполнять и более сложные вычисления, однако следует помнить, что это вызывает соответствующее замедление скорости работы программы.

Теперь давайте добавим вычисляемое поле для первой таблицы (Query1, ORDERS), которое будет отображать сумму значений из поля Total второй таблицы (Query2) для данного заказа.

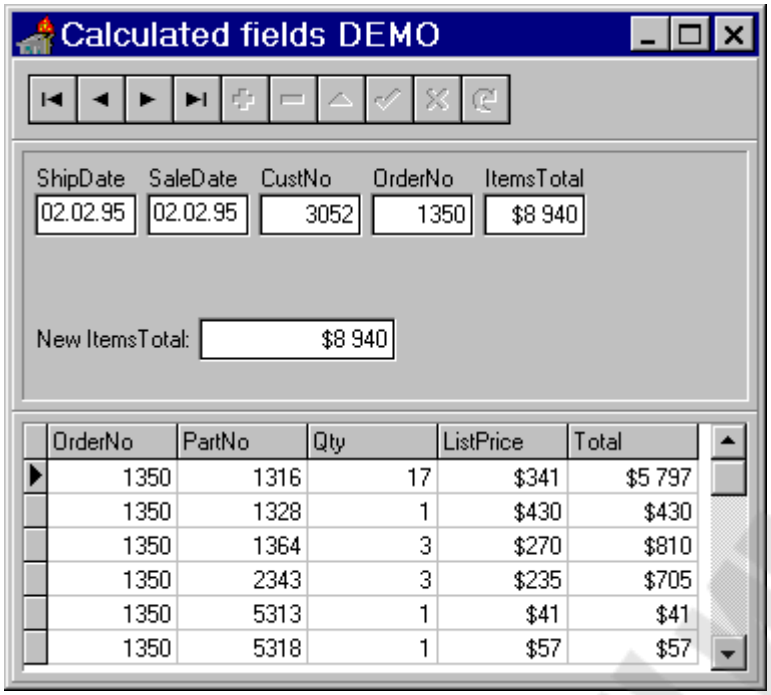
Вызовите редактор DataSet для объекта Query1 и добавьте вычисляемое поле NewItemsTotal типа CurrencyField. В обработчике события OnCalcFields для Query1 нужно подсчитать сумму и присвоить ее полю NewItemsTotal:

```
procedure TForm1.Query1CalcFields
              (DataSet: TDataset);
var
  R : Double;
begin
  R:=0;
  with Query2 do begin
    DisableControls;
    Close;
    Open;
    repeat
      R:=R+Query2Total.Value;
      Next;
    until EOF;
    First;
    EnableControls;
  end;
  Query1NewItemsTotal.Value:=R;
end;
```

В данном примере сумма подсчитывается с помощью простого перебора записей, это не самый оптимальный вариант - можно,

например, для подсчета суммы использовать дополнительный объект типа TQuery.

Метод *DisableControls* вызывается для того, чтобы отменить перерисовку DBGrid при сканировании таблицы.



Запрос Query2 переоткрывается для уверенности в том, что его текущий набор записей соответствует текущему заказу.

Поместите на форму еще один элемент DBEdit и привяжите его к Query1, полю NewItemsTotal. Запустите программу, ее примерный вид показан на рисунке.

Как видно из программы, наличие поля ItemsTotal в таблице ORDERS для данного примера необязательно и его можно было бы удалить (однако, оно необходимо в других случаях).

Тема 11. TDataSource

Класс TDataSource используется в качестве проводника между TTable или TQuery и компонентами, визуализирующими данные, типа TDBGrid, TDBEdit и TDBComboBox (data-aware components).

Компонент DataSource расположен на странице DataAccess. В большинстве случаев, все, что нужно сделать с DataSource - это указать в свойстве DataSet соответствующий TTable или TQuery. Затем, у data-aware (компонента визуализации) в свойстве DataSource указывается TDataSource, который используется в настоящее время.

Свойства

- **Enabled** Оно может быть полезно всякий раз, когда Вы хотите временно отсоединить, например, DBGrid от таблицы или

запроса.

Это требуется, например, если нужно программно пройти через все записи в таблице. Ведь, если таблица связана с визуальными компонентами, то каждый раз, когда Вы вызываете метод TTable.Next, визуальные компоненты будут перерисовываться. Даже если само сканирование в таблице двух или трех тысяч записей не займет много времени, то может потребоваться значительно больше времени, чтобы столько же раз перерисовать визуальные компоненты. В случаях подобных этому, лучше всего установить поле DataSource.Eabled в False. Это позволит Вам просканировать записи без перерисовки визуальных компонент. Это единственная операция может увеличить скорость в некоторых случаях на несколько тысяч процентов.

- AutoEdit указывает, переходит ли DataSet автоматически в режим редактирования при вводе текста в data-aware объекте.

События

DataSource имеет три ключевых события, связанных с состоянием БД: OnDataChange; OnStateChange; OnUpdateData

OnDataChange

происходит всякий раз, когда Вы переходите на новую запись, или состояние DataSet сменилось с dsInactive на другое, или начато редактирование.

Другими словами, если Вы вызываете Next, Previous, Insert, или любой другой запрос, который должен привести к изменению данных, связанных с текущей записью, то произойдет событие OnDataChange.

Если в программе нужно определить момент, когда происходит переход на другую запись, то это можно сделать в обработчике события OnDataChange.

Например, при переходе в DBGrid с одной записи на другую свойству Caption компоненты LabelMestor присваивается значение поля mestor (место рождения)

```
procedure TForm1.DataSourceOsndanDataChange
                        (Sender: TObject; Field: TField);
begin
    labelMestor.Caption:=
       QueryOsndan.fieldbyname('mestor').asstring;
end;
```

OnStateChange

Событие происходит всякий раз, когда изменяется текущее состояние DataSet. DataSet всегда знает, в каком состоянии он находится. Если Вы вызываете Edit, Append или Insert, то TTable знает, что он `теперь находится в режиме редактирования (dsEdit или dsInsert).

Аналогично, после того, как Вы делаете Post, то TTable знает, что данные больше не редактируется, и переключается обратно в режим просмотра (dsBrowse).

Dataset имеет шесть различных возможных состояний, каждое из которых включено в следующем перечисляемом типе:

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,
                                                dsSetKey, dsCalcFields);
```

В течение обычного сеанса работы, БД часто меняет свое состояние между Browse, Edit, Insert и другими режимами. Если Вы хотите отслеживать эти изменения, то Вы можете реагировать на них, написав примерно такой код:

```
procedure TForm1.DataSource1StateChange (Sender: TObject);
var  S: String;
begin
        case Table1.State of
                dsInactive: S := 'Inactive';
                dsBrowse: S := 'Browse';
                dsEdit: S := 'Edit';
                dsInsert: S := 'Insert';
                dsSetKey: S := 'SetKey';
                dsCalcFields: S := 'CalcFields';
        end;
        Label1.Caption := S;
end;
```

OnUpdateData

Событие происходит перед тем, как данные в текущей записи будут обновлены.

Тема 12. Использование визуальных компонент для работы с БД

DBGrid

Компонент DBGrid используется для просмотра и редактирования базы данных в режиме таблицы.

Свойства компонента:

Свойство	Описание
Name	Имя компонента
DataSource	Источник отображаемых в таблице данных [компонент DataSource]
Columns	Свойство Columns представляет собой массив объектов типа TColumn, каждый из которых определяет колонку таблицы и отображаемую в ней информацию
Options	Используется для "тонкой" настройки компонента
FieldName	Поле записи, содержимое которого выводится в колонке
Width	Ширина колонки в пикселях
Font	Шрифт, используемый для вывода текста в ячейках колонки
Color	Цвет фона колонки
Alignment	Способ выравнивания текста в ячейках колонки. Текст может быть выровнен по левому краю [taLeftJustify], по центру [taCenter] или по правому краю [taRightJustify]
Title.Caption	Заголовок колонки. По умолчанию является имя поля записи.
Title.Alingment	Способ выравнивания заголовка колонки. Заголовок может быть выровнен по левому краю [taLeftJustify],

	по центру [taCenter] или по правому краю [taRightJustify]
Title.Color	Цвет фона заголовка колонки
Title.Font	Шрифт заголовка колонки

Управление TDBGrid во время выполнения

Объект DBGrid может быть полностью реконфигурирован во время выполнения программы. Вы можете прятать и показывать колонки, изменять порядок показа колонок и их ширину.

Вы можете использовать свойство Options объекта DBGrid, чтобы изменить ее представление.

Свойство Options может принимать следующие возможные значения:

dgEditing	Установлен по-умолчанию в true, позволяет пользователю редактировать grid. Вы можете также установить свойство ReadOnly grid в True или False.
dgAlwaysShowEditor	Всегда показывать редактор.
dgTitles	Показывать названия колонок.
dgIndicator	Показывать небольшие иконки слева.
dgColumnResize	Может ли пользователь менять размер колонки.
dgColLines	Показывать линии между колонками.
dgRowLines	Показывать линии между строками.
dgTabs	Может ли пользователь использовать tab и shift-tab для переключения между колонками.
dgRowSelect	Выделять всю запись целиком.
dgAlwaysShowSelection	Всегда показывать выбранные записи.
dgConfirmDelete	Подтверждать удаление.
dgCancelOnExit	Отмена изменений при выходе из DBGrid.
dgMultiSelect	Одновременно может быть выделена больше чем одна запись.

Как объявлено в этой структуре:

```
DBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator,
    dgColumnResize, dgColLines, dgRowLines, dgTabs,
```

```
    dgRowSelect,dgAlwaysShowSelection, dgConfirmDelete,
    dgCancelOnExit, dgMultiSelect);
```

Например, Вы можете установить опции в Runtime, написав такой код:

```
DBGrid1.Options := [dgTitles, dgIndicator];
```

Если Вы хотите включать и выключать опции, это можно сделать с помощью логических операций. Например, следующий код будет добавлять dgTitles к текущему набору параметров:

```
DBGrid1.Options := DBGrid1.Options +
                                        [dgTitles];
-
```

Пусть есть переменная ShowTitles типа Boolean, тогда следующий код позволяют включать и выключать параметр одной кнопкой:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if ShowTitles then
    DBGrid1.Options := DBGrid1.Options +
                                          [dgTitles]
  else
    DBGrid1.Options := DBGrid1.Options –
                                          [dgTitles];
  ShowTitles := not ShowTitles;
end;
```

Если Вы хотите скрыть поле в run-time, то можете установить свойство visible в false:

```
Query1.FieldByName('CustNo').Visible := False;
Query1CustNo.Visible := False;
```

Обе строки кода выполняют идентичную задачу. Чтобы показать поле снова, установите видимый в true:

```
Query1.FieldByName('CustNo').Visible := True;
Query1CustNo.Visible := True;
```

Если Вы хотите изменить положение колонки в Runtime, можете просто изменить индекс, (первое поле в записи имеет индекс нуль):

```
Query1.FieldByName('CustNo').Index := 1;
Query1CustNo.Index := 2;
```

По-умолчанию, поле CustNo в таблице Customer является первым. Код в первой строке перемещает это поле во вторую позицию, а следующая строка перемещает его в третью позицию. Помните, что нумерация полей начинается с нуля, так присвоение

свойству Index 1 делает поле вторым в записи. Первое поле имеет Index 0.

Когда Вы изменяете индекс поля, индексы других полей в записи изменяются автоматически.

Если Вы хотите изменить ширину колонки в Runtime, только измените свойство DisplayWidth соответствующего TField.

Query1.FieldByName('CustNo').DisplayWidth := 12;

Query1CustNo.DisplayWidth := 12;

Величина 12 относится к числу символов, которые могут быть показаны в видимом элементе.

DBNavigator

Компонент DBNavigator обеспечивает перемещение указателя текущей записи, активизацию режима редактирования, добавление и удаление записей

Кнопки компонента DBNavigator:

Кнопка	Обозначение	Действие
К первой	nbFirst	Указатель текущей записи перемещается к первой записи файла данных
К предыдущей	nbPrior	Указатель текущей записи перемещается к предыдущей записи файла данных
К следующей	nbNext	Указатель текущей записи перемещается к следующей записи файла данных
К последней	nbLast	Указатель текущей записи перемещается к последней записи файла данных
Добавить	nbInsert	В файл данных добавляется новая

		запись
Удалить	nbDelete	Удаляется текущая запись файла данных
Редактирование	nbEdit	Устанавливает режим редактирования текущей записи
Сохранить	nbPost	Изменения, внесенные в текущую запись, записываются в файл данных
Отменить	nbCancel	Отменяет внесенные в текущую запись изменения
Обновить	nbRefresh	Записывает внесенные изменения в файл

Свойства компонента:

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента
DataSource	Имя компонента, являющегося источником данных. В качестве источника данных может выступать база данных [компонент Database], таблица [компонент Table] или результат выполнения запроса [компонент Query]
VisibleButtons	Видимые командные кнопки

Другие компоненты визуализации

Свойства компонентов:

Свойство	Описание
Name	Имя компонента. Используется для доступа к

	свойствам компонента
DataSource	Компонент-источник данных
DataField	Поле базы данных, для отображения или редактирования которого используется компонент

Компонент **DBText** – аналог обычной метки Label, но связанный с данными. Он позволяет отображать данные некоторого поля, но не дает возможности его редактировать. Тип отображаемого поля может быть различным: строка, число, булева величина, дата. Компонент автоматически переводит соответствующие типы в отображаемые символы.

DBEdit – связанный с данными аналог обычного окна редактирования Edit. Он позволяет отображать и редактировать данные полей различных типов: строка, число, булева величина. Впрочем, если задать в компоненте ReadOnly=true, то он, как и DBText, будет служить элементом отображения, но несколько более изящным, чем DBText.

DBMemo- связанный с данными аналог обычного многострочного окна редактирования Memo. Он позволяет отображать и редактировать данные поля типа Memo, а также данные любых типов, указанных выше для предыдущих компонентов, например в этом компоненте можно отображать и редактировать текст приказа на слушателя, характеристику сотрудника и т.п.

DBRichEdit –связанный с данными аналог обычного многострочного окна редактирования текста в обогащенном формате RTF. Область применения та же, что и для компонента DBMemo.

DBImage – связанный с данными аналог обычного компонента Image. Компонент позволяет отображать графические поля, например, фотографии слушателей.

DBChecBox – связанный с данными аналог обычного индикатора ChecBox. Он позволяет редактировать и отображать данные поля булева типа. Если при выводе данных поле имеет значение true, то индикатор включается. И наоборот, при редактировании поля присваемое ему значение определяется состоянием индикатора.

DBRadioGroup – связанный с данными аналог группы радиокнопок RadioGroup. Компонент позволяет отображать и редактировать поля с ограниченным множеством возможных

значений

Характерной особенностью всех этих компонентов, отличающих их от несвязанных с данными аналогичных компонентов, является отсутствие в окне Инспектора Объектов их основных свойств, отображающих содержание: Caption в DBText, Text в DBEdit, Picture в DBImage и т.п. Все эти свойства имеются в компонентах, но они доступны только во время выполнения.

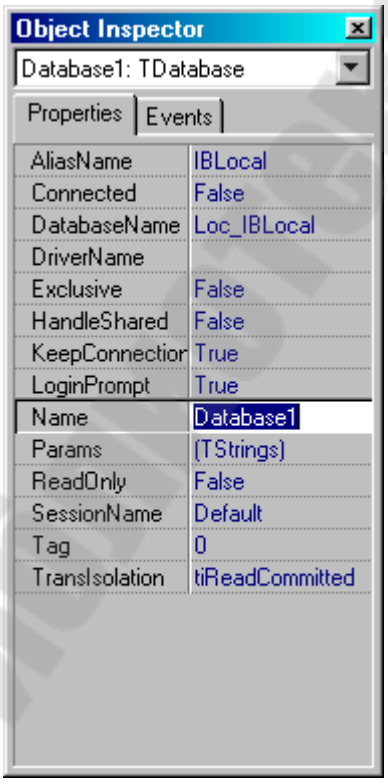
DBLookupComboBox- используется для отображения поля в виде выпадающего списка. Это особенно актуально при нехватке места. Здесь для связи с компонентой DataSource используется свойство ListSourse, а не свойство DataSource, которое используется для связязанных таблиц. Свойство ListField связывается с полем, которое будет отображаться в списке. Свойство KeyField позволит выбрать соответствующее каждой записи ключевое поле.

DBLookupListBox- используется для отображения поля в виде списка. Связь с полями осуществляется аналогично предыдущей компоненте.

DBChart – предназначен для построения диаграмм.

Тема 13. Класс TDataBase. Работа с Session. Транзакции

Класс TDataBase



Объект типа TDataBase не является обязательным при работе с базами данных, однако он предоставляет ряд дополнительных возможностей по управлению соединением с базой данных.

TDataBase служит для:

- Создания постоянного соединения с базой данных
- Определения собственного диалога при соединении с базой данных (опрос пароля)
- Создания локального псевдонима базы данных

- Изменения параметров при соединении
- Управления транзакциями

TDataBase является невидимым во время выполнения объектом. Он находится на странице “Data Access” Палитры Компонент. Для включения в проект TDataBase нужно “положить” его на главное окно вашей программы.

Создание постоянного соединения с базой данных

Если вы работаете с базой данных, то перед началом работы выполняется процедура соединения с этой базой. В процедуру соединения, кроме прочего, входит опрос имени и пароля пользователя (кроме случая работы с локальными таблицами Paradox и dBase через IDAPI).

Если в программе не используется TDataBase, то процедура соединения выполняется при открытии первой таблицы из базы данных.

Соединение с базой данных обрывается, когда в программе закрывается последняя таблицы из этой базы.

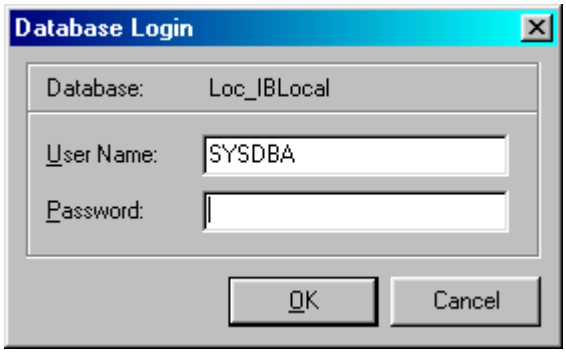
Теперь, если снова открыть таблицу, то процедура установки соединения повторится и это может быть достаточно неудобно для пользователя.

Чтобы соединение не обрывалось даже в том случае, когда нет открытых таблиц данной базы, можно использовать компонент типа TDataBase.

В свойстве AliasName укажите псевдоним базы данных, с которой работает программа; в свойстве DatabaseName - любое имя (псевдоним БД), на которое будут ссылаться таблицы вместо старого псевдонима базы. Свойство Connected установите в True - процедура соединения с базой будет выполняться при запуске программы. И, наконец, свойство KeepConnection нужно установить в True.

Определение собственного диалога при соединении с базой данных

По умолчанию при соединении с базой данных используется диалог опроса имени и пароля пользователя, показанный на рисунке



При желании можно изменить внешний вид диалога или вообще его отменить. Для этого используются свойства и события класса TDataBase - *LoginPrompt, Params* и *OnLogin.*

Чтобы отключить опрос имени и пароля установите свойство LoginPrompt в False. При этом в свойстве Params требуется в явном виде (во время дизайна либо во время выполнения) указать имя и пароль пользователя.

Например, в программе можно написать (до момента соединения с базой, например в событии для Form1 OnCreate):

```
DataBase1.LoginPrompt:=False;
DataBase1.Params.Clear;
DataBase1.Params.Add('USER NAME=SYSDBA');
DataBase1.Params.Add('PASSWORD=masterkey');
DataBase1.Connected:=True;
```

Чтобы использовать свой собственный диалог, в котором можно опрашивать не только имя и пароль пользователя, но и, например, сетевой протокол - создайте обработчик события OnLogin для DataBase1:

```
Procedure TForm1.Database1Login(
            Database: TDatabase;
            LoginParams: TStrings);
begin
  Form2.ShowModal;
  if Form2.ModalResult = mrOK then
    with LoginParams do begin
      Values['USER NAME'] := User_Name;
      Values['PASSWORD'] := User_Pass;
    end;
end;
```

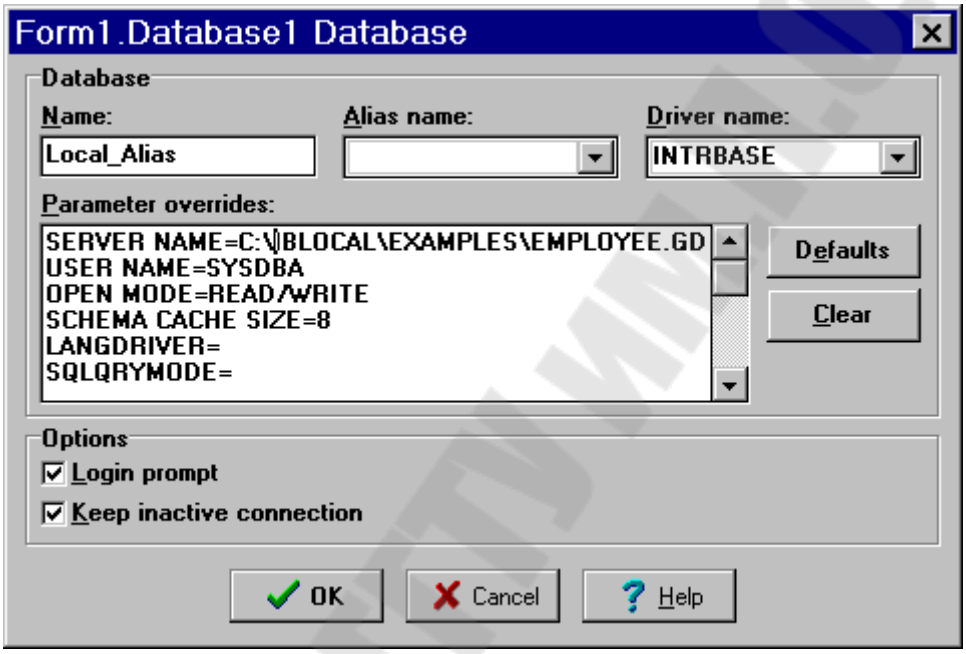
Здесь *Form2* - новое окно-диалог для ввода имени и пароля, *User_Name* и *User_Pass* - строки, куда сохраняются введенные имя и пароль.

Создание локального псевдонима базы данных

Обычно, псевдоним базы данных (Alias) определяется в утилите конфигурации BDE и информация о нем сохраняется в файле конфигурации IDAPI.CFG. Однако, в программе можно использовать не только ранее определенный в утилите конфигурации BDE псевдоним базы данных, но и так называемый локальный (т.е.

