Казанский государственный университет

Факультет вычислительной математики и кибернетики

Кафедра системного анализа и информационных технологий

Устюгова В.Н.

Использование Delphi для создания приложений баз данных

Учебно-методическое пособие

КАЗАНЬ 2010

Представленные методические указания являются методическим пособием для проведения практических занятий и самостоятельного изучения компонентов Delphi для создания приложений баз данных.

Учебно-методическое пособие может быть рекомендовано для студентов изучающих дисциплину «Практикум на ЭВМ».

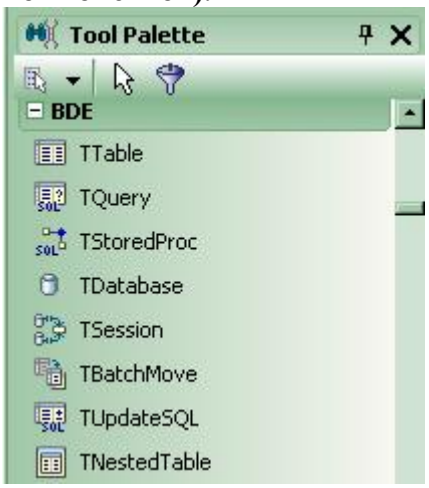
Учебно-методическое пособие публикуется по решению учебно-методической комиссии факультета вычислительной математики и кибернетики КГУ от 13 мая 2010 года.

Лабораторная работа №1

Создание простейшего приложения для работы с базой данных

Одним из традиционных способов взаимодействия приложения, созданного в среде разработки Delphi, и базы данных является использование процессора баз данных Borland Database Engine (BDE). Он представляет собой набор динамических библиотек, функции которых позволяют не только обращаться к данным, но и эффективно управлять ими на стороне приложения.

Для работы с источниками данных при посредстве BDE в Delphi имеется специальный набор компонентов, расположенных на странице «**BDE**» «**Tool Palette**» (**Палитры компонентов)**.



Эти компоненты для работы с базами данных используют возможности BDE, обращаясь к его функциям и процедурам. Механизм доступа к BDE инкапсулирован в базовом классе TBDEDataSet. Поэтому в процессе программирования нет необходимости использовать функции BDE напрямую. Почти все, что можно сделать путем прямого обращения, можно сделать и через компоненты. BDE взаимодействует с базами данных при посредстве драйверов.

Фирма Borland перестанет поддерживать BDE в новых версиях Delphi и рекомендует использовать технологию dbExpress.

В составе BDE поставляются стандартные драйверы, обеспечивающие доступ к СУБД Paradox, dBASE, FoxPro и текстовым файлам. Локальные драйверы устанавливаются автоматически совместно с ядром процессора. Один из них можно выбрать в качестве стандартного драйвера, который имеет дополнительные настройки, влияющие на функционирование процессора БД.

Доступ к данным серверов SQL обеспечивает отдельная система драйверов — SQL Links. С их помощью в Delphi можно разрабатывать приложения для серверов Oracle 8, Informix, Sybase, DB2 и InterBase. Эти драйверы необходимо устанавливать дополнительно.

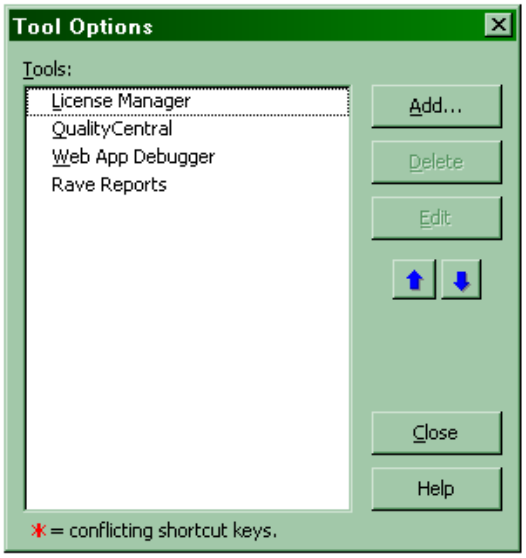
Для доступа к данным приложение и BDE должны обладать информацией о местоположении файлов требуемой базы данных.

Чтобы не перекомпилировать проект в случае изменения маршрута к базе данных, можно использовать псевдоним базы данных, который представляет собой именованную структуру, содержащую путь к файлам БД и некоторые дополнительные параметры. Псевдоним – это имя, связанное с маршрутом к базе данных. Помимо маршрута к файлам базы данных, псевдоним BDE обязательно содержит информацию о драйвере БД, который используется для доступа к данным. Наличие других параметров зависит от типа драйвера, а значит, от типа СУБД. Тогда при переносе приложения на другой компьютер достаточно создать стандартными средствами BDE одноименный псевдоним и настроить его на нужный каталог. При этом само приложение не требует переделок, т.к. работает с псевдонимом.

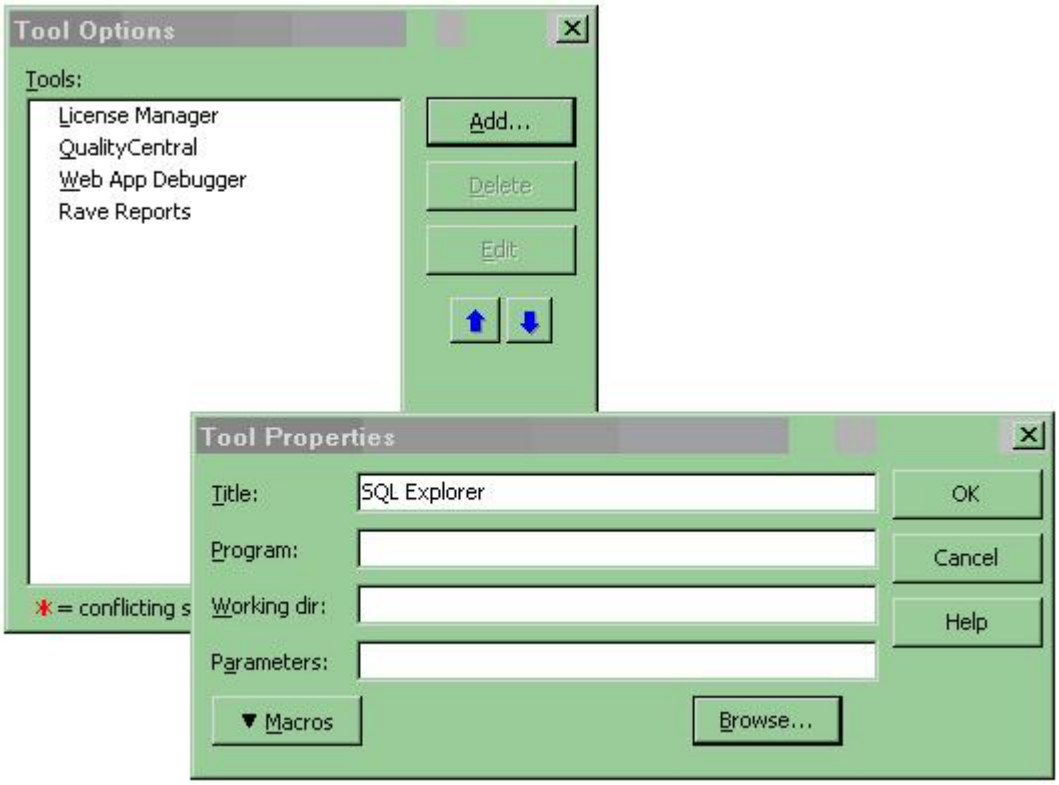
Для управления псевдонимами баз данных в составе BDE имеется специальная утилита - «**SQL Explorer**» (исполняемый файл dbexplorer.EXE). Если пункта «**SQL Explorer**» нет в меню «**Tools**», то эту утилиту следует добавить следующим образом.

Добавление «SQL Explorer» к панели инструментов «Tools»

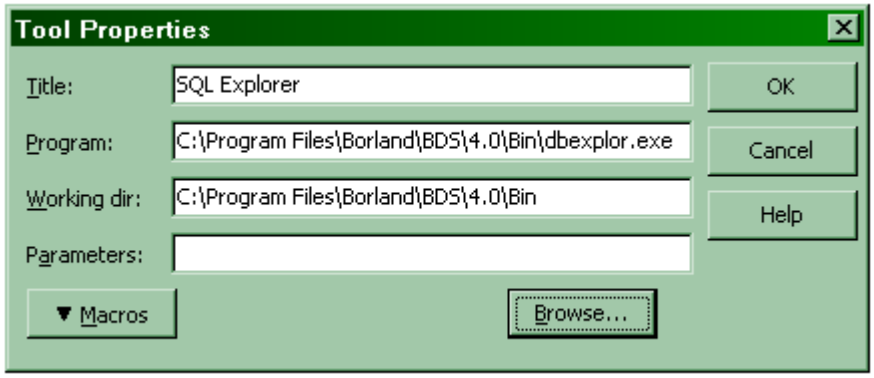
1. Выбрать меню «**Tools**»->«**Configure Tools**». Откроется окно «**Tool Options**»



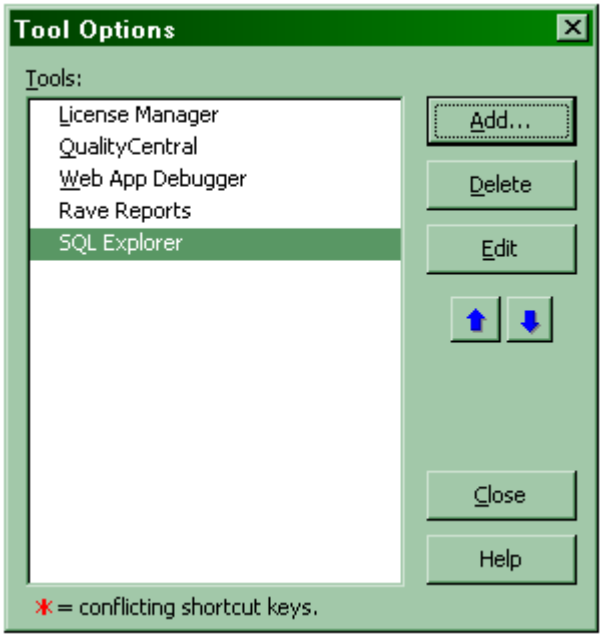
2. Выбрать кнопку «**Add**». В окне «**Tool Properties**» в поле «**Title**» ввести SQL Explorer. Используя кнопку «**Browse**» найти файл dbexplor.exe (обычно папка **bin**, каталога в который установили Delphi).