видимый только внутри данной программы) псевдоним. Это иногда бывает нужно, например, для того, чтобы обезопасить программу в случае удаления используемого псевдонима из файла конфигурации BDE.

Для того, чтобы создать локальный псевдоним БД, положите на главное окно проекта компонент DataBase1. Дальнейшие действия можно выполнить с помощью Инспектора Объектов, но удобнее это сделать через редактор компонент. Щелкните дважды мышкой на DataBase1 - появится диалог.



В этом диалоге требуется указать имя базы данных - это будет ее локальный псевдоним, на который ссылаются таблицы (свойство DatabaseName); тип драйвера (в нашем примере это INTRBASE); а также параметры, используемые при соединении с базой данных. Получить список параметров в поле “Parameter Overrides” можно по нажатию кнопки “Defaults”.

Набор параметров зависит от типа БД, с которой вы работаете. Этим параметрам нужно присвоить требуемые значения - указать путь к серверу, имя пользователя и т.д.

После выхода из редактора компонент имя, указанное в поле “Name” появится в списке имен баз данных для компонент типа TDataSet (TTable, TQuery etc.).

Изменение параметров при соединении

Иногда требуется изменить определенные в утилите конфигурации BDE параметры, используемые при установлении соединения с БД. Это можно сделать во время дизайна с помощью

диалога в поле “Parameter Overrides”. Либо во время выполнения программы (до попытки соединения) прямым присвоением свойству Params объекта DataBase1:

DataBase1.Params.Add(‘LANGDRIVER=ancyrr’);

Управление транзакциями

TDataBase позволяет начать в БД транзакцию (метод StartTransaction), закончить (Commit) или откатить ее (RollBack). Кроме того, можно изменять уровень изоляции транзакций (свойство TransIsoltion).

Объект Session

Объект Session, имеющий тип TSession создается автоматически в программе, работающей с базами данных (в этом случае Delphi подключает в программу модуль DB).

Вам не нужно заботиться о создании и уничтожении данного объекта, но его методы и свойства могут быть полезны в некоторых случаях.

В этом компоненте содержится информация обо всех базах данных, с которыми работает программа. Ее можно найти в свойстве *DataBases*. Со свойством *KeepConnections* данного объекта мы уже знакомы. Это свойство определяет, нужно ли сохранять соединение с базой, если в программе нет ни одной открытой таблицы из этой базы. *NetDir* - директория, в которой лежит общий сетевой файл PDOXUSRS.NET, необходимый BDE. *PrivateDir* - директория для хранения временных файлов.

С помощью методов объекта Session можно получить информацию о настройках BDE, например, список всех псевдонимов, драйверов баз данных или список всех таблиц в базе.

Еще одно важное назначение объекта Session - доступ с его помощью к таблицам Paradox, защищенным паролем. Прежде, чем открыть такую таблицу, требуется выполнить метод *AddPassword :*

Session.AddPassword(‘my_pass’);

Удалить пароль можно с помощью метода RemovePassword или RemoveAllPasswords.

Указание сетевого протокола при соединении с БД

В случае с InterBase можно в явном виде указать, какой сетевой протокол используется при соединении с базой данных. Эта установка выполняется либо в утилите конфигурации BDE, либо в программе - нужно изменить параметр “SERVER NAME”, который содержит полный путь к файлу с базой данных.

Транзакций

Все операции, выполняемые с данными на SQL сервере, происходят в контексте *транзакций.*

Транзакция - это групповая операция, т.е. набор действий с базой данных; самым существенным для этих действий является правило *либо все, либо ни чего*.

Если во время выполнения данного набора действий, на каком-то этапе невозможно произвести очередное действие, то нужно выполнить возврат базы данных к начальному состоянию (произвести откат транзакции).

Таким образом (при правильном планировании транзакций), обеспечивается целостность базы данных.

SQL-выражения для управления транзакциями

Для управления транзакциями имеется три выражения:

SET TRANSACTION - Начинает транзакцию и определяет ее поведение.

COMMIT - Сохраняет изменения, внесенные транзакцией, в базе данных и завершает транзакцию.

ROLLBACK - Отменяет изменения, внесенные транзакцией, и завершает транзакцию.

Запуск транзакции

Выполнять транзакции можно, например, из Windows Interactive SQL, из программы, из сохраненной процедуры. В общем виде, синтаксис команды SQL для запуска транзакции:

SET TRANSACTION [Access mode] [Lock Resolution]
[Isolation Level] [Table Reservation]

Значения, принимаемые по-умолчанию:

выражение

SET TRANSACTION

равносильно выражению

SET TRANSACTION READ WRITE WAIT ISOLATION LEVEL SNAPSHOT

Access Mode - определяет тип доступа к данным. Может принимать два значения:

- READ ONLY - указывает, что транзакция может только читать данные и не может модифицировать их.
- READ WRITE - указывает, что транзакция может читать и модифицировать данные. Это значение принимается по умолчанию.

Пример:
SET TRANSACTION READ WRITE
Isolation Level - определяет порядок взаимодействия данной транзакции с другими в данной базе. Может принимать значения:

- SNAPSHOT - значение по умолчанию. Внутри транзакции будут доступны данные в том состоянии, в котором они находились на момент начала транзакции. Если по ходу дела в базе данных появились изменения, внесенные другими завершенными транзакциями, то данная транзакция их не увидит. При попытке модифицировать такие записи возникнет сообщение о конфликте.
- SNAPSHOT TABLE STABILITY - предоставляет транзакции исключительный доступ к таблицам, которые она использует. Другие транзакции смогут только читать данные из них.
- READ COMMITTED - позволяет транзакции видеть текущее состояние базы.

Конфликты, связанные с блокировкой записей происходят в двух случаях:

- Транзакция пытается модифицировать запись, которая была изменена или удалена уже после ее старта. Транзакция типа READ COMMITTED может вносить изменения в записи, модифицированные другими транзакциями после их завершения.
- Транзакция пытается модифицировать таблицу, которая заблокирована другой транзакцией типа SNAPSHOT TABLE STABILITY.

Lock Resolution - определяет ход событий при обнаружении конфликта блокировки. Может принимать два значения:

- WAIT - значение по умолчанию. Ожидает разблокировки требуемой записи. После этого пытается продолжить работу.
- NO WAIT - немедленно возвращает ошибку блокировки записи.

Table Reservation - позволяет транзакции получить гарантированный доступ необходимого уровня к указанным таблицам. Существует четыре уровня доступа:

- PROTECTED READ - запрещает обновление таблицы другими транзакциями, но позволяет им выбирать данные из таблицы.

• PROTECTED WRITE - запрещает обновление таблицы другими транзакциями, читать данные из таблицы могут только транзакции типа SNAPSHOT или READ COMMITTED.

• SHARED READ - самый либеральный уровень. Читать могут все, модифицировать - транзакции READ WRITE.

• SHARED WRITE - транзакции SNAPSHOT или READ COMMITTED READ WRITE могут модифицировать таблицу, остальные - только выбирать данные.

Завершение транзакции

Когда все действия, составляющие транзакцию успешно выполнены или возникла ошибка, транзакция должна быть завершена, для того, чтобы база данных находилась в непротиворечивом состоянии.

Для этого есть два SQL-выражения:

* COMMIT - сохраняет внесенные транзакцией изменения в базу данных. Это означает, что транзакция завершена успешно.

* ROLLBACK - откат транзакции. Транзакция завершается и никаких изменений в базу данных не вносится. Данная операция выполняется при возникновении ошибки при выполнении операции (например, при невозможности обновить запись).

Управление транзакциями в Delphi

Прежде всего, транзакции в Delphi бывают явные и неявные.

Явная транзакция - это транзакция, начатая и завершенная с помощью методов объекта DataBase: *StartTransaction, Commit, RollBack.* После начала явной транзакции, все изменения, вносимые в данные относятся к этой транзакции.

Другого способа начать явную транзакцию, нежели с использованием DataBase, нет. (Точнее говоря, такая возможность есть, но это потребует обращения к функциям API InterBase. Однако, это уже достаточно низкоуровневое программирование.) Следовательно, в рамках одного соединения нельзя начать две транзакции.

Неявная транзакция стартует при модификации данных, если в данный момент нет явной транзакции. Неявная транзакция возникает, например, при выполнении метода *Post* для объектов *Table* и *Query*. То есть, если Вы отредактировали запись, в DBGrid и переходите на другую запись, то это влечет за собой выполнение *Post*, что, в свою

очередь, приводит к началу неявной транзакции, обновлению данных внутри транзакции и ее завершению.

Важно отметить, что неявная транзакция, начатая с помощью методов *Post, Delete, Insert, Append* и т.д. заканчивается автоматически.

Для модификации данных может использоваться и PassThrough SQL - SQL-выражение, выполняемое с помощью метода *ExecSQL* класса TQuery. Выполнение модификации через PassThrough SQL также приводит к старту неявной транзакции.

Дальнейшее поведение транзакции, начатой таким путем, определяется значением параметра SQLPASSTHRU MODE для псевдонима базы данных (или тот-же параметр в св-ве *Params* объекта DataBase).

Этот параметр может принимать три значения:

* SHARED AUTOCOMMIT - слово SHARED указывает на то, что оба вида транзакций(через Passthrough SQL и через методы TTable и TQuery) разделяют одно и то же соединение к базе данных. Слово AUTOCOMMIT указывает на то, что неявная транзакция, начатая через Passthrough SQL, завершается после выполнения действия по модификации данных (автоматически выполняется COMMIT).
* SHARED NOAUTOCOMMIT - отличается от предыдущего тем, что неявная транзакция, начатая через Passthrough SQL, не завершается после выполнения, ее нужно явно завершить, выполнив SQL-выражение "*COMMIT*".
* NOT SHARED - транзакции разных типов работают через разные соединения с базой. Данное значение параметра подразумевает также NOAUTOCOMMIT. То есть все неявные PassthroughSQL-транзакции нужно завершать явно - выполняя SQL-выражение "*COMMIT*" для Passtrough SQL.

Рассмотрим возможные сценарии поведения транзакций при разных значениях параметра.

В первом случае, если нет в данный момент начатой транзакции, то попытка модификация данных методами TTable или TQuery, как и выполнение через Passtrough SQL какой-либо операции приведет к старту неявной транзакции. После выполнения, такая транзакция будет автоматически завершена (если не возникло ошибки по ходу транзакции). Если уже имеется начатая явно (метод *StartTransaction*

объекта DataBase) транзакция, то изменения будут проходить в ее контексте. Все транзакции используют одно и то-же соединение.

Во втором случае все происходит, как в первом. Отличие в том, что неявная PassthroughSQL-транзакция не завершается, пока не будет выполнена команда "*COMMIT*".

В третьем случае, при выполнении команды Passthrough SQL, будет установлено еще одно соединение, начата неявная транзакция и выполнены действия по модификации данных. Транзакция не будет завершена, пока не будет выполнена команда "*COMMIT*". Наличие транзакции, начатой явно с помощью DataBase никак не отразится на ходе выполнения PassthroughSQL-транзакции. Пока PassthroughSQL-транзакция не завершится, изменения, внесенные ей, не будут видны в объектах Table и Query, работающих через другое соединение. PassthroughSQL-транзакции можно рассматривать в некотором смысле, как транзакции из другого приложения.

Взаимодействие транзакций данной программы с транзакциями из других приложений определяется свойством *TransIsolation* объекта DataBase.

Тема 14. Создание отчетов. Rave Reports. QReport.

Отчеты предназначены для быстрого получения визуальной информации из данных.

Для решения данной проблемы, на наиболее информационный манер, традиционные генераторы отчетов предлагают, секционное, в стиле таблиц представление данных.

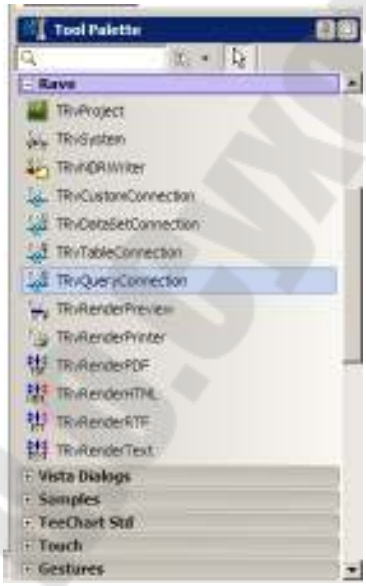
На сегодняшний день имеются более сложные требования к отчетам, которые не так легко решить стандартными секционно-табличными средствами.

Визуальный Rave дизайнер предлагает многие уникальные свойства, которые помогут сделать процесс получения отчетов проще, быстрее и более понятным.

Rave является интуитивной, визуальной средой разработки, которая может легко управлять широким спектром отчетов, много больше, чем чистый, рассчитанный на секции дизайнер.

Rave также включает зеркальное отражение и другие технологии, для повторного использования содержимого ваших

отчетов для быстрого внесения изменений и боле простого обслуживания.



В действительности, Rave был специально разработан для получения гибкости и функциональности, в простом для освоения формате.

В инсталляцию Rave включен проект, названный "RaveDemo", который содержит примеры различных отчетов.

Имеются два типа объектов в Rave, компоненты вывода (Output Components) и классы отчета (Report Classes). Компоненты вывода отвечают за вывод отчета на различные устройства вывода, а классы отчета, которые не являются компонентными классами, отвечают за все остальные задачи.

Компоненты вывода

TrvSystem — Включает в себя стандартный принтер и предварительный просмотр и является одним из самых простых в использовании компонент.

TRvNDRWriter — Создает NDR поток или файл (в должном формате) при выполнении отчета.

TrvRenderPreview — Показывает диалог предварительного просмотра для NDR потока или файла.

TrvRenderPrinter — Посылает NDR поток или файл на принтер.

TRvRenderPDF — Преобразовывает NDR поток или файл в PDF формат.

TRvRenderHTML — Преобразовывает NDR поток или файл в HTML формат.

TRvRenderRTF — Преобразовывает NDR поток или файл в RTF формат.

TRvRenderText — Преобразовывает NDR поток или файл в Text формат.

Классы Rave

Rave использует данные из вашего приложения. Стандартные компоненты для доступа к данным, TRvCustomConnection, TrvDataSetConnection TRvTableConnection и TrvQueryConnection представляют собой мост между данными вашего приложения и визуальными компонентами Rave.

TrvProject — Производит соединение к проекту отчета, который был создан с помощью визуального редактора Rave. Используйте данный компонент для получения списка всех доступных отчетов или для выполнения конкретного отчета.

TrvCustomConnection — Подсоединяет пользовательские данные (сгенерированные через события) к DirectDataViews, созданные с помощью визуального редактора Rave.

TrvDataSetConnection — Подсоединяет TDataSet данные (например, TClientDataSet, или компоненты третьих сторон, наследники от TDataSet) к DirectDataViews, созданные с помощью визуального редактора

	Rave.
 TrvTableConnection	Подсоединяет TTable компоненты к DirectDataViews, созданные с помощью визуального редактора Rave.
 TrvQueryConnection	Подсоединяет TQuery компоненты к DirectDataViews, созданные с помощью визуального редактора Rave.

Компонент TRvProject

Компонент TRvProject является ключом для доступа к визуальным отчетам, создаваемым с помощью Rave. Обычно у вас только один компонент TRvProject на все приложения, но при нужде Вы можете иметь их столько, сколько нужно. Свойство ProjectFile определяет файл проекта вашего приложения, в котором хранятся все определения отчета. Данный файл имеет расширение .RAV и даже если это единственный файл он может хранить столько определений отчета сколько необходимо.

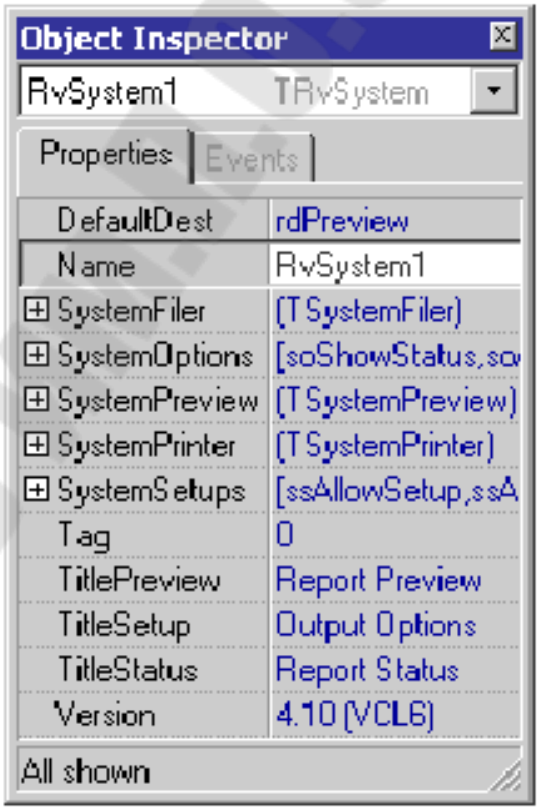
Когда вызывается метод Open объекта TRaveReport, то данный файл загружается в память для подготовки к печати или для изменений пользовательским дизайнером. Вы должны обязательно вызвать метод Close, как только вам не нужен файл проекта или при закрытии вашего приложения. Любые изменения в проекте отчета Вы можете сохранить, вызвав метод Save. TRvProject также имеет несколько свойства и методов, такие как SelectReport, GetReportList, ReportDescToMemo, ReportDesc, ReportName и ReportFullName, что делает эффективным и простым создание интерфейса для ваших пользователей.

Компонент TRvSystem

Описание

Компонент TRvSystem очень мощный компонент, который интегрирует функциональность сразу трех компонент, TRvRenderPreview, TRvRenderPrinter и TRvNDRWriter в одну простую для использования систему. TRvSystem может посылать отчет или на принтер или на экран для просмотра и может показывать диалог настройки (setup) и окно состояния.

Свойства



DefaultDest указывает, куда будет посылаться отчет, если не используется окно настройки или параметры по умолчанию для диалога настройки.

SystemFiler, может показывать все файловые настройки из TRvNDRWriter, TRvRenderPreview и TRvRenderPrinter. Все настройки *SystemFiler* работают так же, как и в других компонентах, за исключением потокового режима для *smMemory,* который не требует указания имени файла, а должен использовать TMemoryStream содержащий отчет.

SystemOptions управляет конфигурацией компонента TRvSystem.

soUseFiler всегда посылает отчет в файл отчета. Это особенно полезно, если в отчете используется метод Macro.

soWaitForOK позволяет указать, требуется ли нажатие кнопки OK пользователем, после того, как отчет будет готов для вывода.

soShowStatus позволяет указать, требуется ли вывод окна состояния при подготовке или печати отчета.

soAllowPrintFromPreview позволяет указать может ли пользователь печатать отчет из окна просмотра отчета.

soPreviewModal позволяет указать модальный режим работы окна просмотра.

soNoGenerate позволяет пропустить фазу генерации отчета и произвести вывод сразу на экран. Данная настройка должна быть

использована только совместно со *StreamMode* и *smFile,* когда отчет был ранее подготовлен и необходимо его только или просмотреть или распечатать.

SystemPreview показывает все настройки просмотра, найденные в TRvRenderPreview.

SystemPrinter показывает все настройки печати, найденные в TRvNDRWriter.

SystemSetups управляет конфигурацией стандартного диалога настроек для TRvSystem.

ssAllowSetup определяет должен ли быть показан диалог настройки.

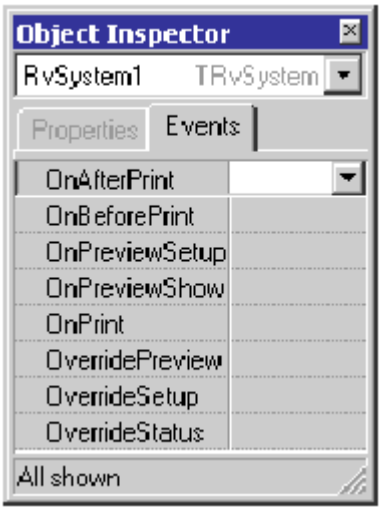
ssAllowCopies, *ssAllowCollate* и *ssAllowDuplex* разрешают включение данных настроек в диалог настройки принтера.

ssAllowDestPreview, *ssAllowDestPrinter* и *ssAllowDestFile* определяют настройки вывода, до которых пользователь может иметь доступ.

ssAllowPrinterSetup определяет, может ли пользователь вызвать диалог настройки принтера, в котором можно сменить принтер и другие параметры принтера.

ssAllowPreviewSetup определяет, может ли пользователь вызвать диалог настройки принтера после просмотра.

События



Все OnXxxx события для TRvSystem работают одинаково, как у TRvNDRWriter. Перекрытые события, *OverridePreview*, *OverrideSetup* и *OverrideStatus* позволяют программисту заменить эти диалоги своими собственными. Документация о том, как это сделать это отсутствует, но можно посмотреть, как это сделано в TRvSystem. Посмотрите методы *OverridePreviewProc*, *OverrideStatusProc* и *OverrideSetupProc* как создать перекрытый метод событие. Модули RpFormPreview, RpFormStatus и RpFormSetup расположены в \RAVE4\SOURCE, также показывают, как взаимодействовать с TRvSystem, и могут быть использованы как начальная точка для собственных версий различных форм.

Подключения данных

Мост к данным

Подключения данных (data connections) предоставляют мост между данными в вашем приложении и Rave отчетом. Первая вещь, которую вы должны сделать, это выбор типа подключения данных. Определяется это типом компонентов баз, которые Вы используете. Ниже приведенная таблица поможет вам сделать выбор, какой тип подключения лучше использовать:

Компонент подключения данных	**Лучшее использовать с …**	**Как подключить**
TRvCustomConnection	Массивы, обычне файлы или не TDataSet компоненты	Определить события, такие как OnFirst, OnNext, OnEOF, OnGetCols и OnGetRow.
TRvDataSetConnection	Наследники TDataSet от третьих сторон	Установите свойство DataSet соответствующему компоненту TDataSet
TRvTableConnection	TTable, заменитель TTable или его наследники	Установить свойство Table компоненты

Об именовании компонент подключения данных

Свойство Name компоненты подключения данных используется для именования подключения. Очень важно использовать уникальные имена для ваших компонент подключений, поскольку не может быть дублирования в вашем приложении.

Также является хорошей практикой использование уникальных идентификатор в имени подключения, поскольку все подключения данных видны во всех других приложениях. Например, если ваше приложение называется Wizbang Object Wizard, то вы можете предварять имена префиксом WOW в каждом подключении, что бы имена ваших подключений были уникальными.