Tool Options
Tools:
License Manager
QualityCentral
Web App Debugger
Rave Reports
Add...
Delete
Edit
= conflicting s
Tool Properties
Title:
SQL Explorer
Program:
Working dir:
Parameters:
Macros
Browse...
OK
Cancel
Help

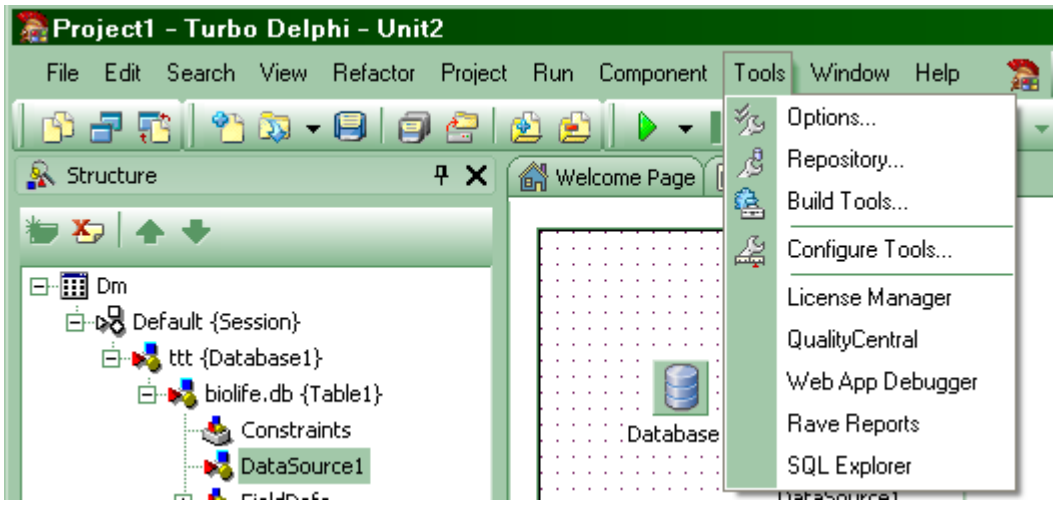

Select Transfer Item
Папка:
bin
Недавние документы
Рабочий стол
Мои документы
Мой компьютер
Сетевое окружение
config
bds.exe
BDSLauncher.exe
bdsreg.exe
brc32.exe
brcc32.exe
CABARC.EXE
convert.exe
DataExplore.exe
dbexplor.exe
DCC32.EXE
drinterop.exe
etm.exe
grep.exe
impdef.exe
IMPLIB.EXE
LicenseManager.exe
make.exe
rmtdbg105.exe
scktsrvr.exe
serverinfo.exe
sqlmon.exe
tasm32.exe
tdump.exe
tlib.exe
tlibimp.exe
tregsvr.exe
webappdbg.exe
WSDLImp.exe
xmlmapper.exe
Имя файла:
dbexplor.exe
Тип файлов:
Programs (*.exe;*.com;*.pif)
Открыть
Отмена
Справка



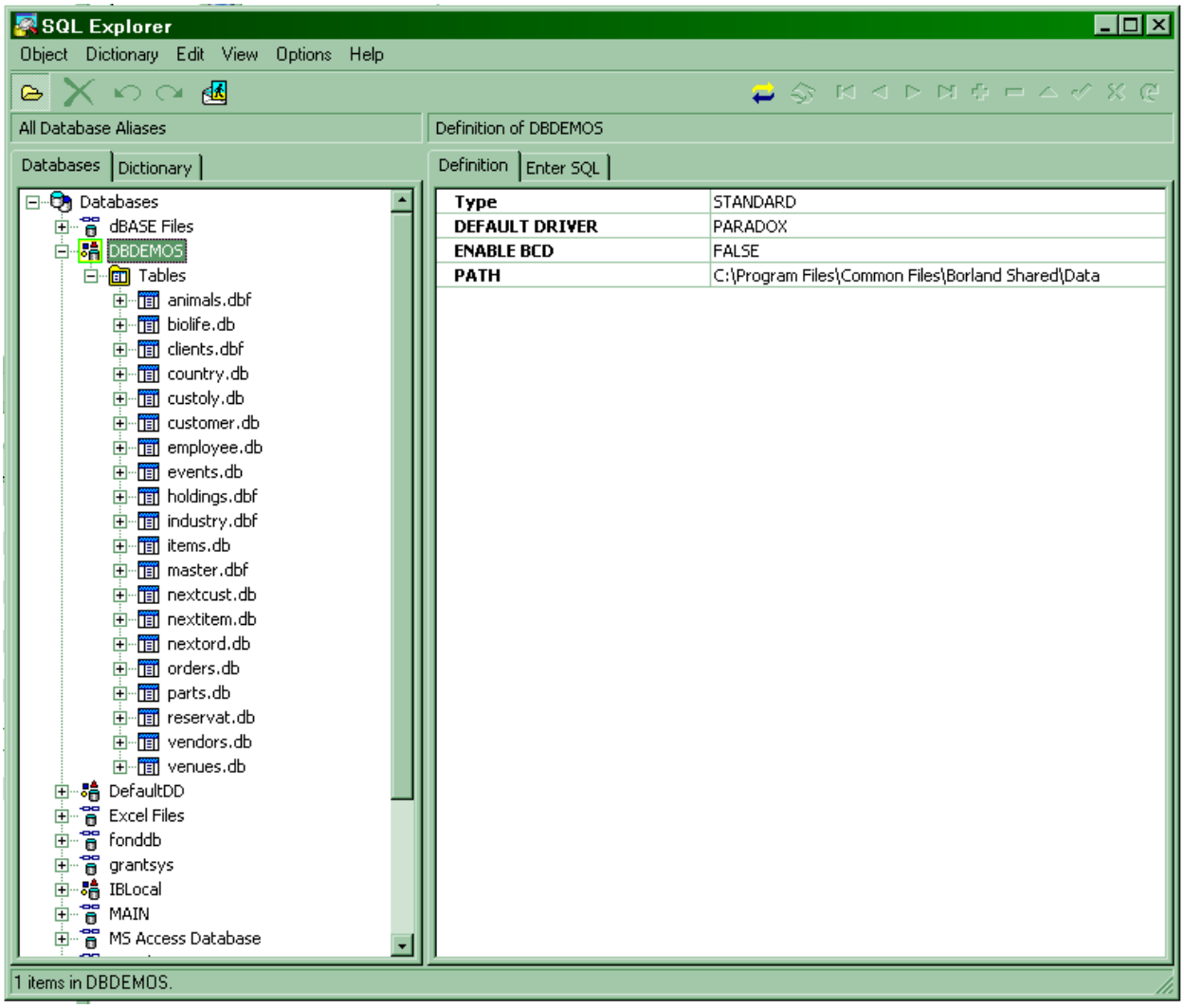
3. Нажать на кнопку «**ОК**».



В список инструментов добавится «**SQL Explorer**».

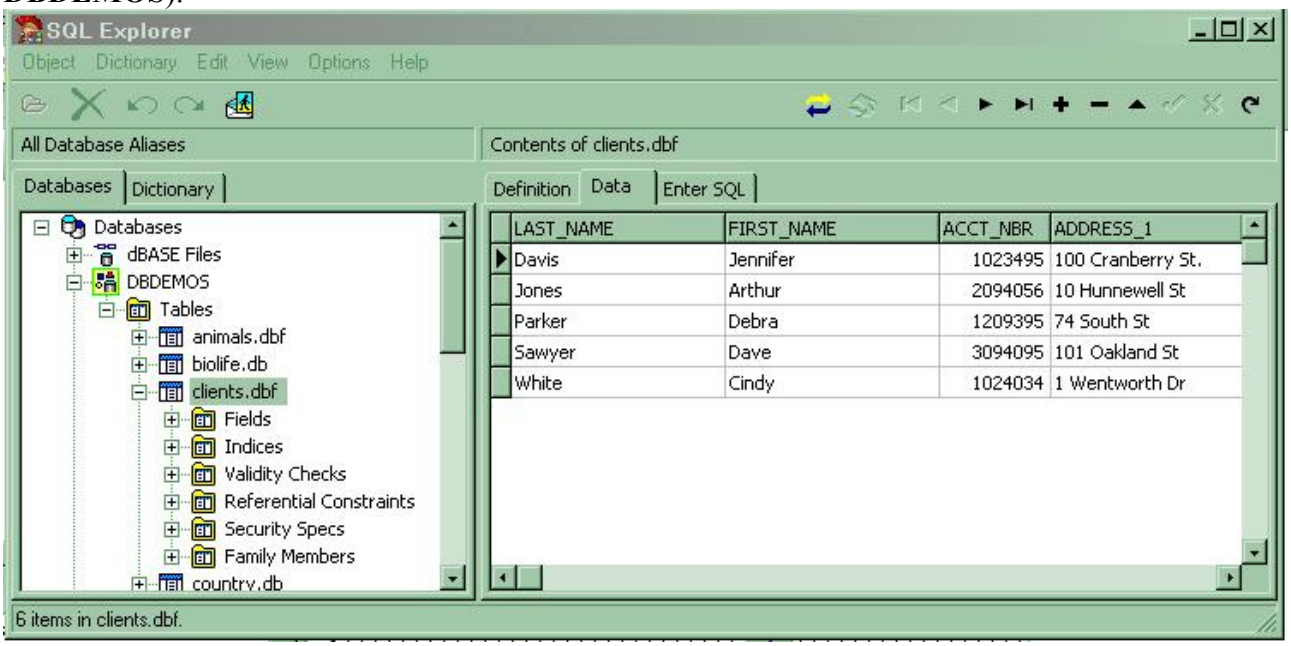


4. Запустить SQL Explorer.