Управление видимостью подключения

Свойство Visible компоненты подключения данных используется для управления видимостью подключения для других приложений или версий визуального дизайнера, поставляемых конечному пользователю. Значение по умолчанию FALSE означает, что подключение данных будет видно только в отчете, печатаемом из

того же самого приложения или из поставляемой программистом версии визуального дизайнера Rave. Значение TRUE означает, что подключения будут видны любому отчету, запущенному из любого приложения или из пользовательской версии дизайнера Rave.

Хотя подключения данных видны в дизайнере конечного пользователя, они не показываются при попытке создать новое представление данных, поскольку они принадлежат приложению, запустившему данный дизайнер. Если Вы желаете сделать подключение из другого приложения доступным конечному пользователю, то лучшим методом будет создание проекта отчета с программисткой версией визуального дизайнера, в которой будут уже определены представления для внешних подсоединений данных.

Настройка подключений данных

Использование событий для настройки подключений данных

Вы можете управлять, как данные будут поступать в ваш отчет через события компонент подключения данных. Для данных, поступающих не из баз, используйте компонент TrvCustomConnection, Вы можете предоставить доступ к вашим данным через его события. Для подключения данных из баз, используйте компоненты, такие как TRvDataSetConnection, достаточно будет переписать событие OnValidateRow.

События подключений данных

Событие	**Описание**
OnEOF	Вызывается, когда Rave желает определить, достигнут ли конец данных. При отсутствии данных требуется вернуть значение TRUE, если больше нет строк или если вызов события OnNext был сделан за пределы последней строки.
OnFirst	Вызывается, когда Rave желает, что бы курсор был перемещен на первую строку данных. С развитой системой буферизации Rave, это событие обычно возникает только раз в начале сессии данных.
OnGetCols	Вызывается, когда Rave желает получить мета данные. Это включает в себя имена полей, типы, размеры символов, полные имена и описание. Подробности смотри ниже.
OnGetRow	Вызывается, когда Rave желает получить данные для текущей строки. Подробности смотри ниже.

OnGetSorts	Вызывается, когда Rave желает получить доступные методы сортировки.
OnNext	Вызывается, когда Rave желает переместить курсор на следующую строку.
OnOpen	Вызывается, когда Rave желает инициализировать начало сессии. Должно быть сохранено текущее состояние, что бы потом его можно было восстановить в событии OnRestore.
OnRestore	Вызывается, когда Rave желает восстановить состояние предыдущей сессии, которое было перед ее открытием.
OnSetFilter	Вызывается, когда Rave желает фильтровать данные, такие как Master-Detail отчеты.
OnSetSort	Вызывается, когда Rave желает сортировать данные. Подробности смотри ниже.
OnValidateRow	Вызывается для каждой строки, позволяя тем сам управлять фильтрацией данных. Для пользовательских приложений, данное событие обычно не нужно, поскольку фильтрация данных через событие OnNext более эффективно. Тем не менее, данное событие очень полезно для других, более автоматизированных подключений, таких как TRvDataSetConnection. Подробности смотри ниже.

TRvCustomConnection

Компонент TRvCustomConnection имеет свойства DataIndex и DataRows типа integer. Они предназначены для использования в событиях пользовательских подключений и если используются, то могут быть определены в OnFirst, OnNext и OnEOF событиях. DataIndex используется как позиция курсора, первая строка имеет номер 0. DataRows используется как счетчик строк данных. Например, если вы определяете пользовательское подключение к массиву данных, вам будет достаточно установить свойство Connection.DataRows равным количеству элементов в массиве и затем позволить Rave управлять OnFirst, OnNext и OnEOF событиями. В событии OnGetRow, Вы должны, обратиться к свойству Connection.DataIndex для определения какой элемент массива требуется вернуть назад (не забывайте, что нумерация идет с 0).

Событие OnGetCols

Событие OnGetCols вызывается, когда Rave желает получить мета данные. Внутри этого события Вы захотите вызвать метод Connection.WriteField для каждого поля (колонки) ваших данных. Определение WriteField следующее:

```
procedure WriteField(Name: string;
   DataType: TRpDataType;
   Width: integer;
   FullName: string;
   Description: string);
```

Name это короткое имя поля и должно состоять только из алфавитно-цифровых символов. DataType это тип данных поля и должен быть одним из следующих типов: *dtString, dtInteger, dtBoolean, dtFloat, dtCurrency, dtBCD, dtDate, dtTime, dtDateTime, dtBlob, dtMemo или dtGraphic*. Width это относительная ширина поля в символах. Full name это полное, более описательное имя поля и может включать в себя пробелы и другие не алфавитно-цифровые символы. Если FullName пустое, то будет использовано короткое имя поля. Description это полное описание поля и обычно редактируется с помощью компонента, поэтому может состоять из нескольких строк. Используйте свойство Description для описания, как используется поле или для любой другой более, нужной информации насчет данного поля.

Пример:

```
procedure TDataForm.CustomCXNGetCols(Connection:
TRvCustomConnection);
begin
   With Connection do begin
      WriteField('Index',dtInteger,8,'Index Field','Описание 1');
      WriteField('Name',dtString,30,'Name Field','Описание 2');
      WriteField('Amount',dtFloat,20,'Amount Field','Описание 3');
   end; { with }
end;
```

Событие OnOpen

Событие OnOpen возбуждается при инициализации сессии данных. В этом событие Вы можете открыть файлы данных, инициализировать переменные и сохранить текущее состояние данных для события OnRestore, которое будет возбуждено при закрытии сессии данных.

Пример:

```
procedure TDataForm.CustomCXNOpen(Connection:
TRvCustomConnection);
begin
   AssignFile(DataFile,'DATAFILE.DAT');
   Reset(DataFile,1);
end;
```

Событие OnFirst

Событие OnFirst вызывается, когда требуется перемещение курсора данных на первую строку данных.

Пример:

```
procedure TDataForm.CustomCXNFirst(Connection:
TRvCustomConnection);
begin
   Seek(DataFile,0);
   BlockRead(DataFile,DataRecord,SizeOf(DataRecord),DataRead);
end;
```

Событие OnNext

Событие OnNext вызывается, когда требуется перемещение курсора данных на следующую строку данных.

Пример:

```
procedure TDataForm.CustomCXNNext(Connection:
TRvCustomConnection);
begin
   BlockRead(DataFile,DataRecord,SizeOf(DataRecord),DataRead);
end;
```

Событие OnEOF

Событие OnEOF вызывается для возврата состояния курсора данных, находится ли он на данных или уже вышел за конец. Значение TRUE должно быть возвращено, если данных больше нет или если вызов события OnNext привел к выходу из последней строки.

Пример:

```
procedure TMainForm.CustomCXNEOF(Connection:
TRvCustomConnection;
   var EOF: Boolean);
begin
   EOF := DataRead < SizeOf(DataRecord);
end;
```

Событие OnGetRow

Событие OnGetRow вызывается для получения данных для текущей строки. Имеется несколько методов для записи данных в специальные буферы используемые Rave. Порядок и типы записываемых полей должен быть точно таким же, как полученные определения полей в событии OnGetCols.

В следующем списке приведены методы объекта Connection для записи данных в буфера.

```
procedure WriteStrData(FormatData: string; NativeData: string);
{dtString}
procedure WriteIntData(FormatData: string; NativeData: integer);
{dtInteger}
procedure WriteBoolData(FormatData: string; NativeData: boolean);
{dtBoolean}
procedure WriteFloatData(FormatData: string; NativeData: extended);
{dtFloat}
procedure WriteCurrData(FormatData: string; NativeData: currency);
{dtCurrency}
procedure WriteBCDData(FormatData: string; NativeData: currency);
{dtBCD}
procedure WriteDateTimeData(FormatData: string; NativeData:
TDateTime);
      {dtDate, dtTime and dtDateTime}
procedure WriteBlobData(var Buffer; Len: longint);
      {dtBlob, dtMemo and dtGraphic}
```

Также имеется специальный метод, названный WriteNullData (без параметров), который может быть использован для некоторых полей, для указания неинициализированных данных (null). Параметр FormatData используется для передачи строки форматирования данных для данного поля. Параметр NativeData предназначен для передачи неформатированных или чистых данных поля. Если строка форматирования определена в отчете Rave, то она используется для форматирования, иначе используется FormatData.

Пример:

```
procedure TDataForm.CustomCXNGetRow(Connection:
TRvCustomConnection);
begin
   With Connection do begin
      WriteIntData('',DataRecord.IntField);
```

```
      WriteStrData(',DataRecord.StrField);
      WriteFloatData(',DataRecord.FloatField);
    end; { with }
  end;
```

Событие OnValidateRow

Событие OnValidateRow возникает для каждой строки данных, позволяя управлять включением строки данных в отчет или нет. Обычно это единственное событие, которое определяется для не пользовательских подключений.

Пример:

```
  procedure TDataForm.CustomCXNValidateRow(Connection:
                    TRvCustomConnection; var ValidRow: Boolean);
  begin
    ValidRow := DataRecord.FloatField >= 0.0;
  end;
```

Событие OnRestore

Событие OnRestore для прекращения текущей сессии и восстановления предыдущего состояния. В этом событие Вы можете закрыть файлы данных, освободить ресурсы и восстановить предыдущее состояние, которое было перед событием OnOpen.

Пример:

```
  procedure TDataForm.CustomCXNRestore(Connection:
  TRvCustomConnection);
  begin
    CloseFile(DataFile);
  end;
```

Компонент TRvNDRWriter

Описание

Компонент TRvNDRWriter используется совместно с TRvRenderPrinter и TRvRenderPreview для записи отчета в специальный двоичный формат, до тех пор, когда он потребуется для печати или просмотра.

Свойства и события

Компонент TRvNDRWriter имеет свойства и события для управления выводом.

AccuracyMethod определяет, каким образом выводятся строки, для получения должного качества печати и просмотра.

FileName это файл, который будет создан, если StreamMode содержит, что отличное от smUser. Используйте smFile для больших отчетов (более 10 страниц или с большим количеством рисунков) и smMemory для маленьких отчетов (менее 10 страниц).

Для отсылки отчета в файл используйте метод Execute.

Компоненты вывода

TRvRenderPreview

Компонент TRvRenderPreview берет файл, созданный компонентом TRvNDRWriter и посылает его на экран для просмотра. TRvRenderPreview имеет много методов и событий, которые позволяют программисту создавать сложный, настраиваемый пользовательский интерфейс.

Свойства TRvRenderPreview

ScrollBox определяет компонент TscrollBox, на котором и рисуется отчет.

FileName и *StreamMode* используются аналогично TRvNDRWriter и TRvRenderPreview.

GridHoriz и *GridVert* определяют горизонтальное и вертикальное расстояние, в дюймах или метрике, между каждой отметкой, рисуемой с помощью *GridPen*.

RulerType совместно с настройками, удобен для разработки точных отчетов, без необходимости производить тестовую печать.

MarginMethod и *MarginPercent* определяет и размер непечатных границ листа.

ShadowDepth определяет количество пикселей для тени листа.

Monochrome определяет, каким битмапом должен быть вывод - монохромным или цветным. Сбрасывается при вызове *NextPage* или *PrevPage*.

ZoomInc определяет величину увеличения или уменьшения для *ZoomIn* и *ZoomOut* при изменении процента текущего zoom, *ZoomFactor*.

События TRvRenderPreview

OnPageChange вызывается, когда сменяется текущая страница и позволяет программисту обновить пользовательский интерфейс для нового номера страницы.

OnZoomChange вызывается, когда изменяется текущий коэффициент масштабирования, *ZoomFactor*, что позволяет программисту обновить пользовательский интерфейс для нового коэффициента масштабирования.

Описание TRvRenderPrinter

Компонент TRvRenderPrinter принимает файл созданный компонентом TRvNDRWriter и посылает его на текущий принтер. TRvRenderPrinter часто используется для печати из окна просмотра. TRvRenderPrinter это простой компонент, но и он имеет методы и свойства для управления печатью.

Свойства и события TRvRenderPrinter

FileName это имя файла отчета созданного компонентом TRvNDRWriter, если *StreamMode* равен *smMemory* или *smFile*. Потоковый режим *smUser* используется, когда программист хочет предоставить свой собственный потоковый объект (любой наследник от TStream) , назначив свойство *Stream* компоненту TRvNDRWriter, TRvRenderPrinter и/или TRvRenderPreview. У компоненты TrvRenderPrinter отсутствую свои события. Для посылки отчета на принтер сделайте вызов *Execute* или *ExecuteCustom*.

Описание TRvRenderPDF

Компонент TRvRenderPDF позволяет создавать документы PDF (Adobe Acrobat) прямо из отчета. Шрифты могут быть встроены в документ PDF, путем установки свойства *EmbedFonts* в TRUE. Качество рисунков, включенных в PDF файл можно установить через указание процента качества в свойстве *ImageQuality*. Сжатые PDF документы можно создать, если установить свойство *UseCompression* в TRUE, подключив ZLib в uses и определив событие OnCompress так:

```
With TCompressionStream.Create(clMax, OutStream) do try
   CopyFrom(InStream, InStream.Size);
finally
   Free;
end; { with }
```

Самый простой путь создания возможностей вывода в приложении – это бросить компонент на форму, которая автоматически

зарегистрирует этот формат и стандартные диалоги настройки и просмотра в TRvSystem. Если требуется более автоматический вывод, то можно вызвать метод *Render* с передачей или объекта NDR TStream или имени файла NDR как единственный параметр.

TRvRenderHTML

Компонент TRvRenderHTML преобразовывает поток NDR или файл в HTML страницы. Поддержаны текст, графика, линии и прямоугольники. Результат вывода в формате HTML 4.0 и разработан так, что бы вывод на печать был максимально точным.

TRvRenderRTF

Компонент TRvRenderRTF преобразовывает поток NDR или файл в формат RTF. Результат вывода в формате RTF и разработан так, что бы вывод на печать был максимально точным. Элементы в документе включаются как отдельные кадры "frames", для поддержки точного позиционирования на странице.

TRvRenderText

Компонент TRvRenderText преобразовывает поток NDR или файл в текстовый формат. Поддержаны только текстовые элементы, все остальные объекты, такие как графика или линии игнорируются. Свойство *CPI* позволяет указать количество символов на дюйм и свойство *LPI* позволяет указать количество строк на дюйм на выходное устройство. Примечание: конечный вывод на принтер может не совпадать с установками *CPI* или *LPI* поскольку в файл не вставляются никакие команды форматирования.

Среда RAVE
Report Authoring Visual Environment

Структура Rave

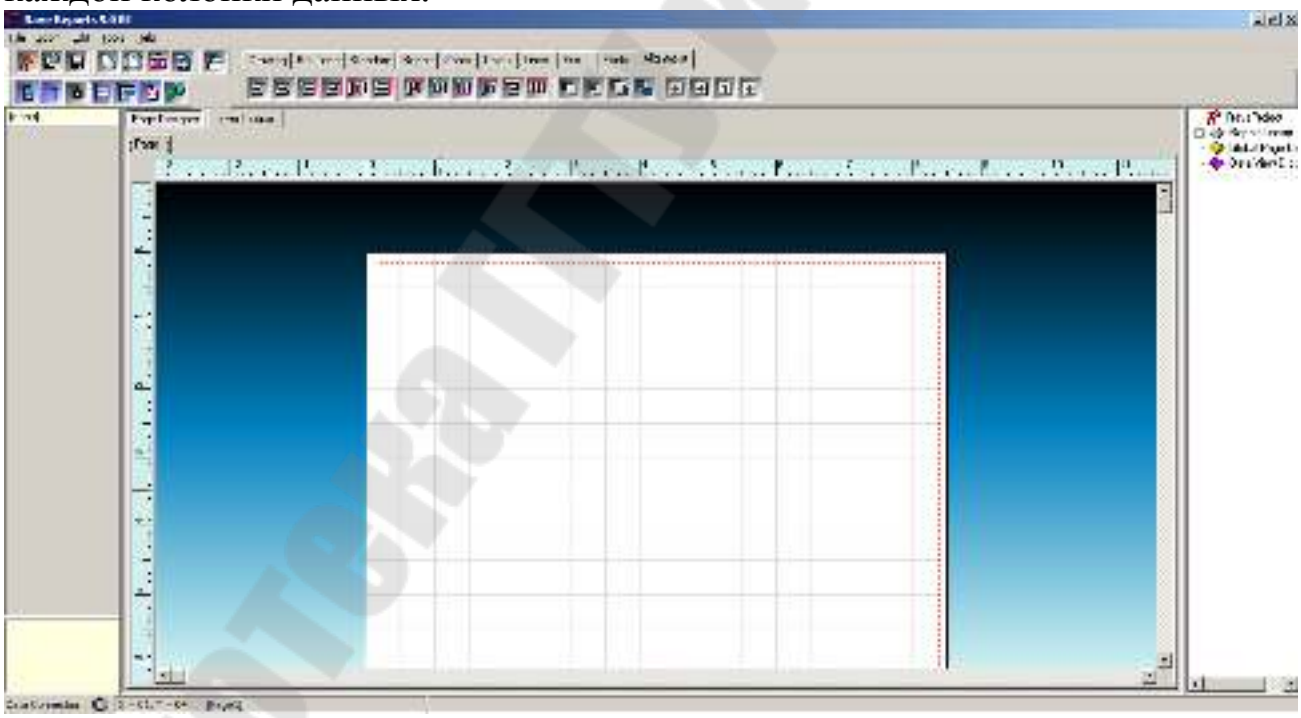
Файл проекта отчетов (.RAV) - файл проекта отчетов, в котором хранятся определения проекта, создается визуальным дизайнером отчетов Rave. Это двоичный файл, подобный файлу .DFM Delphi. Все отчеты, глобальные страницы и данные просмотров для проекта записываются в единственный файл. Вы можете экспортировать и импортировать элементы из файла или в файл проекта. Используя методы компоненты TRvProject, Вы можете также записывать файл проекта в поле базы данных, типа blob или в другое место.

Reports – библиотека (Report Library) страниц отчета проекта. Отчет Rave создает страницы отчета и визуальные компоненты на данных страницах. Вы можете создать столько определений страниц,

сколько пожелаете, и объединять их большим количеством различных методов.

Global Pages – глобальные страницы записываются в каталог глобальных страниц проекта. Компоненты на глобальных страницах, в отличие от страниц отчета, видны во всех отчетах. Глобальные страницы служат для хранения шаблонов, которые отражаются в других страницах отчета.

Data Views – представления данных (Data views) записываются в словарь представлений (Data View Dictionary) проекта. Представления предоставляют интерфейс к компонентам доступа. При создании нового представления, вы должны иметь активный компонент доступа, или в работающем приложении, или на загруженной Delphi или C++Builder форме. Просмотры затем запрашивают компоненты доступа к данным для получения мета информации о данные, такой как - имена полей, типы данных и т.д. Компоненты полей содержатся внутри каждого представления, позволяя установку свойств для каждой колонки данных.



Две группы панелей, отображаемых в дизайнере Rave:

Компоненты (Component)

Утилиты (Utility)

Компоненты, это такие элементы, которые "бросаются" или видны на странице редактора. *Это могут быть секции, штрих коды, линии, графические примитивы, и другие. Если объект виден на странице, то это компонент.*

Компонентные панели используются для помещения компонент, которые работают и имеют функции внутри отчета. Большинство из компонент видимы, при размещении на странице, но есть несколько не видимых компонент, не видимы, означает, что данные компоненты появляются только в дереве проектов.

Панели, которые могут изменять компоненты, являются панелями инструментов. Есть несколько таких панелей: выравнивания, палитра цветов, редактор шрифтов и другие.

Панели утилит используются для установки свойств, которые влияют на вид компонент в отчете, например, панель цвета может изменять цвет текста в компоненте DataText.

Пример создания отчета

Созданные отчеты Rave можно сохранять в удобном формате. Это позволяет экспортировать и открывать их из других программ (Adobe Reader, Internet Explorer, MS Word...) и в случае форматов .rtf и .txt редактировать.

Для начала создадим новый проект. Скинем на форму Ttable, TDataSource, TDbGrid. Свяжем их с таблицей Animals.dbf Создаваемый отчет будет содержать информацию из таблицы Table1.

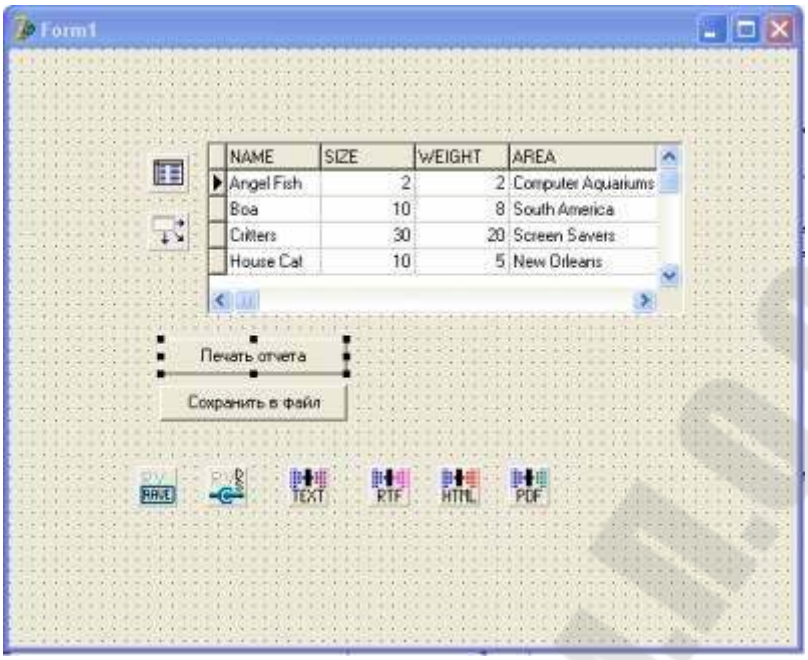
Для работы с Rave необходимо перенести на форму следующие компоненты:

для вывода и печати отчета TRvProject; TRvDataSetConnection;

для печати отчета в другие форматы:

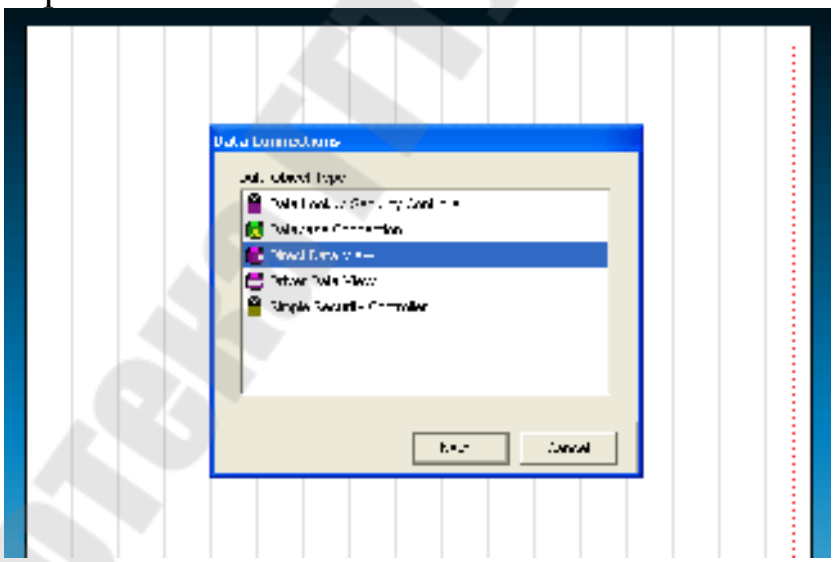
TRvNDRWriter;
TRvRenderText;
TRvRenderRTF;
TRvRenderHTML;
TRvRenderPDF.