5. Научиться работать в программе SQL Explorer.

Используя SQL Explorer необходимо ознакомиться со структурой и содержанием таблиц, которые будем использовать для выполнения заданий (псевдоним базы данных **DBDEMOS**).



Если Вы работаете в других версиях Delphi, то далее приводимые компоненты могут находиться на других страницах палитры компонентов.

Для создания приложения баз данных необходимо настроить три типа компонентов:

- компонент набора данных (НД): **TDatabase**, **TTable**, **TQuery**, **TStoredProc** (на странице «**BDE**» «**Tool Palette**»);
- компонент **TDataSource** (на странице «**Data Access**» «**Tool Palette**»);
- один или несколько компонентов отображения данных (на странице «**BDE**» «**Data Controls**»).

Компоненты для работы с базой данных можно разделить на **невизуальные**, предназначенные для подключения к базам данных и они не видны в запущенном приложении, и **визуальные**, которые используются для отображения данных.

Компонент TDatabase

Компонент **TDatabase** (невизуальный) позволяет создавать локальные псевдонимы **BDE**, так что приложению не потребуются псевдонимы, содержащиеся в конфигурационном файле **BDE**. Этим локальным псевдонимом в приложении могут воспользоваться все имеющиеся **TTable**, **TQuery** и **TStoredProc**. **TDatabase** также позволяет разработчику настраивать процесс подключения, подавляя диалог ввода имени и пароля пользователя, или заполняя необходимые параметры. Более важно то, что **TDatabase** может обеспечивать единственную связь с базой данных, суммируя все операции с базой данных через один компонент. Это позволяет элементам управления для работы с БД иметь возможность управления транзакциями.

Компонент TTable

Компонент **TTable (**невизуальный, реализован классом TTable**)** инкапсулирует таблицу реляционной базы данных, причем независимо от типа базы данных. Для доступа к данным компонент использует функции BDE .

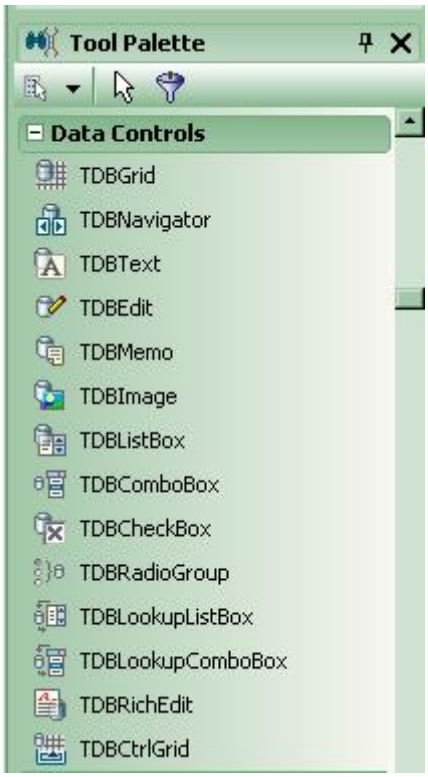
Необходимая для работы база данных задается свойством **DatabaseName**, в котором можно указать зарегистрированный в BDE псевдоним БД, локальное имя базы данных компонента **TDatabase** или полный путь к файлам БД. Таблица, на основе которой создается набор данных, определяется свойством **TableName**. Тип таблицы задается свойством **TableType**, обычно это свойство имеет значение ttDefault, т.к. включается автоматическое определение типа таблицы по расширению файла. При помощи методов **Open** и **Close** набор данных (НД) открывается и закрывается. О состоянии НД можно судить по значению свойства **Active** (True – НД открыт). Свойства **MasterSource**, **MasterField**, **IndexName** дают возможность установить отношение типа главный/подчиненный с другой таблицей.

От классов **TDataSet** и **TBDEDataSet** унаследован обширный набор методов-обработчиков событий, позволяющий решать любые задачи по управлению набором данных.

Компонент TDatasource

Все визуальные компоненты могут подсоединится к **TTable** только через вспомогательный компонент **TDatasource** (невизуальный) - находящийся на странице «**Data Access**» «**Tool Palette**». Свойство **Dataset** у этого компонента должно указывать на существующий НД (TTable, TQuery). Все визуальные компоненты будут "видеть" данные из таблицы через компонент **TDatasource**.

Визуальные компоненты



Визуальные компоненты предназначены для отображения данных.

Модуль данных

Для размещения компонентов доступа к данным в приложении баз данных можно использовать специальную форму — модуль данных. В модуле данных можно размещать только невизуальные компоненты. Модуль данных доступен разработчику, как и любой другой модуль проекта. Пользователь приложения не может увидеть модуль данных во время выполнения.

Для создания модуля данных выбрать в меню «**File**»->«**New**»->«**Delphi Projects**»->«**Delphi Files**»->«**Data Module**».

У модуля данных почти полностью отсутствуют свойства и методы-обработчики событий, но так как его основное назначение — хранение других невизуальных компонентов, то этого минимума свойств вполне достаточно.

1. Изучение приложения **mastapp.**

1.1. Открыть приложение **mastapp** (файл проекта **mastapp.bdsproj**) из папки
c:\Program Files\Borland**BDS\4.0\Demos\DelphiWin32\VCLWin32\Db\MastApp**
Если Вы работаете в другой версии Delphi, путь к файлу проекта может быть другим.

1.2. Запустить приложение на выполнение (из меню «**Run**»->«**Run**», или с панели инструментов «**Run**» **(F9)**). Разобраться с тем, что делает приложение.

1.3. В режиме редактирования (конструктора) открыть форму **MastData** (меню «**View**»->«**Forms**», или с панели инструментов «**View Forms**» (Shift F12))

1.4. Открыть окно инспектора объектов (меню «**View**»->«**Object Inspector**»)

1.5. Исследовать свойства (Properties) и события (Events) объектов (компонентов), находящихся на форме. (**TDatabase, TTable, TDataSource, TQuery**). Обратить внимание на связи между этими объектами.

1.6. Открыть форму **EdOrderForm**. Определить какого типа объекты находятся на форме. Исследовать свойства объектов, связанных с таблицами базы данных. Обратить внимание на то, как эти объекты связаны с объектами формы **MastData (модуль данных)**.

 **2.** Создать приложение, которое будет работать с таблицей **animals** базы данных **C:\Program Files\Common Files\Borland Shared\Data.**

2.1. Создать собственный проект «**File**»->«**New**»->«**VCL Forms Application**» – «**Delphi for Win32**».

2.2. Создать модуль данных («**File**»->«**New**»->«**Delphi Projects**»-> «**Delphi Files**»->«**Data Module**»), в котором разместить невизуальные компоненты для работы с базой данных (Из палитры компонентов «**BDE**» и «**Data Access**»).

2.3. Создать форму (например, Form1), на которой разместить компоненты **TDBGrid, TDBNavigator, TBDEdit, TDBImage** (из палитры компонентов **Data Controls**). Связать визуальные компоненты формы с невизуальными компонентами модуля данных.

2.4. Запустить созданное приложение на выполнение.

2.5. Исследовать какие действия выполняет компонент навигатор (**TDBNavigator**). Смоделировать работу навигатора, используя компонент **TButton** (кнопка).

2.6. При добавлении новой записи уметь в таблицу положить картинку, а при просмотре записи уметь сохранить картинку в файл. (Использовать компоненты **TOpenPictureDialog, TSavePictureDialog** из палитры компонентов **Dialogs**).

Возможные проблемы:

1. В форме не получается сослаться на невизуальные компоненты из модуля данных.

(Чтобы из формы **Form1** были видны компоненты модуля данных, надо с закладки «**Design**» формы **Form1** перейти на закладку «**Code**» и в секции «**interface**» в строку «**uses**» добавить имя, указанное после «**unit**» в коде модуля данных (Например, DataMod, если в модуле данных указано unit DataMod;.).

2. Нет отображения данных из таблицы на форме.

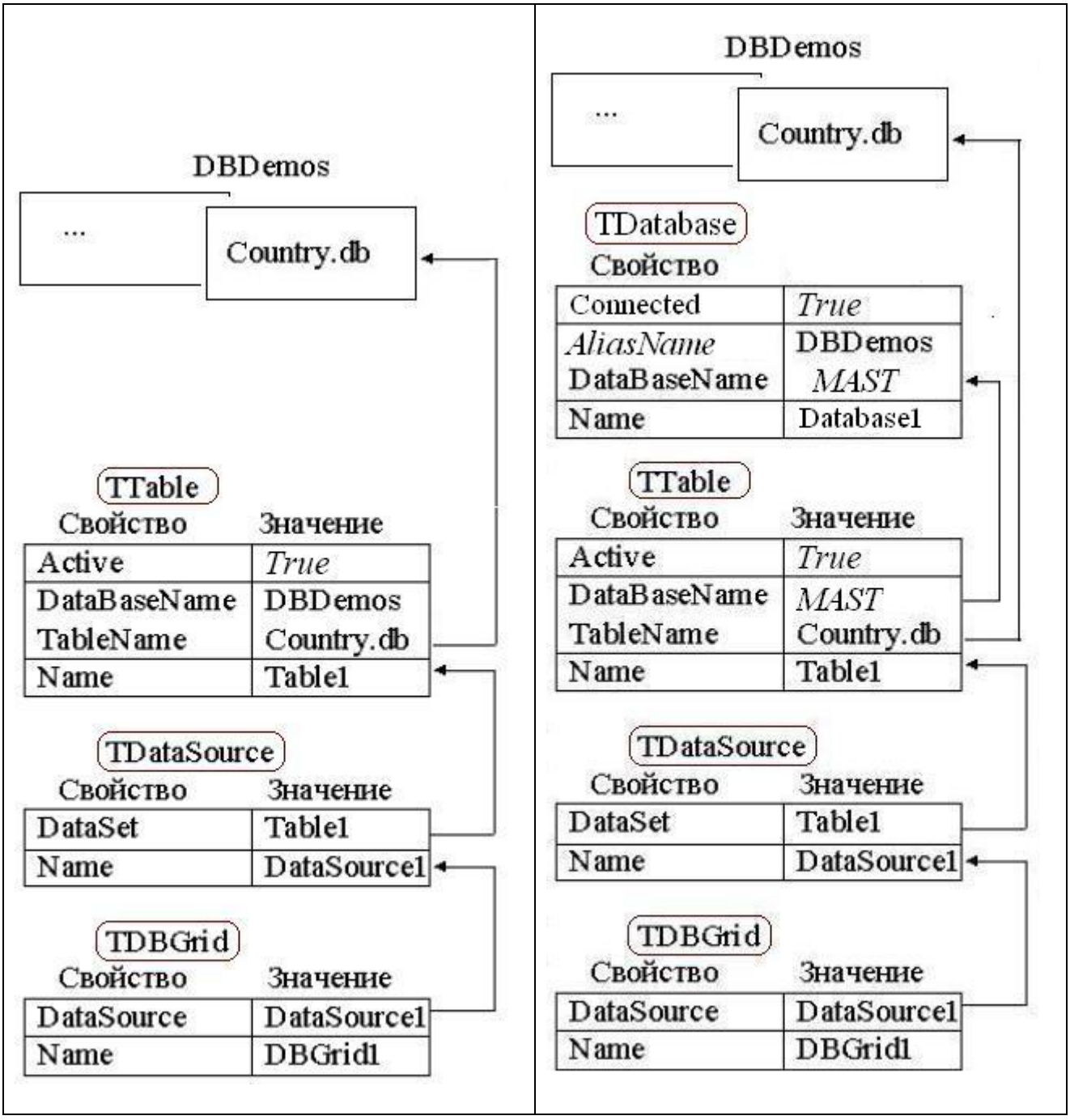
- Вы неверно связали компоненты меджу собой.
- Не установили свойство **Active** компонента **TTable** в **true** (или не открыли таблицу методом **Open**, например на событии **OnActivate** формы).