Печать отчета и сохранение в отдельный файл будем осуществлять при нажатии на соответствующую кнопку. В результате полученная форма будет выглядеть так:

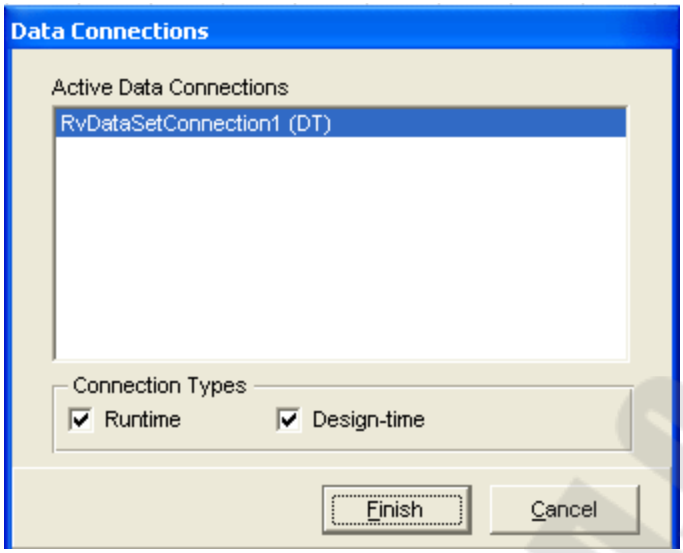


Теперь присвоим RvDataSetConnection1 свойство Dataset значение table1. Таким образом в Rave будут передаваться данные из Table1. Переходим в Rave Designer (Tools Rave Designer).

Автоматически будет создан новый проект. Свяжем проект с Table1 через RvDatasetConnection1. Для этого нажмите кнопку New Data Object и выберите Direct Data View

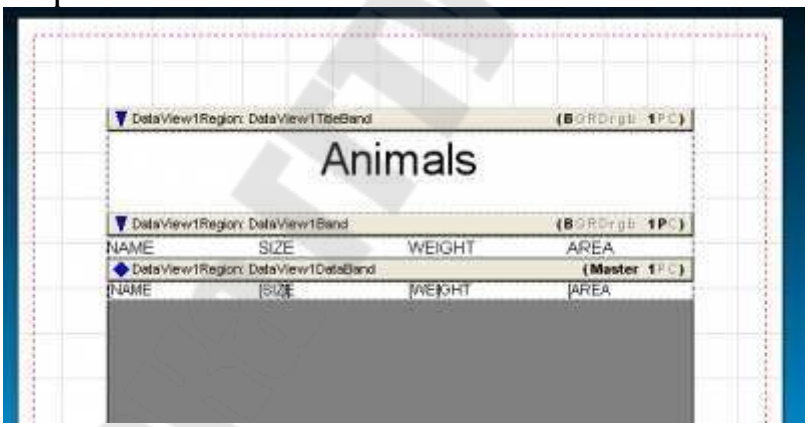


Нажимаем Next и в появившемся окне должны быть отражены все активные Rave соединения. В нашем случае это будет выглядеть так



После нажатия Finish справа в Rave Designer в Data View Dictionary появится Dataview1, раскрыв который можно увидеть поля Table1.
Теперь создадим быстро простой отчет. Для этого воспользуемся Tools\Report Wizards\Simple Table.

Выберим Dataview1, выделим все поля и завершим создание. После небольшой корректировки полученный лист отчета будет выглядеть следующим образом



В верхнем поле (TitleBand) заголовок страницы, ниже заголовки колонок и в последней помещаются DataText Component, которые будут принимать значения соответвующего поля (колонки).
Загляните в Report Library (справа) и посмортите, что бы имя текущего листа отчета было Report1. Сохраним полученный проект в папку с проектом Delphi. Получится файл Project1.rav.
Теперь вернемся к нашему проекту в Delphi и напишем обработчик события для кнопки печать отчета. Соединим RvProject1 с полученным файлом Project1.rav. Можно сделать это программно. И далее запустим наш отчет.

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
      RvProject1.SetProjectFile(ExtractFilePath(Application.ExeName)+'Pr
      oject1.rav');
      RvProject1.ExecuteReport('Report1');
      RvProject1.Close;
end;
```

Теперь в результате нажатия на кнопу мы увидим диалоговое окно, в котором можно выбрать печатать отчет на принтере, посмотреть перед печатью или сохранить в файл.
Уже сейчас можно выбрать и сохранить в файл в удобном для нас формате.

Далее рассмотрим как сделать это напрямую без диалогового окна через программный код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NdrStream: TMemoryStream;
  OutStream: TMemoryStream;
  name:string;
begin
  NdrStream := TMemoryStream.Create;
  OutStream := TMemoryStream.Create;
  RvProject1.Close;

RvProject1.SetProjectFile(ExtractFilePath(Application.ExeName)+'Project
1.rav');
  RvProject1.SelectReport('Report1',true);
  try
    RvNDRWriter1.StreamMode := smUser;
    RvNDRWriter1.Stream := NdrStream;
    RvProject1.Engine := RvNDRWriter1;
    RvProject1.Execute;
    name:='Report1';
    RvRenderText1.PrintRender(NdrStream,name+'.txt');
    RvRenderRTF1.PrintRender(NdrStream,name+'.rtf');
    RvRenderHTML1.PrintRender(NdrStream,name+'.html');
    RvRenderPDF1.PrintRender(NdrStream,name+'.pdf');
```

```
  finally
    FreeAndNil(NdrStream);
    FreeAndNil(OutStream);
  end;
  RvProject1.Close;
end;
```

В результате нажатия на кнопку будут созданы 4 файла в папке с программой: Report1.txt, Report1.rtf, Report1.html, Report1.pdf.

QReport

На странице палитры компонентов *QReport* расположено более двух десятков компонентов, применяемых для построения отчетов.

Центральным компонентом является *TQuickRep,* определяющий поведение отчета в целом. С помощью других компонентов создаются составные части отчета.

TQRBand – заготовка для расположения данных, заголовков, титула отчета и др. Отчет, в основном, строится из компонентов *TQRBand*, которые реализуют:

- область заголовка отчета;
- область заголовка страницы;
- область заголовка группы;
- область названий столбцов отчета;
- область детальных данных, предназначенную для отображения данных самого нижнего уровня детализации;
- область подвала группы;
- область подвала страницы;
- область подвала отчета.

TQRStringsBand – имеет то же назначение, что и *TQRBand*. Отличается встроенным списком строк *Items*, содержимое которого становится видным в режиме печати и предварительного просмотра, если на компонент *TQRStringsBand* положен компонент *TQRExpr*. Для каждой строки в *Items* выводится своя полоса *TQRStringsBand.*

TQRSubDetail – дочерняя полоса. Привязывается к родительской полосе и служит для ее расширения. Любая полоса может стать родительской с помощью установки значения *True* в ее свойство *HasChild*.

TQRGroup – применяется для группировок данных в отчетах.

TQRLabel – позволяет разместить в отчете произвольную текстовую строку.

TQRDBText – служит для вывода в отчет содержимого текстового поля набора данных.

TQRExpr – применяется для вывода значений, являющихся результатом вычислений выражений. Алгоритм вычисления выражений строится при помощи редактора формул данного компонента.

TQRSysData – служит для вывода в отчете системной величины: даты, времени, номера страницы и т.п.

TQRMemo – вставляет в отчет многостраничный текст.

TQRExprMemo – используется для создания многострочных вычисляемых полей.

TQRRichText – вставляет в отчет многострочный текст в формате *RTF*.

TQRDBRichText – служит для вывода в отчете полей НД, содержащих многострочный текст в формате *RTF*.

TQRShape – служит для вывода в отчете графических фигур, например, прямоугольников.

TQRImage – служит для вывода в отчете графической информации, источником которой является поле набора данных.

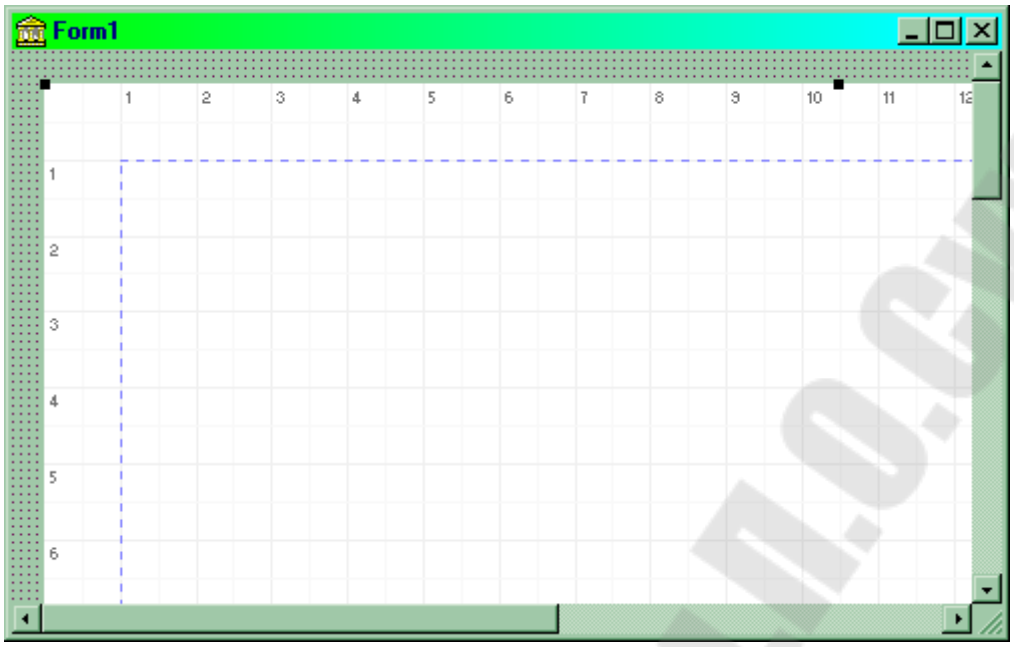
TQRPreview – базовый компонент для создания нестандартных окон предварительного просмотра. Стандартное окно реализуется с помощью метода *Preview* компонента *TQuickRep*.

TQRXXXFilter – фильтрующие компоненты для преобразования отчета в текст, страницу *HTML* и т.п. при печати отчета.

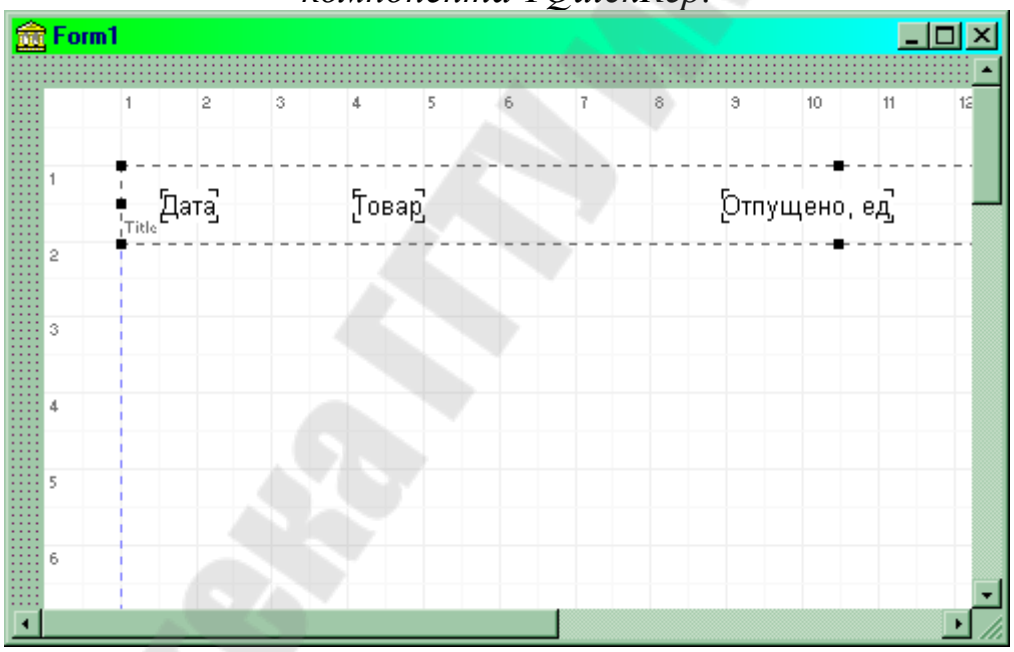
TQRChart – служит для встраивания в отчет графиков.

Chapter 11 Компонент TQuickRep

При размещении этого компонента на форме в ней появляется сетка отчета (рис.1). В дальнейшем в этой сетке располагаются составные части отчета, например, полосы *TQRBand* (рис.2).



Пустая сетка отчета. Образуется после размещения на форме компонента TQuickRep.



Сетка отчета с размещенными в ней компонентами отчета.

Перечислим важнейшие свойства, методы и события компонента *TQuickRep*.

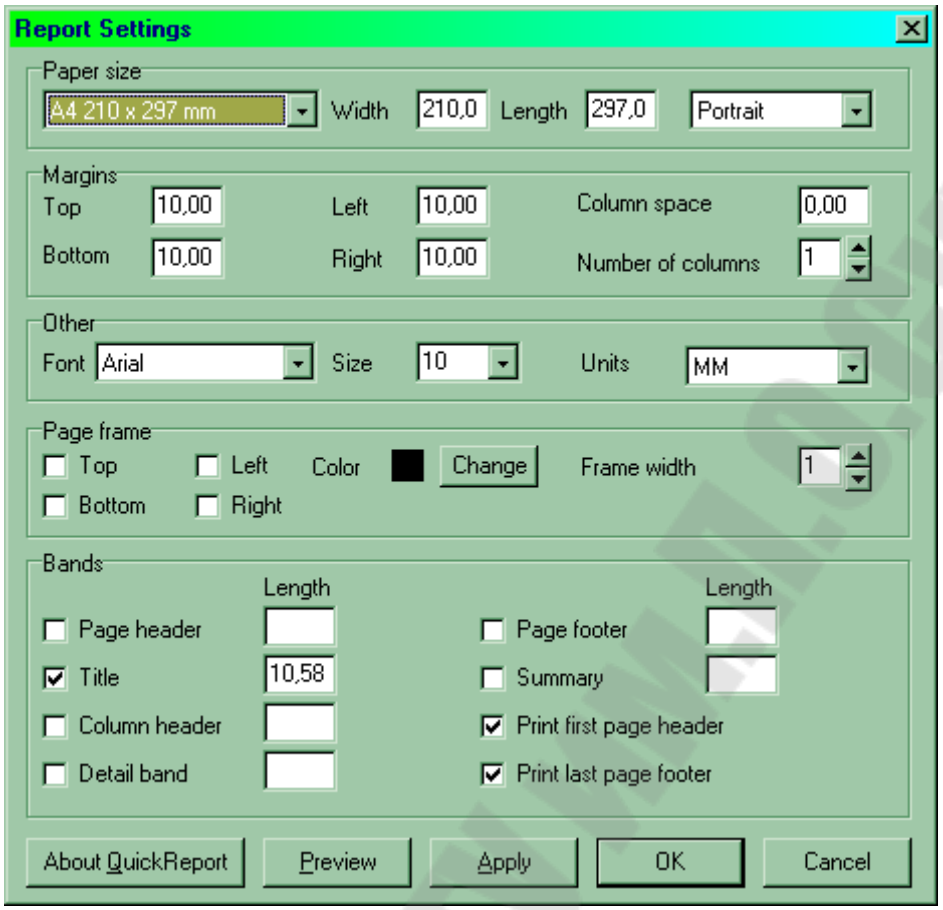
Свойства

Свойство	**Назначение**
property Bands: TQuickRepBands;	Объект *Bands* содержит логические свойства, которые после установки в них значений *True* включают в отчет: *HasColumnHeader* – заголовки столбцов; *HasDetail* – детальную

	информацию; *HasPageFooter* – подвал страницы; *HasPageHeader* – заголовок страницы; *HasSummary* – подвал отчета; *HasTitle* – заголовок отчета.
property Dataset: TDataSet;	Указывает набор данных на основе которого создается отчет. Если нужно вывести связанную информацию из нескольких таблиц БД, ее объединяют в одном НД при помощи компонента *TQuery*. Информацию из нескольких связанных НД можно включать в отчет, если эти НД связаны в приложении отношением главный-подчиненный. В этом случае в качестве НД отчета указывается главный набор, а ссылка на соответствующие подчиненные наборы осуществляется в компонентах *TQRSubDetail*. Если в отчет нужно включить информацию из несвязанных НД, применяется композитный отчет, то есть отчет, составленный из группы других отчетов.
property Frame: TQRFrame;	Определяет параметры рамки отчета: *Color* – цвет линий; *DrawBottom* – наличие линии снизу; *DrawLeft* – наличие линии слева; *DrawRight* – наличие линии справа; *DrawTop* – наличие линии сверху; *Style* – стиль линии (сплошная, пунктирная и т.п.); *Width* – толщина линии в пикселях.
property Options: TQuickReportOptions;	Содержит множество из следующих логических значений: *HasFirstHeader* – разрешает печатать заголовок первой страницы; *HasLastFooter* – разрешает печатать подвал последней страницы; *Compression* – разрешает сжимать отчет при выводе его в метафайл.
property Page: TQRPage;	Определяет параметры страницы отчета. Все подсвойства этого сложного

	свойства доступны в окне *Report Setting* (см. ниже группы *Page size* и *Margin* окна редактора свойств).
property PrintIfEmpty: boolean;	Разрешает/запрещает печатать отчет в том случае если он не содержит данных.
property ReportTitle: **String**;	Имя отчета (не его заголовок !). Используется для идентификации отчета в задании на сетевую печать, возвращается компонентом *QRSysData* при *Data = ReportTitle* и может использоваться для набора одного из нескольких доступных отчетов.
property ShowProgress: boolean;	Разрешает/запрещает показывать индикатор процесса печати отчета.
property SnapToGrid: boolean;	Если содержит *True*, размещаемые в отчете компоненты привязываются к сетке отчета.
type TQRUnits = (Inches, MM, Pixels, Native, Characters); **property** Units: TQRUnits;	Определяет единицы измерения расстояний в отчете: *Inches* – дюймы; *MM* – миллиметры; *Pixels* – пиксели; *Native* – внутренние единицы *TQuickRep* (равны 0,1 мм); *Characters* – символы текста.
property Zoom: Integer;	Определяет масштаб отображения отчета (в процентах от его размеров на листе бумаги) на этапе разработки. Может иметь значение в диапазоне 1..300. Значение свойства не учитывается при печати отчета или в режиме его предварительного показа.

Многие свойства отчета можно установить на этапе конструирования с помощью редактора свойств – вызовите локальное меню компонента *TQuickRep* и выберите опцию *Report Settings*.



Окно установки параметров отчета.

Группа *Paper size* задает характеристики страницы: ее формат (*A4 210 x 270 mm*), ширину (*Width*), длину (*Length*) и направление печати – вдоль короткой стороны листа (*Portait*) или вдоль длинной (*Landscape*).

Группа элементов *Margin* указывает поля отчета: сверху (*Top*), снизу (*Bottom*), слева (*Left*), справа (*Right*), а также количество колонок (*Number of columns*) и расстояние между ними (*Column Space*).

С помощью элементов группы *Other* можно задать шрифт (*Font*), его высоту (*Size*) и используемые единицы измерения длины (*Units*).

Группа *Page frame* определяет свойства рамки: наличие линии сверху (*Top*), снизу (*Bottom*), слева (*Left*), справа (*Right*), цвет линий (*Color*) и их толщину (*Width*).

Группа *Bands* определяет наличие полос заголовков и подвалов (*Page header* – заголовок страницы; *Title* – заголовок отчета; *Column header* – заголовок колонок; *Detail band* – полоса для детальной информации; *Page footer* – подвал страницы; *Summary* – подвал отчета), а также высоту соответствующей полосы (строка *Length* справа от переключателя выбора). После выбора типа и высоты

полосы она появляется в отчете, если окно закрыто кнопкой *OK* или была нажата кнопка *Applay*. Элементы *Print first page header* и *Print last page footer* управляют соответственно печатью заголовка на первой странице и подвала на его последней странице.

Методы

Метод	**Назначение**
procedure NewColumn**;**	Реализует вывод информации в следующей колонке отчета, а если определена единственная колонка, – в его следующей странице.
procedure NewPage;	Реализует вывод информации в следующей странице отчета.
procedure Prepare;	Готовит отчет для вывода в файл (см. ниже примечание 1).
procedure Preview;	Выводит стандартное окно предварительного просмотра (см. ниже примечание 2).
procedure Print;	Печатает отчет на принтере.
procedure PrintBackGround;	Инициирует печать отчета в фоновом режиме (в отдельном потоке команд). После завершения печати вызывается обработчик события *OnAfterPrint*.
procedure PrinterSetup;	Вызывает стандартное окно установки параметров принтера.