3. Не знаете что писать в процедуре OnClick нажатия на кнопку

- Каждую кнопку связать с методом компонента TTable, например, встать на последнюю запись
Table1. Last; // если компонент TTable находится на форме
MastData.Orders.Last; // если компонент TTable находится в модуле данных MastData

4. В компонентах TBDEdit, TDBImage,…и других предназначенных для визуализации данных, находящихся в «Data Controls» нет отображения данных.

- Забыли связать эти компоненты с соответствующим полем из отображаемой таблицы.



5. Нет на экране окна палитры компонентов «Tool Pallete».

- меню «**View**»-> «**Tool Pallete**»

Лабораторная работа №2

Создание связей между таблицами

В этом задании надо будет создать маленькое приложение из одной формы и модуля данных. Форма должна отображать данные из двух таблиц: главной и подчиненной (Customers, Orders).

 **1**. Изучение приложения **mastapp.**

1.1. Открыть приложение mastapp (файл проекта **mastapp.bdsproj**) из папки

c:\Program Files\Borland**BDS\4.0\Demos\DelphiWin32\VCLWin32\Db\MastApp**

1.2. Запустить приложение на выполнение (из меню «**Run**»->«**Run**», или с панели инструментов **Run**» **(F9**)). Разобраться с тем, что делает приложение.

1.3. В режиме редактирования (конструктора) открыть форму **MastData** (меню «**View**»-> «**Forms**», или с панели инструментов «**View Forms**» (Shift F12))

1.4. Открыть окно инспектора объектов (меню «**View**»->«**Object Inspector**»)

1.5. Исследовать свойства (Properties) и события (Events) объектов (компонентов), находящихся на форме. (**TDatabase, TTable, TDataSource, TQuery**). Обратить внимание на связи между этими объектами.

1.6. Открыть форму **EdOrderForm**. Определить какого типа объекты находятся на форме. Исследовать свойства объектов, связанных с таблицами базы данных. Обратить внимание на то, как эти объекты связаны с объектами формы **MastData (модуль данных)**.

 **2.** Создать приложение, которое будет работать с таблицами **Customers и Orders** базы данных **C:\Program Files\Common Files\Borland Shared\Data.**

2.1. Создать собственный проект «**File**»->«**New**»->«**VCL Forms Application**» – «**Delphi for Win32**».

2.2. Создать модуль данных («**File**»->«**New**»->«**Delphi Projects**»-> «**Delphi Files**»->«**Data Module**»), в котором разместить невизуальные компоненты для работы с базой данных (Из палитры компонентов «**BDE**» и «**Data Access**»), в нашем случае для связи таблиц **Customers, Orders.**

2.3. Создать форму (например, Form1), на которой разместить 2 компонента **TDBGrid и 1 компонент TDBNavigator (аналог формы EdOrderForm из учебного приложения)**. Связать визуальные компоненты формы с невизуальными компонентами модуля данных.

2.4. Запустить созданное приложение на выполнение

Возможные проблемы:

1. В форме не получается сослаться на невизуальные компоненты из модуля данных.