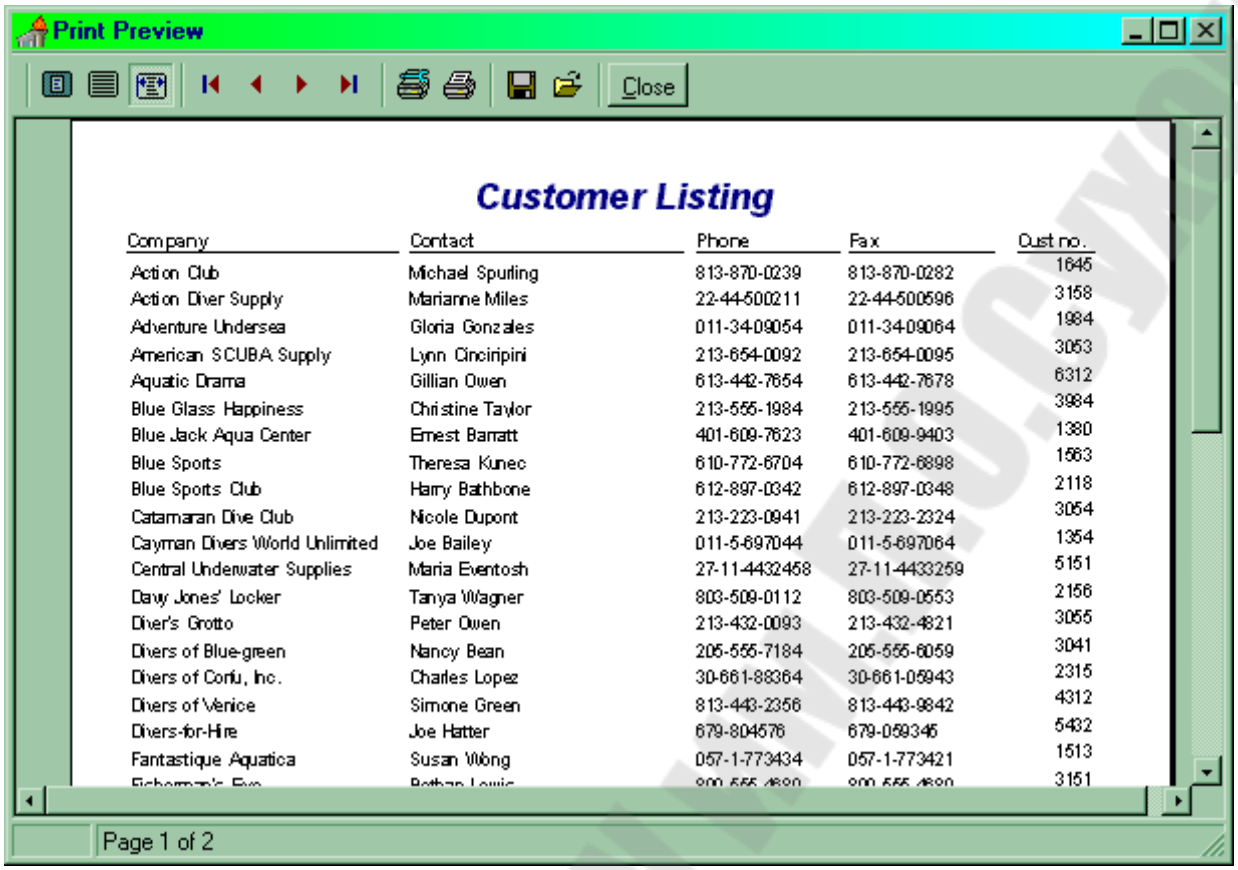
Примечание 1.

Для вывода отчета в файл нужно сначала подготовить его с помощью обращения к методу *Prepare*, затем сохранить в файле методом *Save* объекта *TQuickRep.QRPrinter*, после чего уничтожить этот объект и поместить *NIL* в свойстве *TQuickRep.QRPrinter*:

```
MyReport.Prepare;
MyReport.QRPrinter.Save('REport.QRP');
MyReport.QRPrinter.Free;
MyReport.QRPrinter := NIL;
```

Примечание 2.

Стандартное окно предварительного просмотра показано на рис. 4.



Окно предварительного просмотра отчета.

Чтобы на этапе конструирования просмотреть в окне предварительного просмотра содержимое отчета в том виде, как он будет выводиться на печать, нужно выбрать опцию Preview во вспомогательном меню компонента QuickRep. Следует заметить, что при этом не будут видны некоторые данные, например значения вычисляемых полей. Они будут выводиться только во время выполнения.

Назначение инструментальных кнопок окна:

Масштабирует отчет так, чтобы его страница полностью показывалась в окне.

Отображает отчет в масштабе 1:1.

Масштабирует отчет так, чтобы ширина страницы отчета соответствовала ширине окна.

Показывает первую (последнюю) страницу отчета.

Показывает предыдущую (следующую) страницу отчета.

Вызывает стандартное окно настройки принтера (печатает отчет).

Сохраняет отчет в файле (загружает отчет из файла).

События

Событие	Назначение
property AfterPreview : TQRAfterPreviewPrint;	Возникает в момент закрытия окна предварительного просмотра отчета.
property AfterPrint: TQRAfterPrintEvent;	Наступает после печати отчета или его подготовки к печати.
property BeforePrint: TQRBeforePrintEvent;	Наступает в момент начала генерации отчета (до выдачи окна предварительного просмотра отчета или до его печати).
property OnEndPage: **procedure** (Sender: TObject);	Возникает в момент подготовки к генерации последней страницы отчета.
property OnNeedData: **procedure** (Sender: TObject; **var** MoreData: boolean);	Используется при создании отчета по данным, которые берутся не из НД, а из текстового файла, списка строк, массива и т.п. В параметре *MoreData* обработчик должен вернуть *True*, если источник данных еще не исчерпан.
property OnPreview: **procedure** (Sender: TObject);	Используется для связывания с отчетом нестандартного окна просмотра (см. ниже).
property OnStartPage: **procedure** (Sender: TObject);	Возникает в момент подготовки к генерации первой страницы отчета.

С помощью компонента *QRPreview* программист может создать нестандартное окно предварительного просмотра. Для связи с отчетом используется событие *OnPreview* по следующей схеме:

```
Procedure RepForm.MyREportOnPreviewEvent(Sender: TObject);
begin
  MyPrevForm.QRPreview1.QRPrinter := TQRPrinter(Sender);
  MyPreviewForm.Show;
end;
```

Чтобы явное приведение типа TQRPrinter(Sender) стало возможным, необходима ссылка на модуль *QRPrntr* в предложении *Uses* соответствующего модуля (в примере – модуля RepForm).

Chapter 12 Компонент TQRBand

Компоненты *TQRBand* являются основными частями отчета и используются для размещения на них отображающих компонентов, таких как *TQRLabel*, *TQRDBText*, *TQRImage* и т.п.

Свойства компонента:

Свойство	**Назначение**
property AlignToBottom: boolean;	Если имеет значение *True* полоса печатается непосредственно над подвалом страницы вместо обычного расположения справа/снизу от предыдущей полосы.
type TQRBandType = (rbTitle, rbPageHeader, rbDetail, rbPageFooter, rbSummary, rbGroupHeader, rbGroupFooter, rbSubDetail, rbColumnHeader); **property** BandType: TQRBandType;	Указывает назначение полосы: *rbTitle* – содержит заголовок отчета; *rbPageHeader* – содержит заголовок страницы (на первой странице печатается под *rbTitle*); *rbDetaul* – содержит информацию из НД; выводится всякий раз при переходе на новую запись НД; эта полоса повторяется для всех записей *DataSet*, начиная с первой записи и заканчивая последней; позицирование на первую запись и последовательный их перебор осуществляется компонентом *TQuickRep* автоматически; *rbPageFooter* – содержит подвал страницы; выводится в конце каждой страницы отчета после всех других полос; *rbSummary* – подвал отчета; выводится на последней странице отчета после всей иной информации, но перед подвалом последней страницы; *rbGroupHeader* – содержит заголовок группы; применяется при группировках информации в отчете и выводится всякий раз при выводе новой группы; *rbGroupFooter* – содержит подвал группы; выводится всякий раз при окончании вывода группы, после всех данных группы; *rbSubDetail* – содержит детальную информацию из подчиненного НД при выводе в отчете информации из двух или более наборов

	данных, связанных в приложении как главный-подчиненный; этот тип назначается полосе автоматически при размещении на форме компонента *TQRSubDetail*; *rbColumnHeader* – содержит заголовки столбцов; размещается на каждой странице отчета после заголовка страницы.
property Enabled: boolean;	Разрешает/запрещает печать полосы.
property ForceNewColumn: boolean;	Если содержит True, полоса печатается в следующей колонке.
property ForceNewPage: boolean;	Если содержит *True,* полоса печатается на новой странице.
property HasChild: boolean;	Если содержит *True*, полоса имеет дочернюю полосу *TChildBand*. Установка *True* в это свойство автоматически создает в отчете дочернюю полосу.

События

property AfterPrint: TQRAfterPrintEvent;

и

property BeforePrint: TQRBeforePrintEvent;

наступают соответственно до и после печати полосы. Метод

function AddPrintable(PrintableClass: TQRNewComponentClass): TQRPrintable;

используется для вставки в полосу отображающего компонента в процессе прогона программы. Он автоматически устанавливает между полосами отношение собственности. Два следующих фрагмента выполняют одинаковую работу:

```
with DetailBand1.AddPrintable(TQRLabel) do
begin
  Size.Left := 20;
  Size.Top := 5;
  Caption := ‘Новая полоса’;
end;
var
  aLabel : TQRLabel;
begin
```

```
    aLabel := TQRLabel.Create(ReportForm);
    aLabel.Parent := DetailBand1;
    with aLabel do
    begin
      Size.Left := 20;
      Size.Top := 5;
      Caption := 'Новая полоса';
    end;
  end;
```

Chapter 13 Создание простейшего отчета

Компоненты *TQuickRep* и *TQRBand* являются минимально достаточными для создания простейшего отчета, не содержащего внутри себя группировок информации.

Пусть имеется таблица БД *Rashod.DB*, содержащая сведения об отпуске материалов со склада. В состав ТБД входят поля

- *N_RASH* – уникальный номер события отпуска товара;
- *DEN* – номер дня;
- *MES* – номер месяца;
- *GOD* – номер года;
- *TOVAR* – наименование отпущенного товара;
- *POKUP* – наименование покупателя;
- *KOLVO* – количество единиц отпущенного товара.

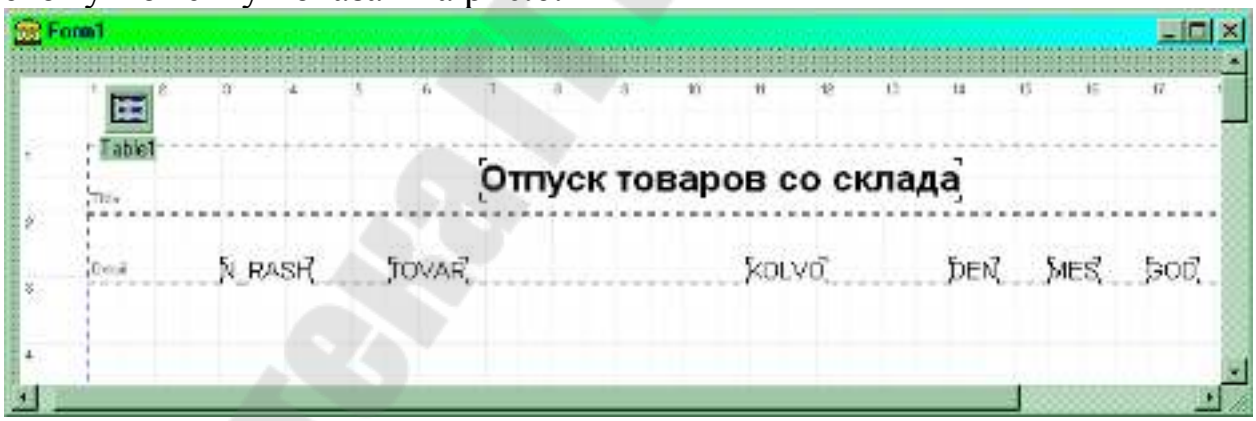
Заметим, что дата отпуска товара хранится в разбивке на день, год и месяц. Сделано так специально, с целью показать, как в отчетах используются выражения и вычисляемые поля.

Создадим простейший отчет, состоящий из заголовка и сведений об отпуске товара. В отчет включаются все факты отпуска товара. Сортировка производится по номеру события отпуска товара. Для этого разместим на форме компонент *TTable*, свяжем его с таблицей *Rashod.DB* и откроем (*Active = True*). Разместим на форме компонент *TQuickRep*. Поместим в его свойство *DataSet* значение *Table1*, назначив таким образом отчету НД, записи которого будут выводиться в отчете. Добавим в отчет компонент *TQRBand*. В его свойство *BandType* компонента *QRBand1* по умолчанию будет установлено значение *rbTitle*, то есть компонент *QRBand1* определяет заголовок отчета Разместим на *QRBand1* компонент *TQRLabel*. Установим в свойство *Caption* этого компонента значение Отпуск товаров со склада и выберем в свойстве *Font* жирный наклонный

шрифт высотой 16 пунктов. Вид формы отчета к этому моменту показан на рис.5.

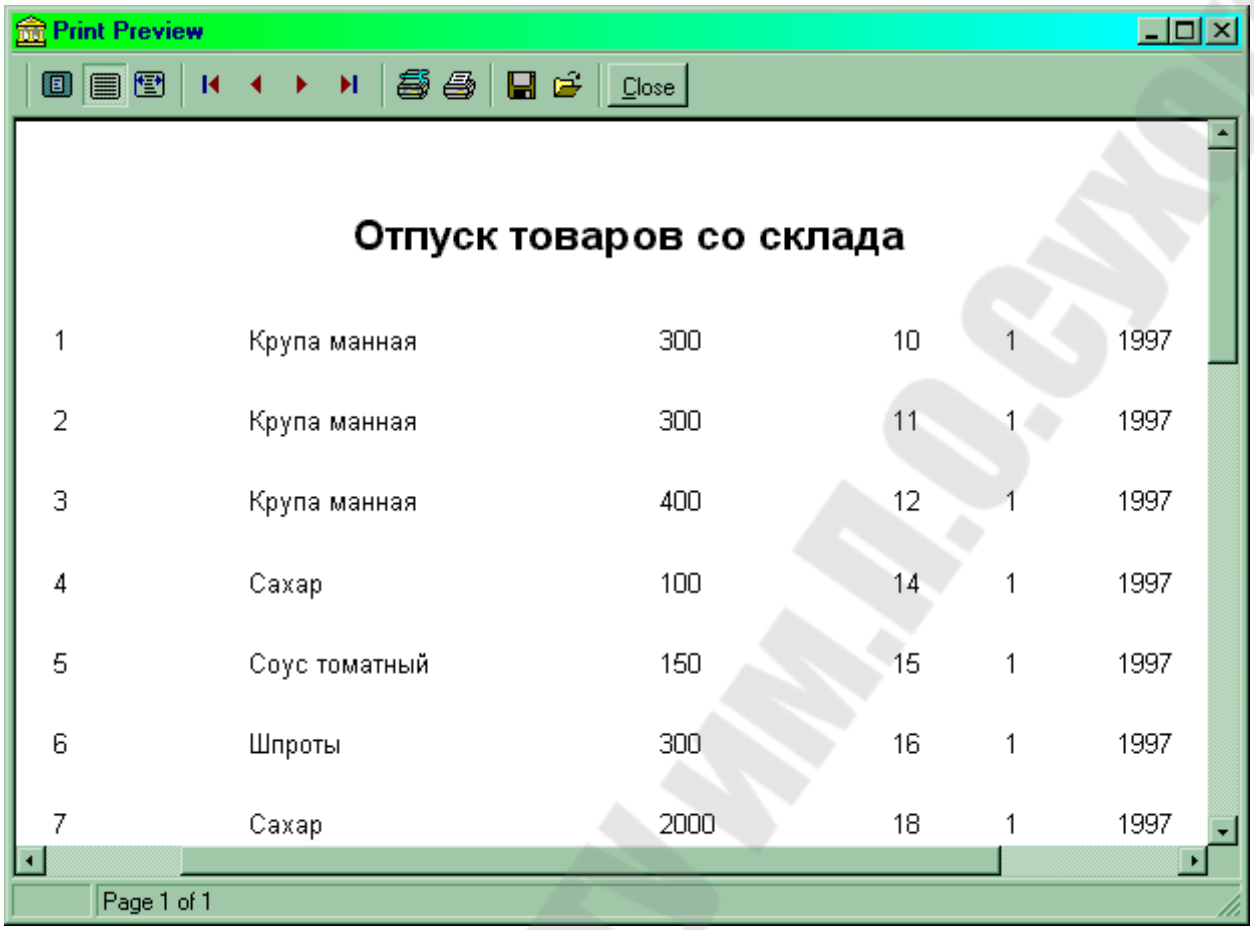
В отчете определен только его заголовок.

Теперь разместим в отчете данные, соответствующие текущей записи таблицы *Rashod.* Для этого поместим в отчет новый компонент *TQRBand* (имя *QRBand2*) и установим в его свойство *BandType* значение *rbDetail*. Затем разместим на полосе *QRBand2* шесть компонентов *TQRDBText*. Свяжем эти компоненты с полями НД – *N_RASH*, *TOVAR*, *KOLVO*, *DEN*, *MES*, *GOD*. Для этого в свойство *DataSet* каждого компонента *QRDBText* установим значение *Table1*, а в свойство *DataField* – имя соответствующего поля. Вид отчета к этому моменту показан на рис.6.



Отчет с заголовком и группой детальной информации.

Для просмотра получившегося отчета щелкнем по нему правой кнопкой мыши и из всплывающего меню выберем элемент *Preview.* Окно предварительного просмотра отчета показано на рис. 7.



Содержимое отчета в окне предварительного просмотра.

Чтобы окно предварительного просмотра открывалось при активизации формы, создадим такой обработчик события *OnActivate* формы:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  QuickRep1.Preview;
end;
```

а чтобы после выхода из окна предварительного просмотра закрывалась бы форма, на которой расположен текст, используем такой обработчик события *AfterPreview*:

```
procedure TForm1.QuickRep1AfterPreview(Sender: TObject);
begin
  Form1.Close;
end;
```

Использование компонента TQREXPR

Из рис.7 видно, что в простейшем отчете выводится дата, составленная из трех полей – *DEN, MES, GOD*. Объединим значения

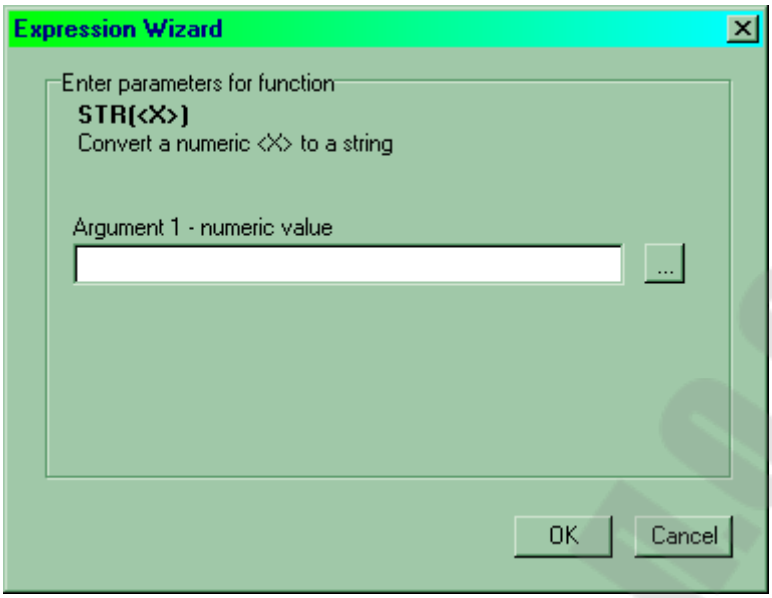
из этих полей в одно значение, являющееся результатом вычисления выражения. Выражение в отчетах формируется при помощи компонента *TQRExpr*. Удалим из компонента *QRBand2* компоненты *QRDBText4*, *QRDBText5* и *QRDBText6*, связанные с полями *DEN*, *MES*, *GOD*. Вместо них разместим в отчете компонент *TQRExpr* (имя *QRExpr1*).

Выражения в *TQRExpr* формируются с помощью специального редактора, который вызывается в окне инспектора объектов кнопкой в поле данных свойства Expression этого компонента (рис.8).

Окно редактора формул компонента TQRExpr.

В поле *Enter expression* можно ввести или отредактировать выражение, которое обычно состоит из имен полей НД, преобразующих функций и переменных, связанных операциями отношения. Имена полей НД добавляются в текущее положение курсора (поле *Enter expression*) с помощью вспомогательного окна, связанного с кнопкой *Function*, а переменные – с кнопкой *Variable*.

Нажмите кнопку *Function*, в левом окне выберите категорию *Other* (другие) и функцию *STR* в правом окне – эта функция преобразует числовое значение в строковое. Нажмите *Continue*, чтобы перейти к вводу параметров (рис.9). Надпись над строкой ввода окна *Expression Wizard* напоминает о том, что выбранная нами функция имеет один числовой параметр.



Формирование части выражения.

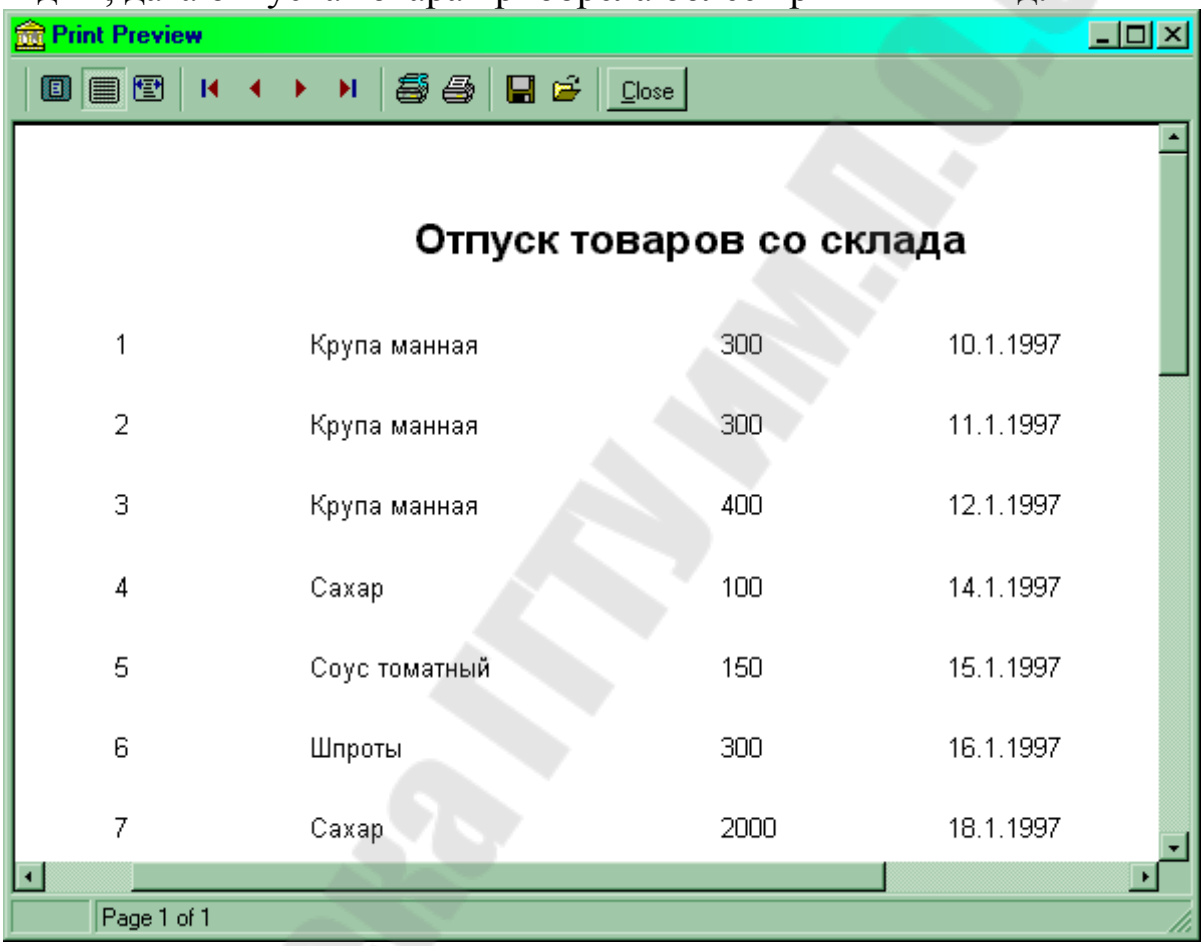
Для его ввода нажмите кнопку справа от строки ввода – на экране вновь появится начальное окно редактора формул. Поскольку мы хотим преобразовать в строку номер дня, нажмите кнопку Database field и выберите поле *DEN* в списке полей таблицы *Table1*. Нажмите *OK*, чтобы завершить ввод параметра. В поле *Enter* expression будет сформирована часть формулы – STR(Table1.DEN). На панели I*nsert at cursor position* нажмем кнопку «+» и вручную введем разделитель ‘.’ (рис.10).

Создание части формулы выражения.

Продолжите формировать выражение так, чтобы в конце концов оно приобрело такой вид:

STR(Table1.DEN) + ‘.’ + STR(Table1.MES) + ‘.’ +
STR(Table1.GOD)

(возможно проще ввести его вручную). Затем нажмите кнопку *OK*, чтобы закрыть окно редактора формул. С помощью Инспектора объектов установите в свойство *AutoSize* компонента *QRExpr1* значение *False*, измените размеры компонента так, чтобы он мог отображать примерно 10 символов, и установите выравнивание вправо (свойство *Alignment* = *taRightJustify*). Запустите режим предварительного просмотра содержимого отчета (рис.11). Как видим, дата отпуска товара приобрела более привычный вид.



Результат вычисления выражения появился в отчете.

Замечание.

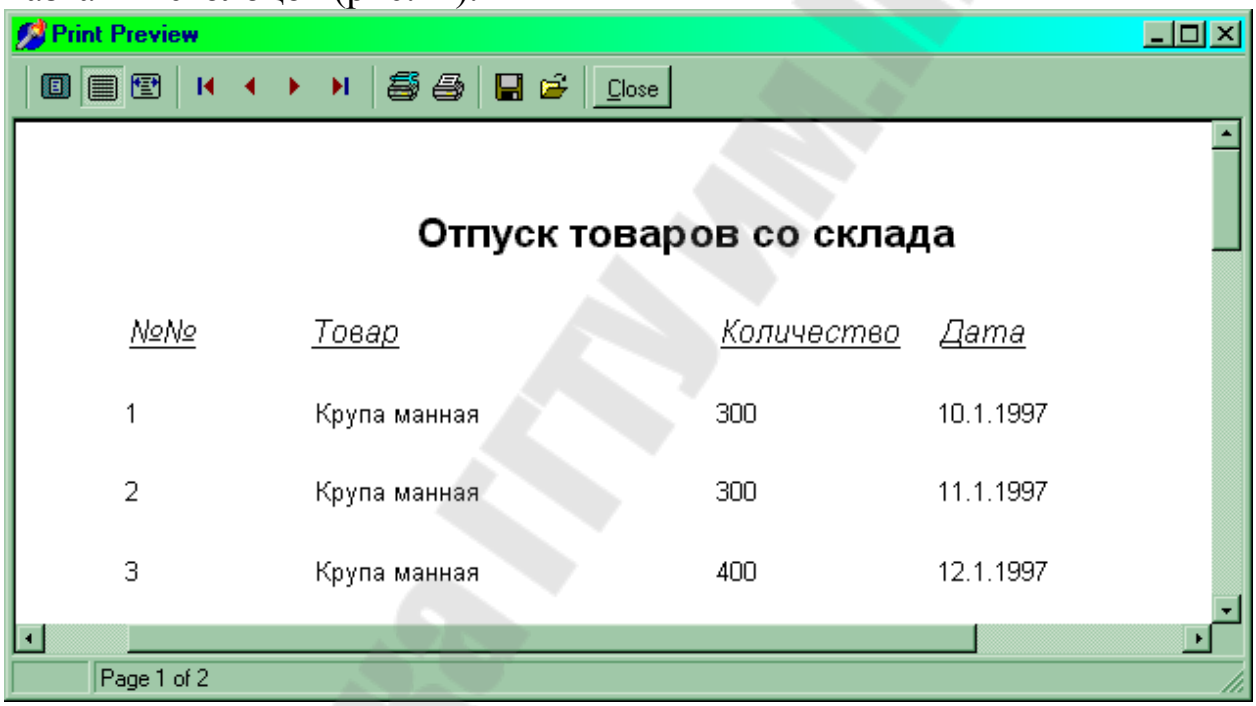
Другим способом составления значения даты из трех полей могло бы быть создание вычисляемого поля (например, *SumData*) и определение алгоритма вычисления его значения в таком обработчике события *OnCalcFields*:

```
procedure TForm1.TableCalcFields(DataSet: TDataSet);
begin
  Table1SumData.Value := Table1DEN.AsString + '.' +
  Table1MES.AsString + '.' + Table1GOD.AsString;
end;
```

Использование TQRBand для представления заголовков столбцов

Компонент *TQRBand*, у которого в свойство *BandType* установлено значение *rbColumnHeader*, используется для размещения заголовков столбцов. Собственно заголовки столбцов формируются при помощи компонентов *TQRLabel*.

В рассмотренном в предыдущих разделах отчете разместим компонент *TQRBand* (имя *QRBand3*) и установим в свойства *Caption* этих компонентов соответственно значения №№, Товар, Количество, Дата. В свойствах *Font* компонентов выберем наклонный и подчеркнутый шрифт. Вызовем окно предварительного просмотра отчета – для каждой страницы отчета теперь будут выводиться названия столбцов (рис.12).



В отчете появились заголовки столбцов.

Использование TQRBand для показа заголовка и подвала страницы.

Компонент TQRBand, у которого в свойство BandType установлено значение rbPageHeader, используется для показа заголовка страницы, а если это свойство установлено в rbPageFooter, – для показа подвала страницы. Заголовок выводится в начале каждой страницы, а подвал – в ее конце. Информация в заголовке и подвале страницы может формироваться на основе статического текста (компоненты TQRLabel), значений полей (компоненты TQRDBText) и результатов вычислений выражений (компоненты TQRExpr).

Вернувшись к предыдущему примеру, разместим в отчете компонент TQRBand (имя QRBand4) и установим в его свойство BandType значение rbPageHeader. Не будем размещать в заголовке никакого текста, просто отчеркнем линию вверху страницы. Для этого установим в свойство компонента страницы *Frame.DrawTop* значение *True*, что обеспечивает вывод линии по верхнему краю области, занимаемой компонентом. Аналогичным образом определим в отчете компонент подвала страницы (имя *QRBand5*) и установим в его свойство *Frame.DrawBottom* значение *True*, что обеспечивает вывод линии по нижнему краю области, занимаемой компонентом.

Войдя в режим предварительного просмотра, увидим, что вверху и внизу каждой страницы отчета выводятся линии.

Использование компонента TQRSysData

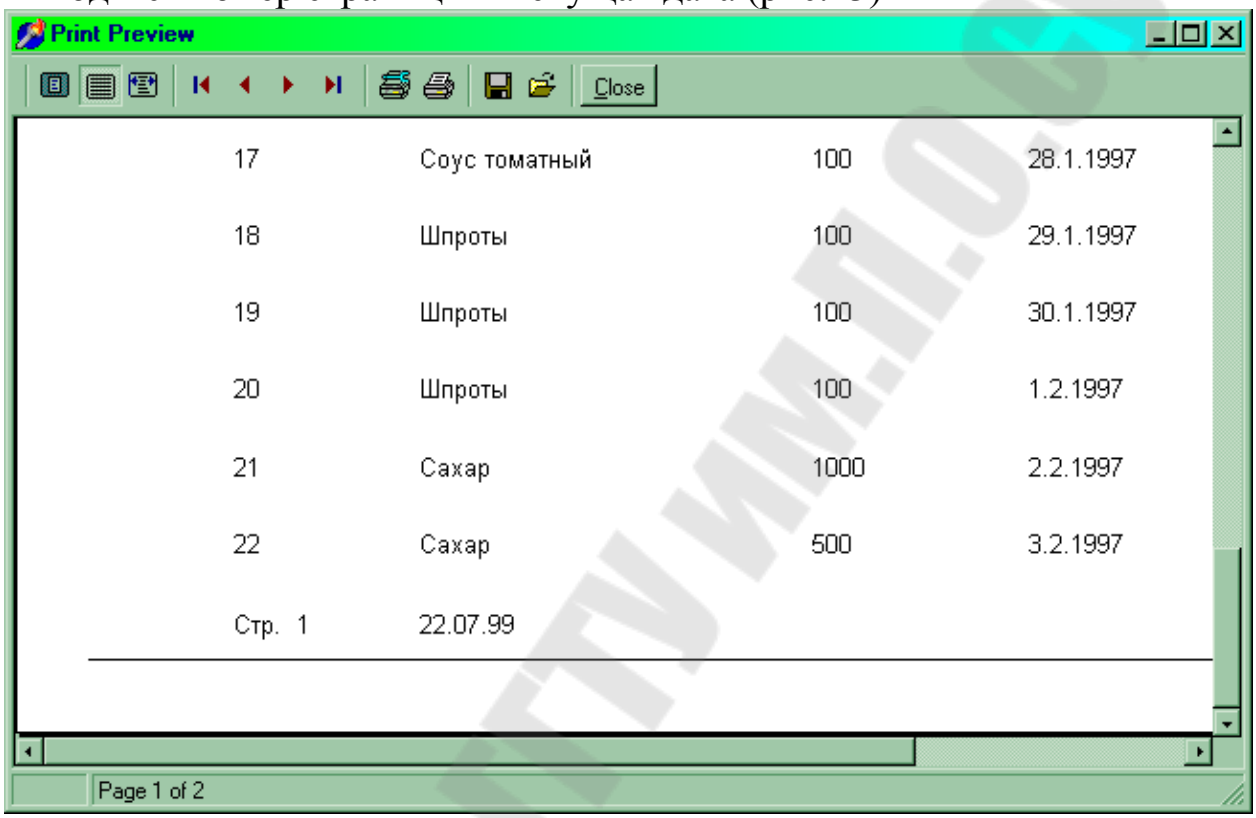
Компонент *TQRSysData* используется для показа вспомогательной и системной информации. Вид показываемой информации определяется свойством

property Data: TQRSysDataType;

Ниже указаны возможные значения этого свойства.

Значение	**Что выводится**
qrsColumnNo	Номер текущей колонки отчета (для одноколоночного отчета всегда 1).
qrsDate	Текущая дата.
qrsDateTime	Текущие дата и время.
qrsDetailCount	Количество записей в НД, а при использовании нескольких НД – количество записей в главном наборе. Для случая, когда НД представлен компонентом *TQuery*, эта возможность может быть недоступной, что связано с характером работы компонента *TQuery*, который возвращает столько записей, сколько необходимо для использования в текущий момент, а остальные предоставляет по мере надобности.
qrsDetailNo	Номер текущей записи в НД.
qrsPageNumber	Номер текущей страницы отчета.
qrsPageCount	Общее количество страниц отчета.
qrsReportTitle	Заголовок отчета.
qrsTime	Текущее время

Разместим в компоненте *QRBand5* подвала отчета два компонента *TQRSysData*. В свойство *Data* первого из них установим значение *qrsDate*, второго – *qrsPageNumber*. В режиме предварительного просмотра увидим, что теперь в подвале страницы выводятся номер страницы и текущая дата (рис.13)



Показ номера страницы и текущей даты в подвале страницы.

Группировки данных

Для группировок информации используется компонент *TQRGroup*. Его свойство Expression указывает некоторое выражение, которое используется для группировки, иными словами, в группу входят записи, удовлетворяющие условию этого выражения. При смене выражения происходит смена группы.

Для каждой группы выводятся ее заголовок и подвал. В качестве заголовка группы используется компонент *TQRBand* со значением свойства *BandType*, равным *rbColumnHeader*, а в качестве подвала – со значением *rbGroupFooter*. Свойство *FooterBand* компонента *TQRGroup* должно содержать ссылку на компонент подвала группы. В заголовке группы, как правило, выводится группирующее выражение, а в подвале группы – агрегированная информация: суммарные, средние и т.п. значения по группе в целом.

Пример.

Построим отчет о расходе товара со склада, в котором информация группируется по наименованию товара. Для этого определим набор данных отчета (компонент *TTable*, имя *Table1*). Установим у НД текущим индекс по полю *TOVAR* (в свойстве *FieldIndexNames* или *IndexName*). Разместим в отчете:

- заголовок отчета – компонент *TQRBand* с именем *QRBand1*, свойство *BandType=rbTitle*;
- заголовок столбцов – компонент *TQRBand* с именем *QRBand2*, свойство *BandType=rbColumnHeader*;
- группу – компонент *TQRGroup* с именем *QRGroup1*;
- область детальной информации – *TQRBand* с именем *QRBand3*, свойство *BandType=rbDetail*;
- подвал группы – *TQRBand* с именем *QRBand4*, свойство *BandType=rbGroupFooter*;

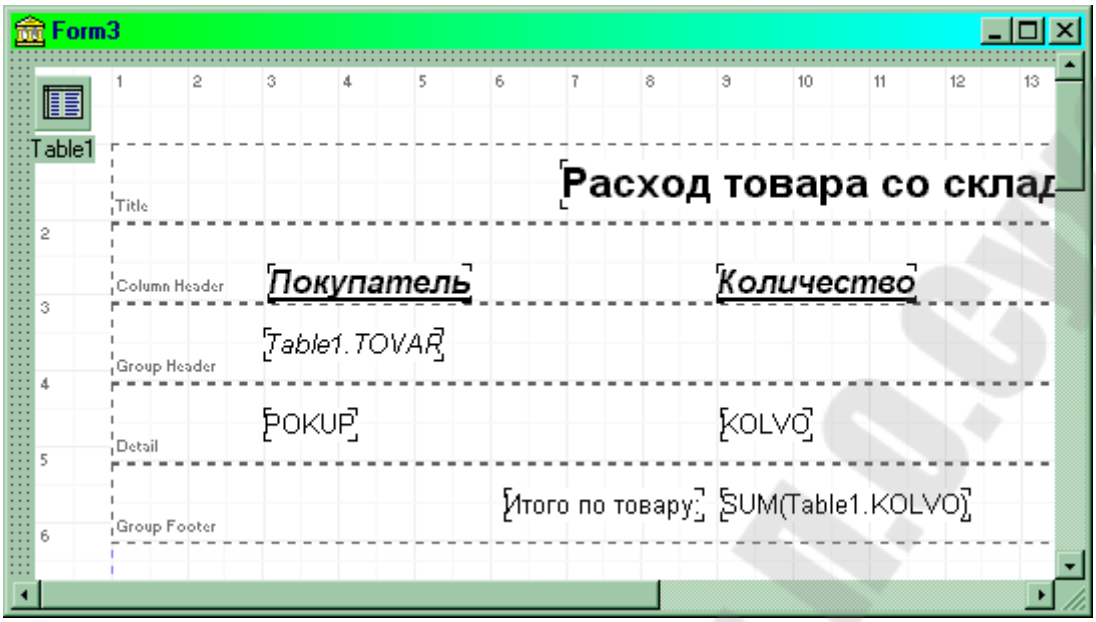
В компоненте *QRGroup1* установим:

- в свойство *FooterBand* значение *QRBand4*;
- в свойство *Expression* значение Table1.TOVAR, которое является формулой и строится в редакторе формул.

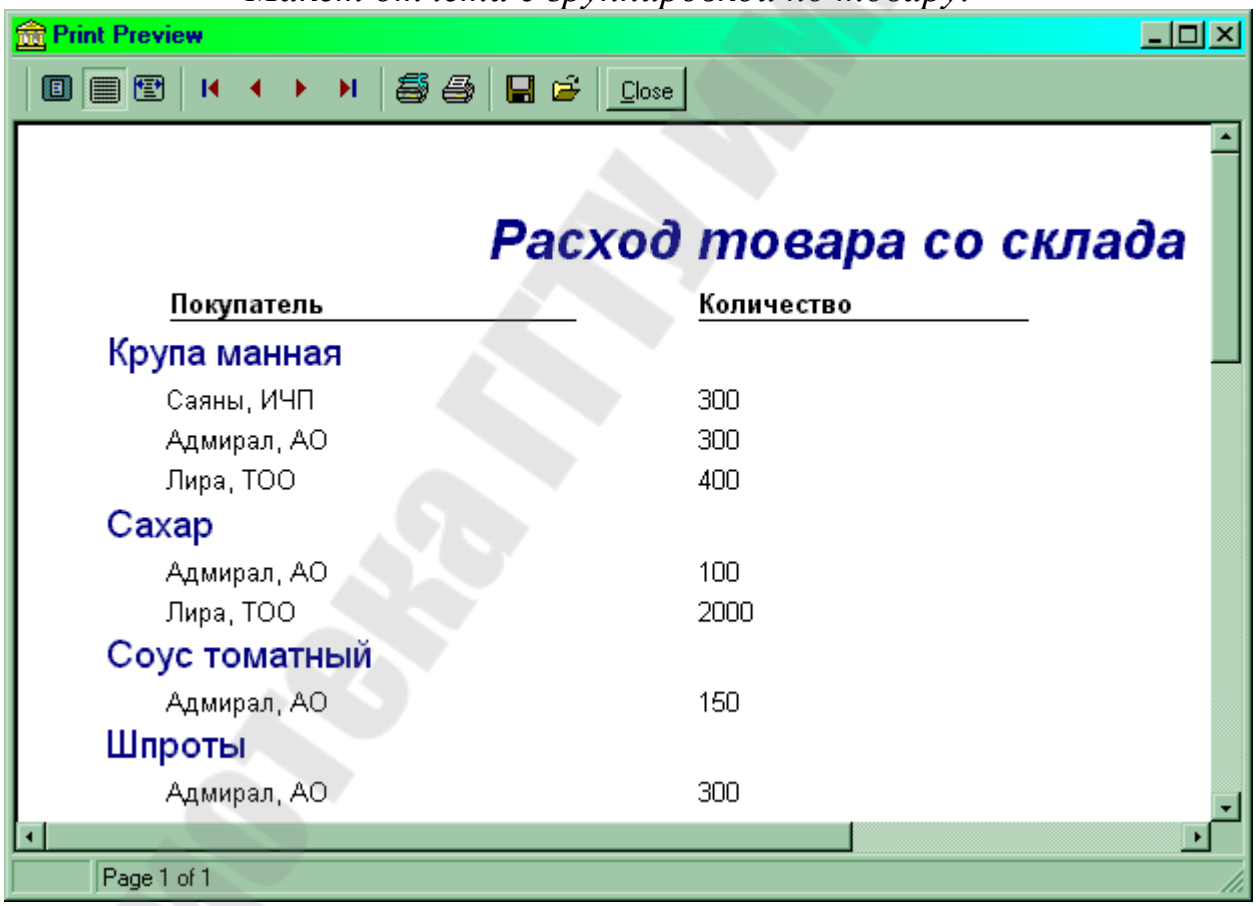
Поскольку свойство *Expression* не визуализирует значения выражения, необходимо разместить в группе компонент *TQRExpr* (имя *QRExpr1*) и определить значение его свойства *Expression* так, чтобы оно содержало Table1.TOVAR.

В компоненте подвала группы *QRBand4* будем подсчитывать сумму по полю *KOLVO* (сумму отпущенного конкретного товара). Для этого разместим в подвале группы компонент *TQRExpr* (имя *QRExpr2*) и определить значение его свойства *Expression* так, чтобы оно содержало формулу SUM(Table1.TOVAR).

В группе детальной информации разместим компоненты *TQRDBText*, связанные с полями *Pokup* и *Kolvo*. Заполним области отчета статическим текстом, как это показано на рис.14.



Макет отчета с группировкой по товару.



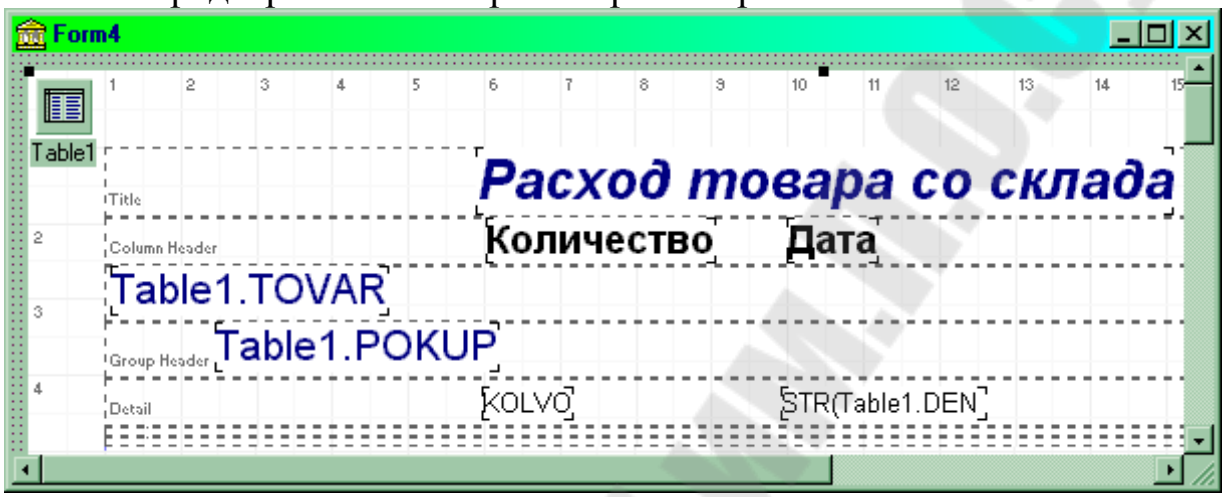
Отчет с группировкой по товару в окне предварительного прсмотра.

Множественная группировка данных

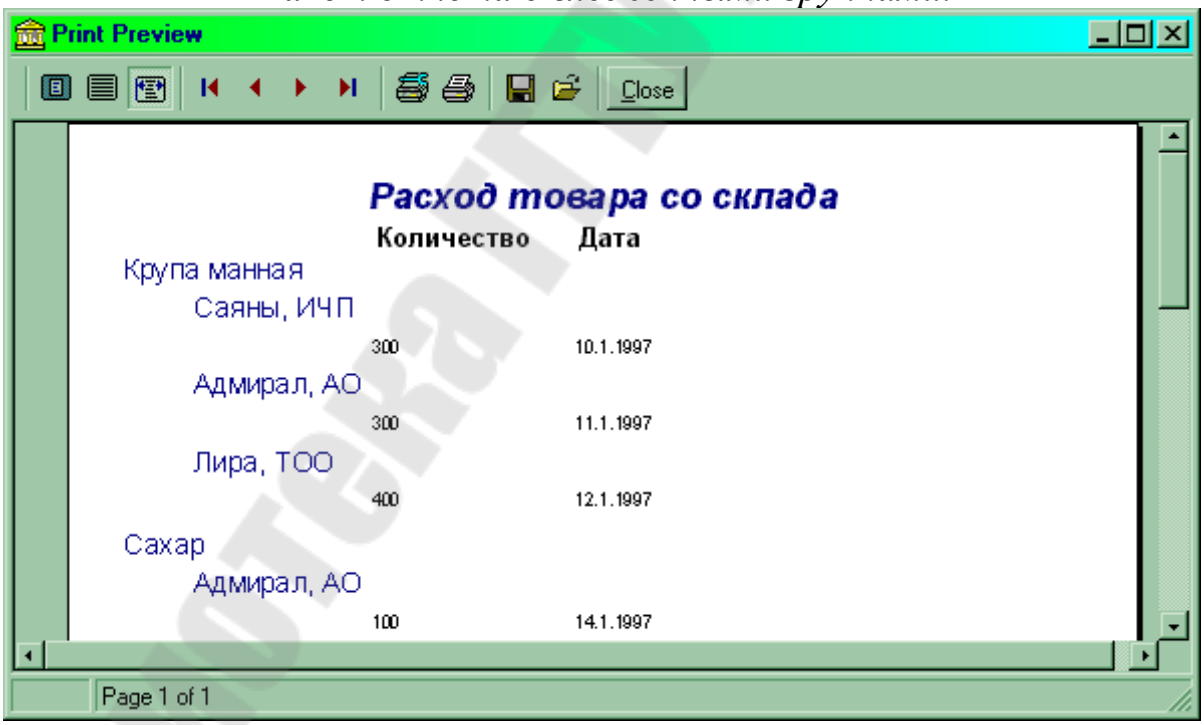
Часто внутри группы должны содержаться другие группы, например, по названию товара и внутри каждой группы – по

покупателям. В этом случае внутри одной группы определяют другую посредством дополнительных компонентов *TQRGroup*.

Пусть требуется представить в отчете сведения о расходе товаров со склада группируя данные по товарам, а внутри группы – по покупателям. Установим текущий индекс по полям *TOVAR*, *POKUP*. Общий вид отчета на этапе разработки приводится на рис.17, а в окне предварительного просмотра – на рис.18.



Макет отчета с вложенными группами.



Отчет с вложенными группами.

Построение отчета главный-детальный

Если необходимо построить отчет на основе более чем одной ТБД, можно поступить двумя способами:

1. с помощью компонента *TQuery* произвести соединение данных из нескольких таблиц БД в один НД, после чего определить в отчете нужные группировки;
2. создать в приложении по одному НД на каждую таблицу, соединить эти наборы между собой связью главный-детальный (используя свойства *MasterSource*, *MasterFields* набора данных) и применить в отчете компонент (или несколько компонентов) *TQRSubDetail* для вывода информации из детального НД (или группы детальных НД); для вывода информации из главного НД, как и в обычных отчетах, применяется компонент *TQRBand*, у которого в свойстве *BandType* установлено значение *rbDetail*.

Построение отчета для первого случая осуществляется аналогично тому, как это описано выше. Построение отчета для второго случая имеет некоторые отличительные особенности. Рассмотрим второй способ.

Компонент *TQRSubDetail* предназначен для показа в отчете информации из детального НД. Его свойство

Property DataSet: TDataSet;

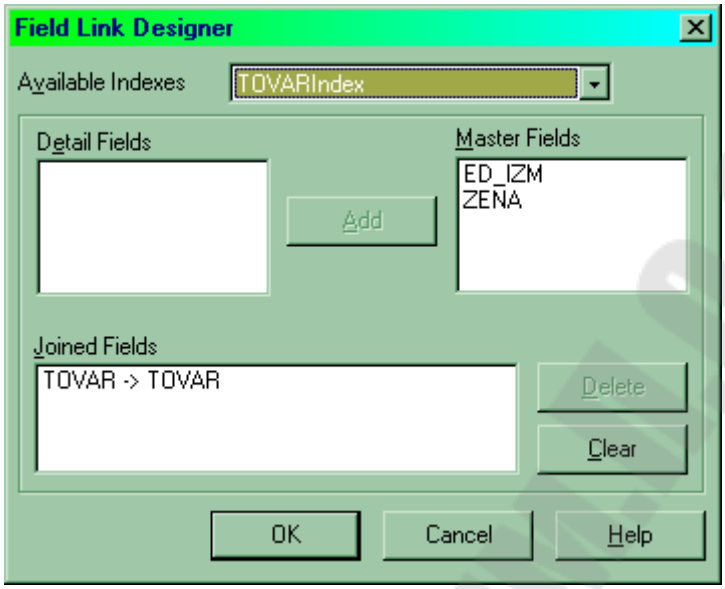
указывает имя детального НД, информация из которого будет выводиться в пространстве компонента *TQRSubDetail*. В остальном использование этого компонента аналогично использованию компонента *TQRBand*, у которого в свойство *BandType* установлено значение *rbDetail*.

Пусть имеется таблица БД TOVARY.DB, содержащая помимо прочих поле *TOVAR* (название товара). Пусть также имеется таблица БД RASHOD.DB, содержащая сведения об отпуске материалов со склада. В ее состав входят поля *N_RASH* (уникальный номер события отпуска товара), *DEN* (номер дня), *MES* (номер месяца), *GOD* (номер года), *TOVAR* (наименование отпущенного товара), *POKUP* (наименование покупателя) и *KOLVO* (количество единиц отпущенного товара).

Таблицы TOVARY.DB и RASHOD.DB находятся в отношении *один-ко-многим*, то есть одному товару может соответствовать более одного факта отпуска товара со склада.

Разместим на форме компонент *TTable* (им *TovaryTable*), ассоциированный с ТБД TOVARY.DB, и связанный с ним компонент *TDataSource* (имя *DS_TovaryTable*). Разместим также еще один компонент *TTable* (им *RashodTable*), ассоциированный с ТБД RASHOD.DB, и установим между НД связь главный-детальный. Для

этого установим в свойство *RashodTable.MasterSource* значение *DS_TovaryTable*, а в свойство *RashodTable.MasterFields* значение *TOVAR*.



Установка связи главный-детальный.

Заметим, что после установления связей НД и НД *RashodTable* текущим индексом должен быть индекс по полю *Tovar* (свойство *RashodTable.IndexFieldNames*).

Приступим к разработке отчета. Определим заголовок отчета – компонент *TQRBand* с именем *QRBand1*, в свойство *BandType* которого установлено значение *rbTitle*. Установим в качестве основного НД отчета *TovaryTable*, указав *QuickRep1.DataSet = TovaryTable*. Разместим в отчете компонент *TQRBand* с именем *QRBand2* и установим в его свойство *BandType* значение *rbDetail*. Этот компонент будет использоваться для отображения детальной информации из НД *TovaryTable*.

Разместим в отчете компонент *TQRSubTetail* (имя *QRSubDetail*). Установим в его свойство *DataSet* значение *RashodTable*, связав таким образом данный компонент с подчиненным НД. Разместим в области компонента *QRSubDetail* три компонента TQRDBText и свяжем их соответственно с полями *Pokup*, *Kolvo* и *D* НД *RashodTable* (поле *D* определено в НД *RashodTable* как вычисляемое по значениям полей *DEN*, *MES*, *GOD*). Разместим в области компонента *QRBand2* заголовки столбцов.

Вид формы отчета показан на рис.20.

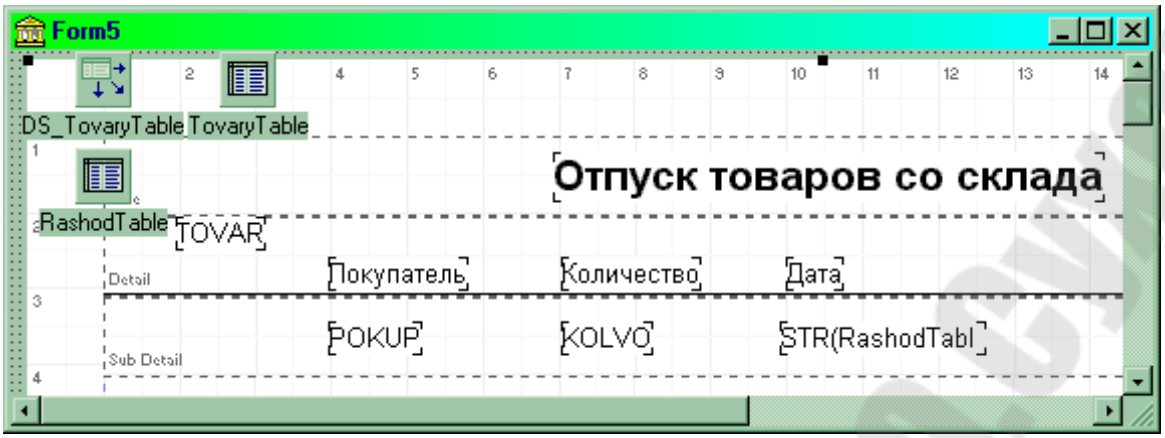


Рис. 20. Макет отчета, в котором показываются записи из связанных наборов данных.

В результирующем отчете (рис.21) для каждой записи НД *TovaryTable* выводятся подчиненные ей записи из НД *RashodTable*.

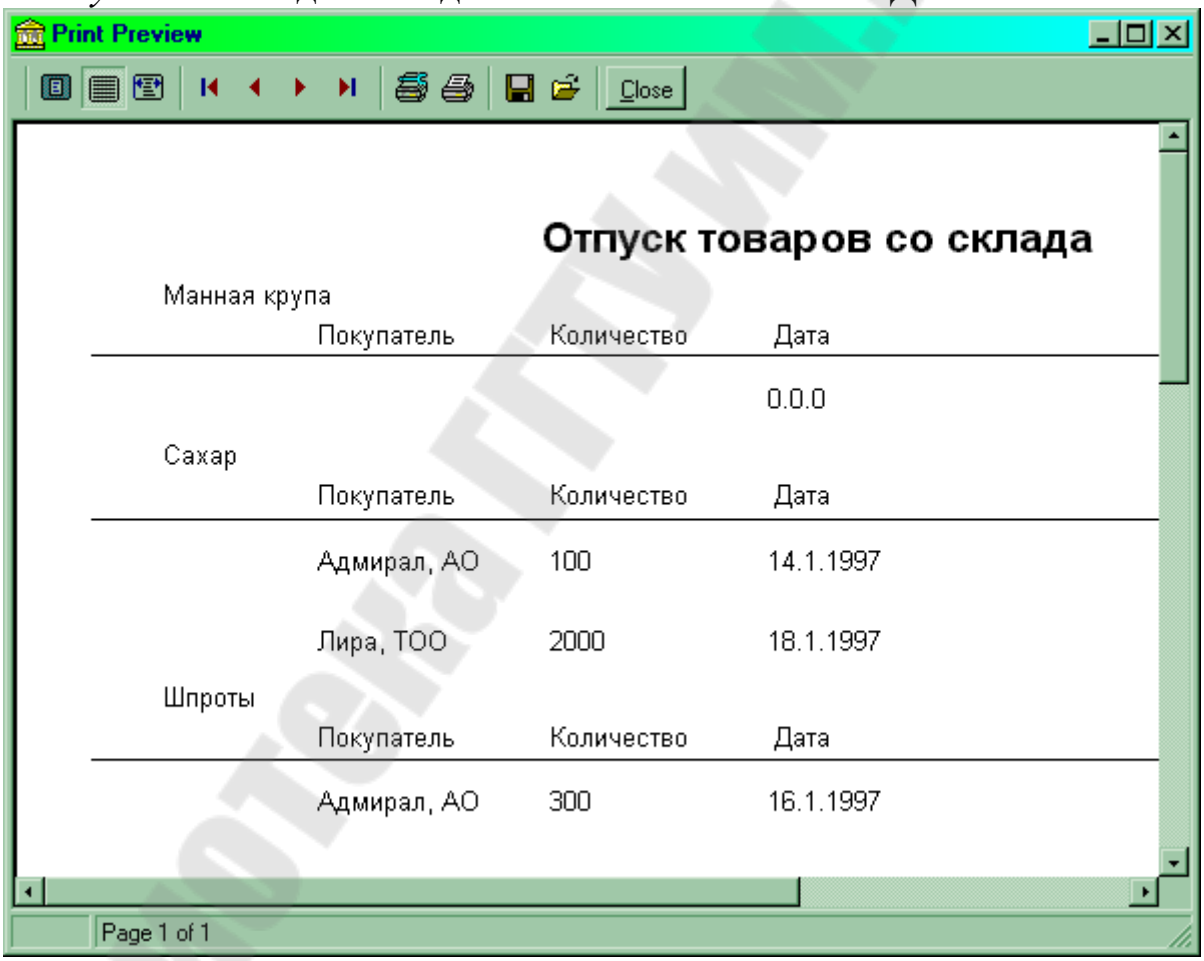


Рис. 21. Отчет, в котором показываются записи из связанных наборов данных.

Замечание.

Если необходимо определить заголовок и подвал для информации, группируемой в компоненте *TQRSubDetail*, следует воспользоваться свойством

property Bands: TQRSubDetailGroupBands;

этого компонента, которое имеет два логических подсвойства (*HasHeader* и *HasFooter*), указывающих на наличие или отсутствие соответственно заголовка и подвала.

Построение композитного отчета

Композитный (составной, сложный) отчет объединяет в себе несколько простых отчетов. При печати композитного отчета, входящие в его состав простые отчеты печатаются друг за другом.

Композитный отчет реализуется при помощи компонента *TQRCompositeReport*. В его обработчике события *OnAddReport* ранее определенные простые отчеты добавляются в списковое свойство Report. Например, так:

```
property
TCompositnyjOtchet.QRCompositeReport1AddReports(Sender:
TObject);
begin
  with QRCompositeReport1 do
  begin
    Reports.Add(ManyGroup.QuickRep1);
    Reports.Add(Prostoj.QuickRep1);
  end
end;
```

В этом примере композитный отчет составляется из двух отчетов: *QuickRep1* (определенный в форме *ManyGroup*) и *QuickRep1* (определенный в форме *Prostoj*). Почать композитного отчета или его предварительный просмотр осуществляется так же, как для простых отчетов, например

QRCompositeReport1.Preview;

На рис.22 показан композитный отчет, построенный из двух ранее разработанных нами отчетов – простейшего отчета и отчета с группировками данных.

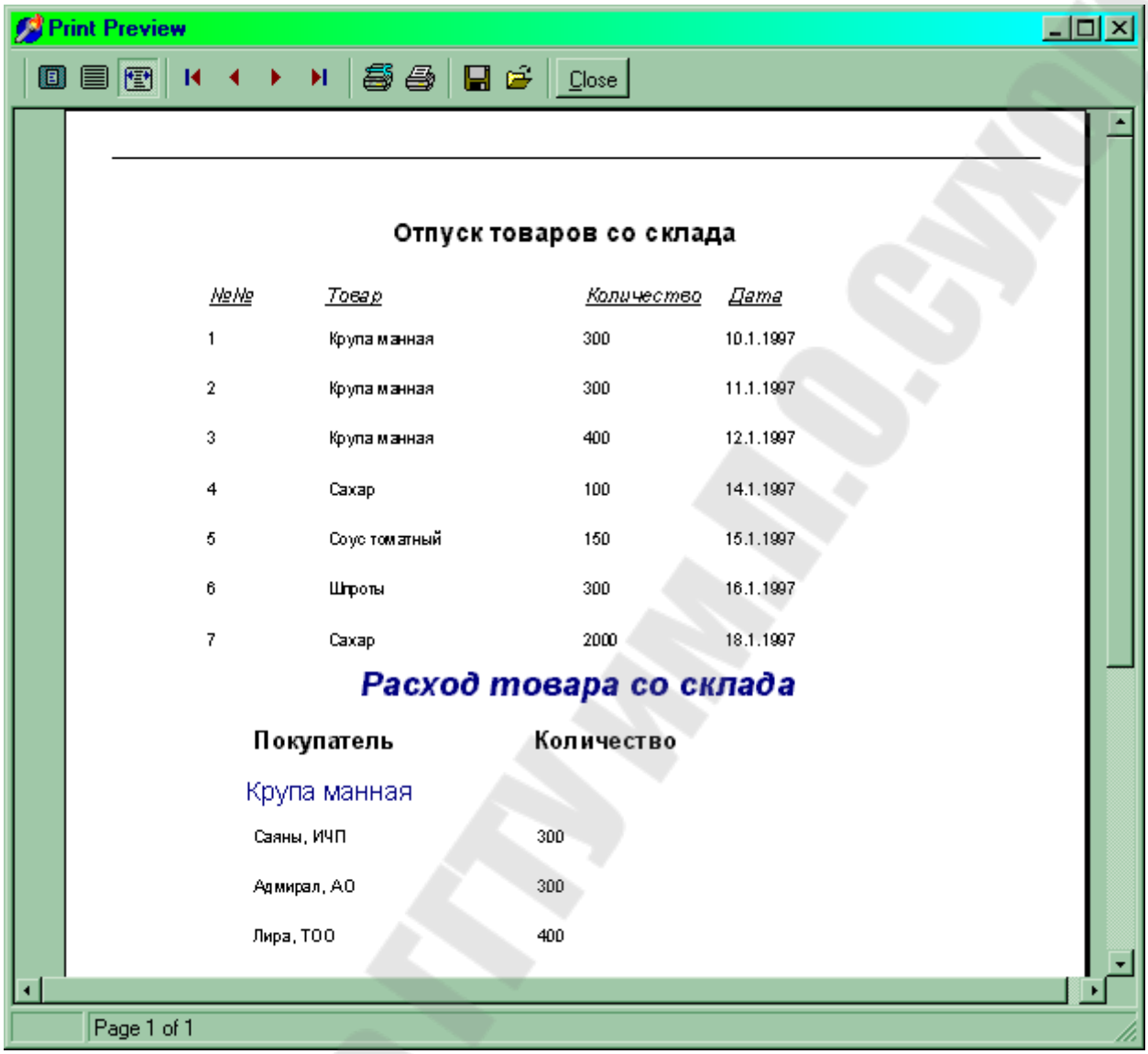


Рис. 22. Композитный отчет, составленный из двух простых отчетов.

Тема 15. Работа с БД по технологии ADO

Для работы с базами в Delphi есть несколько наборов компонентов. Каждый набор очень хорошо подходит для решения определенного круга задач. Все они используют разные технологии доступа к данным и отличаются по возможностям.

В Delphi 5.0 появились компоненты для работы с Microsoft® ActiveX® Data Objects (далее ADO). ADO это технология стандартного обращения к реляционным данным от Microsoft. Эта

технология аналогична BDE по назначению и довольно близка по возможностям.

ADO [Active Data Objects]

технология доступа к данным, разработанная корпорацией Microsoft. Рекомендуется использовать для работы с базами данных Microsoft, а именно MS Access или MS SQL Server.

.

ADOConnection

Компонент ADOConnection обеспечивает соединение с базой данных:

Свойства компонента:

Свойство	Описание
ConnectionString	Строка соединения. Содержит информацию, необходимую для подключения к базе данных
LoginPromt	Признак необходимости запросить имя и пароль пользователя в момент подключения к базе данных. Если значение свойства равно "False", то окно "Login" в момент подключения к базе данных не отображается
Mode	Режим соединения. Соединение с базой данных может быть открыто для чтения [cmRead], записи [cmWrite], чтения/записи [cmReadWrite]
Connected	Признак того, что соединение установлено

ADOTable

Компонент ADOTable представляет собой таблицу [все столбцы] базы данных [обеспечивает доступ к таблице]:

Свойства компонента:

Свойство	Описание

Connection	Ссылка на компонент [ADOConnection], который обеспечивает соединение с источником [базой] данных
ConnectionString	Строка соединения. Содержит информацию о базе данных, элементом которой является таблица, подключение к которой обеспечивает компонент
TableName	Таблица базы данных
Filter	Фильтр - условие отбора записей из таблицы. Позволяет скрыть записи, не удовлетворяющие критерию отбора
Filtered	Признак фильтрации записей. Если значение свойства равно "True", то записи, не удовлетворяющие критерию запроса, не отображаются
ReadOnly	Запрещает [True] или разрешает [False] изменение данных таблицы
Active	Признак того, что соединение с таблицей активно

ADODataSet

Компонент ADODataSet хранит данные, полученные из базы данных в результате выполнения сервером SQL-команды. В отличие от ADOTable, который представляет собой всю таблицу [все столбцы], компонент ADODataSet позволяет прочитать только нужные столбцы или может быть заполнен данными из разных таблиц:

Свойства компонента:

Свойство	Описание
Connection	Ссылка на компонент [ADOConnection], который обеспечивает соединение с источником
CommandText	Команда, которая направляется серверу
Parameters	Параметры команды

Filter	Фильтр. Позволяет скрыть записи, не удовлетворяющие критерию отбора
Filtered	Признак использования фильтра
Activate	Открывает или делает недоступным набор данных

ADOQuery

Компонент ADOQuery позволяет направить запрос серверу базы данных. Обычно он используется, если данные, которые надо получить из базы данных, распределены по нескольким таблицам [если данные находятся в одной таблице, то рекомендуется использовать компонент ADOTable]:

Свойства компонента:

Свойство	Описание
Connection	Ссылка на компонент [ADOConnection], который обеспечивает соединение с источником [базой] данных
ConnectionString	Строка соединения. Содержит информацию о базе данных, которой направляется запрос [SQL-команда]
SQL	SQL-команда [запрос], которая направляется серверу
Filter	Фильтр - условие отбора записей из таблицы. Позволяет скрыть записи, не удовлетворяющие критерию отбора
Filtered	Признак фильтрации записей. Если значение свойства равно "True", то записи, не удовлетворяющие критерию запроса, не отображаются
Active	Признак того, что соединение с таблицей активно

TADOStoredProc – вызов хранимой процедуры. В отличие от BDE и InterBase хранимые процедуры в ADO могут возвращать набор данных, по этому компонент данного типа является потомком от

TDataSet и может выступать источником данных в компонентах типа **TDataSource**.

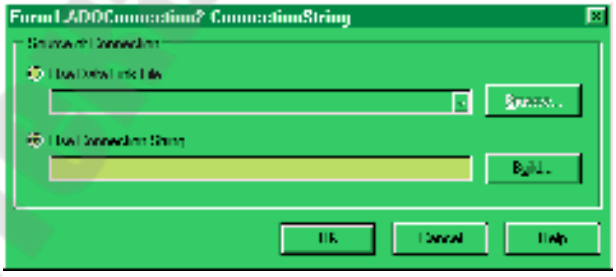
TADOCommand и **TADODataSet** являются наиболее общими компонентами для работы с ADO, но и наиболее сложными в работе. Оба компонента позволяют выполнять команды на языке провайдера данных (так в ADO называется драйвер базы данных).

Разница между ними в том, что команда, исполняемая через **TADODataSet,** должна возвращать набор данных и этот компонент позволяет работать с ними средствами Delphi (например, привязать компонент типа **TDataSource**). А компонент **TADOCommand** позволяет исполнять команды не возвращающие набор данных, но не имеет штатных средств Delphi для последующего использования возвращенного набора данных.

Очевидно, что все компоненты должны связываться с базой данных. Делается это двумя способами либо через компонент **TADOConnection** либо прямым указанием базы данных в остальных компонентах. К **TADOConnection** остальные компоненты привязываются с помощью свойства **Connection**, к базе данных напрямую через свойство **ConnectionString**.

База данных может быть указана двумя способами через файл линка к данным (файл в формате Microsoft Data Link, расширение UDL), либо прямым заданием параметров соединения.

Значение свойства всех **ConnectionString** этих компонент могут быть введены напрямую в текстовой форме, но куда проще вызвать редактор свойства нажав на кнопку «…» в конце поля ввода. Окно этого свойства выглядит так:



При выборе «Use data link file» и нажатии на кнопку «Browse…» появляется стандартный диалог выбора файла. Этот файл можно создать в любом окне explorer-а (в этом окне открытия файла, в самом explorer, на desktop и т.д.) вызвав контекстное меню и выбрав пункт «New/Microsoft Data Link». Потом вызовите локальное меню для созданного файла и выберите в нем пункт «Open». После этого появится property sheet описанный чуть ниже. Эти же вкладки

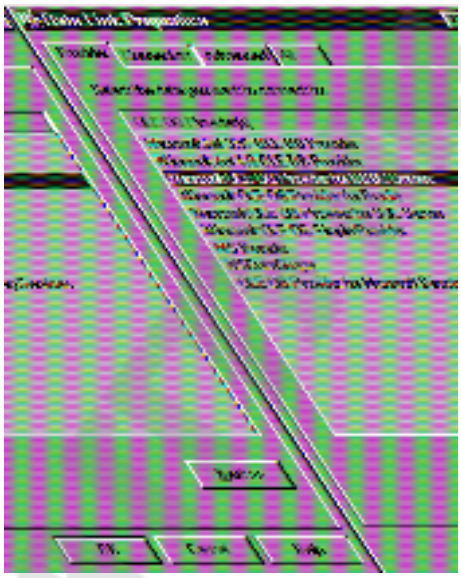
содержит и property sheet, вызываемый через пункт «Property» локального меню UDL файла, но в нем еще есть вкладки относящиеся к самому файлу.

Использование файлов Microsoft Data Link упрощает поддержку приложений, так как возможно использовать средства Windows для настройки приложения.

При выборе в редакторе свойства «Use connection string» и нажатии на кнопку «Build…» появляется такой же property sheet, как и при выборе «Open» для Microsoft Data Link файла.

В этом окне выбирается тип базы данных, местоположение базы и параметры соединения.

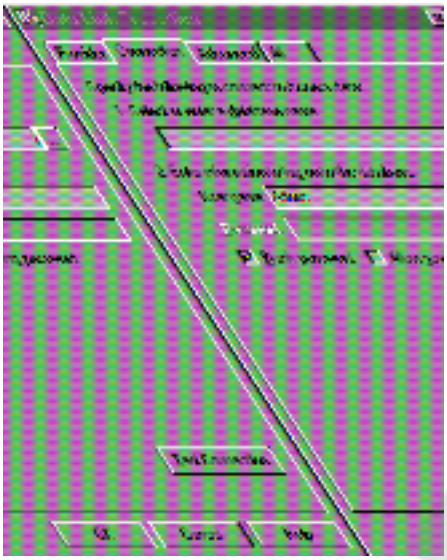
На первой странице выбирается тип базы данных или Provider, в терминах ADO.



Базы MS Access доступны как через «Microsoft Jet OLE DB Provider», так и через «Microsoft OLE DB Provider for ODBC».

Следующая страница зависит от выбранного типа базы, однако для всех типов есть кнопка «Test connection» позволяющая проверить правильность и полноту параметров.

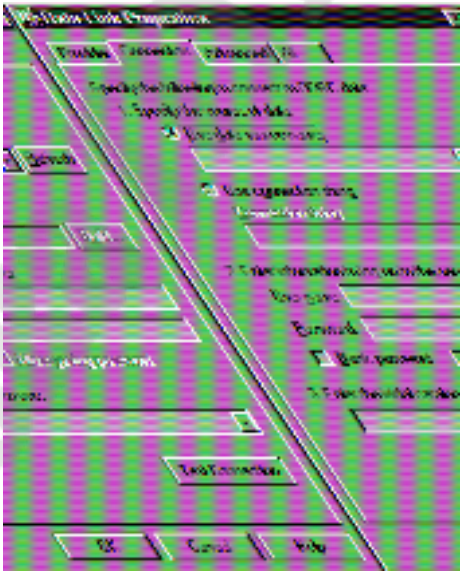
Для «Microsoft Jet OLE DB Provider» она выглядит так:



Checkbox «Blank password» подавляет диалог ввода идентификатора и пароля пользователя при установлении соединения, если поле пароля пустое.

Checkbox «Allow saving password» сохраняет пароль в строке параметров соединения. Если он не отмечен, то введенный пароль будет использоваться только при выполнении тестового соединения.

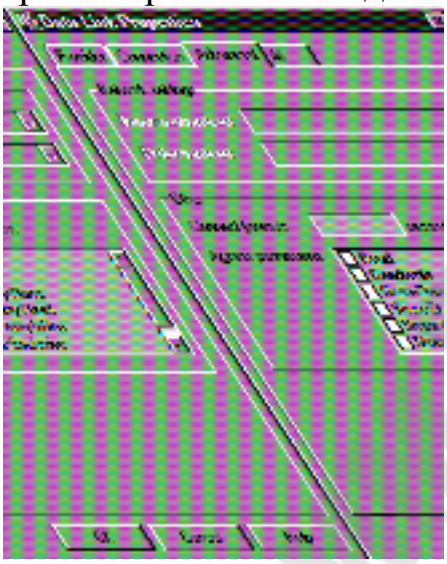
Для «Microsoft OLE DB Provider for ODBC» эта страница выглядит так:



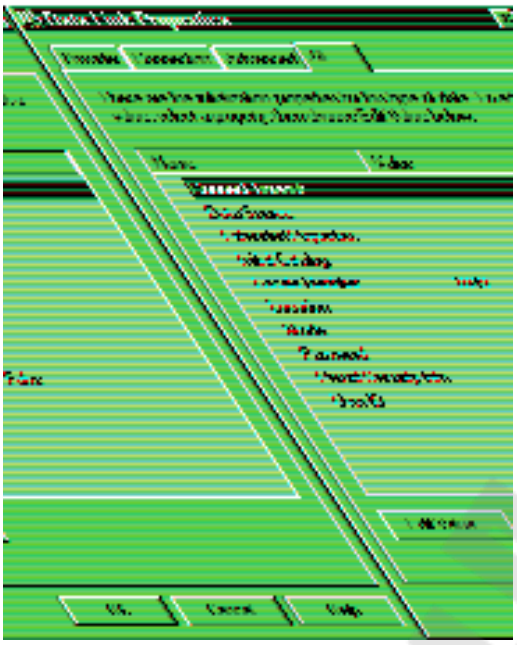
Радиокнопка «Use data source name» позволяет ввести предустановленный алиас ODBC, а через «Use connection string» вводится как алиасы так и тип ODBC драйвера и параметры соединения.

Параметры идентификации пользователя аналогичны выше описанным.

На странице «Advanced» расположены дополнительные параметры, с помощью которых устанавливается уровень доступа к файлу базы данных, таймаут сетевого соединения (то есть время через которое связь будет считаться потерянной, если сервер не отвечает) и уровень глубины проверки секретности соединения.



На странице «All» можно отредактировать все параметры с предыдущих страниц и параметры зависящие от провайдера, но не вошедшие на страницу «Connection». Редактирование осуществляется в виде параметр – значение, причем в текстовой форме, никаких диалогов нет. Помощи то же нет, эти параметры описаны только в документации на провайдер. Ее можно найти в MSDN Data Access Services/Microsoft Data Access Components (MDAC) SDK/Microsoft ActiveX Data Objects (ADO)/Microsoft ADO Programmer's Reference/Using Providers with ADO.



В компоненте **TADOConnection** есть свойства **Provider**, **DefaultDatabase** и **Mode** которые являются альтернативным методом задания частей строки параметров соединения – провайдера, базы данных (например, пути до базы MS Access) и режима совместного использования файлов базы данных. Эти значение этих свойств автоматически включаются в строку соединения, если были заданы до активизации компонента и автоматически выставляются после соединения.

Простейшее приложение

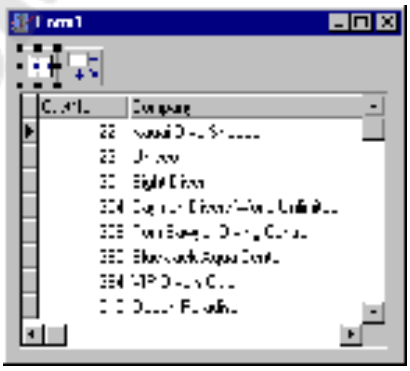
Создадим простейшее приложение, состоящее из одной таблицы.

Создаем форму состоящую из трех компонент

TADOTable с палитры компонент ADO,

TDataSource с палитры компонент Data Access,

TDBGrid с палитры компонент Data Controls.



Тема 16. Работа с MS Office

Работа с MS Office

Так как многие разрабатываемые приложения формируют множество выходных отчетных форм для печати, то возможность формирования ими документов непосредственно в формате Word или Excel выгодно отличало бы их от своих аналогов.

Чтобы работать с MS Office из Delphi существует два основных пути.

- Первый способ (для таблицы Excel) –подключение через ADO, DAO(Data Active Objects) или ODBC.
- Второй способ – это создание его как COM объект.

Через ODBC/ADO

Это не самый быстрый и далеко не первый по возможностям метод (DAO работает на порядок быстрее и предоставляет куда больше возможностей).

1. заходим в Панель управления Windows, идем в свойства ODBC, делаем DSN, используя Excel драйвер, не забываем указать в свойствах на файл Excel. Закрываем ODBC.
2. Ставим на форму ADOConnection. В ConnectionString - строим строку подключения - надо выбрать только ODBC провайдер и на следующей вкладке указать сделанный DSN. Ее можно вообще упростить до вида: "DSN=MyDsn". Теперь вам доступны листы файла как таблицы, а весь файл как база данных.
3. Подключаем ADOQuery к ADOConnection. Создаем таблицу, т.е. новый лист:

Code:

```
Create Table MyTable1 (
Field1 varchar(20),
Field2 varchar(10) )
```

4. Снова переходим в дизайн - ставим на форму ADOTable, указываем как Connection наш компонент с ADOConnection,

теперь если кликнуть на свойстве TableName - вы сможете увидеть в списке сделанную нами таблицу "MyTable1".

5. Соедините таблицу с DBGrid. Работа с таблицей в Excel мало отличается от работы с другими базами данных.

Com соединения

Формирование документа MS Word

Текстовый редактор Word представляет собой COM-сервер и может получать и обрабатывать запросы от внешних программ. Все это позволяет организовать процесс управления и создания документа из внешней программы. Используя этот механизм, можно создать документ программно так же, как это делается вручную (посредством меню, кнопок и клавиатуры), но гораздо быстрей и эффектней.

Word предоставляет три объекта, через которые можно получить доступ к внутренним объектам Word-а и документов.

Word.Application,

Word.Document,

Word.Basic.

Ко всем остальным объектам (текст, таблицы, кнопки, меню и др.) доступ возможен только через них.

Рассмотрим несколько функций, которые позволят запустить Word, создать документ, изменить документ (записать текст), сохранить документ и закрыть Word.

Для создания объекта и его использования применяем переменную W типа variant и библиотеку UWordExcel.

Code:

```
uses UWordExcel;
var W:variant;
Function CreateWord:boolean;
begin
CreateWord:=true;
try
W:=CreateOleObject('Word.Application');
except
CreateWord:=false;
end;
End;
```

Для получения доступа к объекту Word.Application в нашей функции CreateWord используем конструктор CreateOleObject ('Word. Application'). Если редактор Word не установлен в системе, то будет сгенерирована ошибка, и мы получим значение функции = false, если Word установлен, и объект будет создан, то получим значение функции = true.

Эта функция создает объект (W), свойства и методы которого мы будем использовать в дальнейшем.

Если выполнить нашу функцию CreateWord, то Word будет запущен, но не появится на экране, потому что по умолчанию он запускается в фоновом режиме. Чтобы его активировать (сделать видимым) или деактивировать (сделать невидимым), используйте свойство visible объекта W.

Code:

```
Function VisibleWord (visible:boolean):boolean;
begin
VisibleWord:=true;
try
W.visible:= visible;
except
VisibleWord:=false;
end;
End;
```

Для создания документа используем объект Documents объекта W. Этот объект имеет метод Add, используя который, и создаем новый документ.

Code:

```
Function AddDoc:boolean;
Var Doc_:variant;
begin
AddDoc:=true;
try
Doc_:=W.Documents;
Doc_.Add;
except
```

```
AddDoc:=false;
end;
End;
```

Следующим шагом, естественно, является запись любого текста непосредственно в документ.

Code:

```
Function SetTextToDoc(text_: string;InsertAfter_: boolean): boolean;
var Rng_:variant;
begin
  SetTextToDoc:=true;
  try
   Rng_:=W.ActiveDocument.Range;
   If InsertAfter_ then Rng_.InsertAfter(text_)
                   else Rng_.InsertBefore(text_);
  except
  SetTextToDoc:=false;
end;
End;
```

В этой функции используем объект Range и его методы InsertAfter и InsertBefore для того, чтобы вставить текст в документ с позиции курсора или до позиции курсора. Наша функция будет вставлять текст в активный документ в область курсора или выделенного текста.

Code:

```
Rng_:=W.ActiveDocument.Range;
if InsertAfter_ then Rng_.InsertAfter(text_)
                else Rng_.InsertBefore(text_);
```

можно заменить следующим фрагментом:

Code:

```
if InsertAfter_  then W.ActiveDocument.Range. InsertAfter(text_)
                 else W.ActiveDocument.Range. InsertBefore(text_);
```

Для сохранения документа используем метод SaveAs объекта ActiveDocument. Функция SaveDocAs использует этот метод и сохраняет документ в заданный файл.

Code:

```
Function SaveDocAs(file_:string):boolean;
begin
  SaveDocAs:=true;
  try
    W.ActiveDocument.SaveAs(file_);
  except
    SaveDocAs:=false;
  end;
End;
```

Закрыть сохраненный документ можно, используя метод Close объекта ActiveDocument.

Code:

```
Function CloseDoc:boolean;
begin
  CloseDoc:=true;
  try
    W.ActiveDocument.Close;
  except
    CloseDoc:=false;
  end;
End;
```

Закрыть Word можно, используя метод Quit объекта Application(W).

Code:

```
Function CloseWord:boolean;
begin
  CloseWord:=true;
  try
    W.Quit;
  except
    CloseWord:=false;
```

```
  end;
End;
```

Код в Delphi

Code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if CreateWord
then begin
  Messagebox(0,'Word запущен.','',0);
  VisibleWord(true);
  Messagebox(0,'Word видим.','',0);
  VisibleWord(false);
  Messagebox(0,'Word невидим.','',0);
  VisibleWord(true);
  Messagebox(0,'Word видим.','',0);
  If AddDoc then begin
   Messagebox(0,'Документ создан.','',0);
   SetTextToDoc('Мой первый текст',true);
   Messagebox(0,'Добавлен текст','',0);
   SaveDocAs('c:\Мой первый текст');
   Messagebox(0,'Текст сохранен','',0);
   CloseDoc;
  end;
  Messagebox(0,' Текст закрыт','',0);
  CloseWord;
end;
end;
```

Для открытия ранее созданного документа используем метод Open коллекции Document. Функция Open, кроме обязательного аргумента (имени файла), может иметь ряд дополнительных аргументов, которые определяют режим открытия. Она возвращает ссылку на объект типа Document.

Code:

```
Function OpenDoc (file_:string):boolean;
Var Doc_:variant;
begin
OpenDoc:=true;
try
Doc_:=W.Documents;
Doc_.Open(file_);
except
OpenDoc:=false;
end;
End;
```

Для перевода курсора в начало документа используем свойства End и Start объекта W.Selection. Эту функцию необходимо использовать каждый раз перед началом поиска текста, чтобы поиск осуществлялся с начала документа. Свойства End и Start объекта Selection можно использовать и для выделения диапазона текста, при этом в Start записывается номер начального символа фрагмента в тексте, а в End - номер конечного. В данном случае необходимо в оба поля записать нули.

Code:

```
Function StartOfDoc:boolean;
begin
StartOfDoc:=true;
try
W.Selection.End:=0;
W.Selection.Start:=0;
except
StartOfDoc:=false;
end;
End;
```

Функция поиска (FindTextDoc) фрагмента текста состоит из трех операторов. Первый и второй задают направление поиска (от начала к

концу) и фрагмент для поиска, соответственно. Третий оператор выполняет поиск и возвращает результат. Функция возвращает True, если поиск удачный, и False - если нет. Во всех трех операторах используем поля и методы объекта Selection.

Code:

```
Function FindTextDoc (text_:string):boolean;
begin
FindTextDoc:=true;
Try
W.Selection.Find.Forward:=true;
W.Selection.Find.Text:=text_;
FindTextDoc := W.Selection.Find.Execute;
except
FindTextDoc:=false;
end;
End;
```

Функция FindTextDoc находит и выделяет фрагмент текста в документе. Для того, чтобы вставить новый текст вместо выделенного, создадим еще одну функцию. PasteTextDoc состоит из двух операторов, удаления выделенного фрагмента и вставки нового текста с положения курсора. Оба эти оператора используют объект Selection объекта W. Действие этой функции отличается от SetTextToDoc тем, что она вставляет изменения вместо выделенного фрагмента текста.

Code:

```
Function PasteTextDoc (text_:string):boolean;
begin
PasteTextDoc:=true;
Try
W.Selection.Delete;
W.Selection.InsertAfter (text_);
except
PasteTextDoc:=false;
end;
End;
```

Более удобной является функция, которая одновременно подставляет новый текст на место найденного фрагмента.

Code:

```
Function FindAndPasteTextDoc
(findtext_,pastetext_:string): boolean;
begin
FindAndPasteTextDoc:=true;
try
W.Selection.Find.Forward:=true;
W.Selection.Find.Text:= findtext_;
if W.Selection.Find.Execute then begin
  W.Selection.Delete;
  W.Selection.InsertAfter (pastetext_);
end else FindAndPasteTextDoc:=false;
except
FindAndPasteTextDoc:=false;
end;
End;
```

Печать документа. Диалог активизируется через метод Show объекта Dialogs(wdDialogFilePrint).

Code:

```
Function PrintDialogWord:boolean;
Const wdDialogFilePrint=88;
begin
PrintDialogWord:=true;
try
W.Dialogs.Item(wdDialogFilePrint).Show;
except
PrintDialogWord:=false;
end;
End;
```

Имея необходимый набор функций, можно приступать к написанию программы (процедуры) создания простого документа Word средствами Delphi.

Для формирования документа нам необходим шаблон - текст в формате Word (файл с расширением doc, rtf). Создадим вручную этот файл ("C:\Шаблон платежного поручения.doc").

Общий алгоритм формирования документа таков:

1. Открываем шаблон, используя функцию открытия ранее созданного документа.
2. Ищем слова-переменные и подставляем вместо них реальные значения, например, из базы данных.
3. Сохраняем документ под новым именем.
4. Печатаем документ, если это необходимо.
5. Закрываем документ.

Документ готов, и с ним можно работать как обычно, копировать, переименовывать и др.

Литература

1. Гамильтон Б. ADO.NET Сборник рецептов. Для профессионалов. – СПб.: Питер, 2005. – 576 с.

2. Фаронов В.В. Программирование баз данных в Delphi 7. Учебный курс. – СПб.: Питер, 2006. – 459 с.

3. Архангельский А.Я. Приемы программирования в Delphi на основе VCL. – М.: Бином-Пресс, 2006. – 944 с.

4. Архангельский А.Я. Справочное пособие: язык Delphi, классы, функции Win32 и .NET. – М.: Бином-Пресс, 2006. – 1152 с.

5. Шпак Ю.А. Delphi 7 на примерах. – К.: Юниор, 2003. – 384 с.

6. Дарахвелидзе П.Г., Марков Е.П. Программирование в Delphi 7. – СПб.: БХВ-Петербург, 2003. – 784 с.

7. Бакнелл Дж.М. Фундаментальные алгоритмы и структуры данных в Delphi / Пер. с англ. – СПб.: ДиаСофтЮп, 2003. – 560 с.

Содержание

Рябченко Алексей Иванович

СРЕДСТВА ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ

Курс лекций
по одноименной дисциплине
для слушателей специальности 1-40 01 73
«Программное обеспечение
информационных систем»
заочной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 04.01.14.
Рег. № 44Е.
http://www.gstu.by