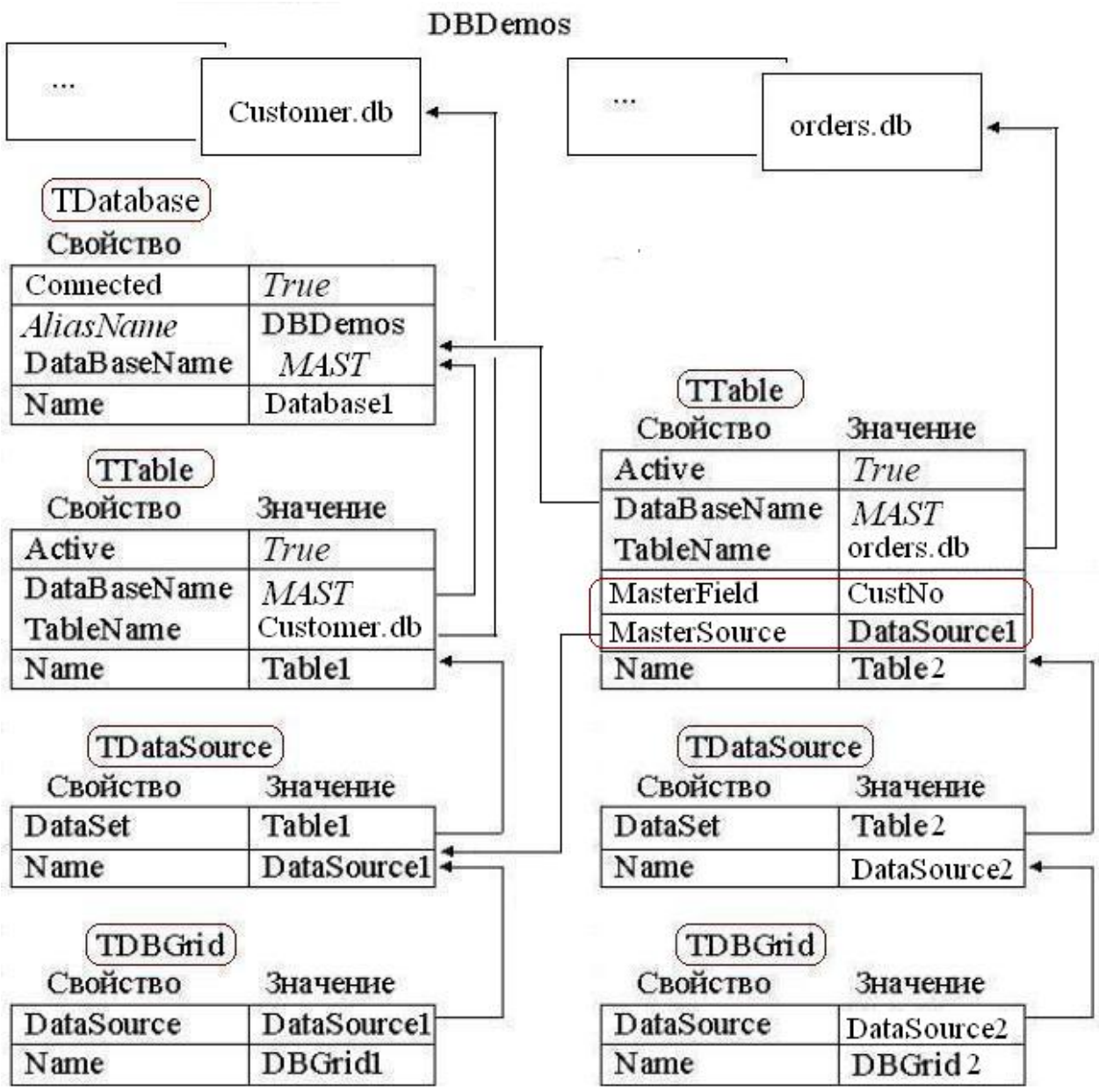
(Чтобы из формы **Form1** были видны компоненты модуля данных, надо с закладки «**Design**» формы **Form1** перейти на закладку «**Code**» и в секции «**interface**» в строку «**uses**» добавить имя, указанное после «**unit**» в коде модуля данных (Например, DataMod, если в модуле данных указано unit DataMod;.).

2. Нет отображения данных из таблицы на форме.

- Вы неверно связали компоненты меджу собой.
- Не установили свойство **Active** компонента **TTable** в **true** (или не открыли таблицу методом **Open**, например на событии **OnActivate** формы).

3. В компонентах TBDEdit, TDBImage,…и других предназначенных для визуализации данных, находящихся в «Data Controls» нет отображения данных.

- Забыли связать эти компоненты с соответствующим полем из отображаемой таблицы.



4. Нет на экране окна палитры компонентов «Tool Pallete».

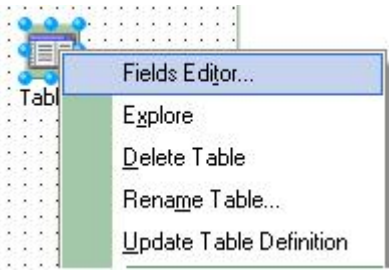
- меню «**View**»-> «**Tool Pallete**»

Лабораторная работа № 3

Использование редактора полей

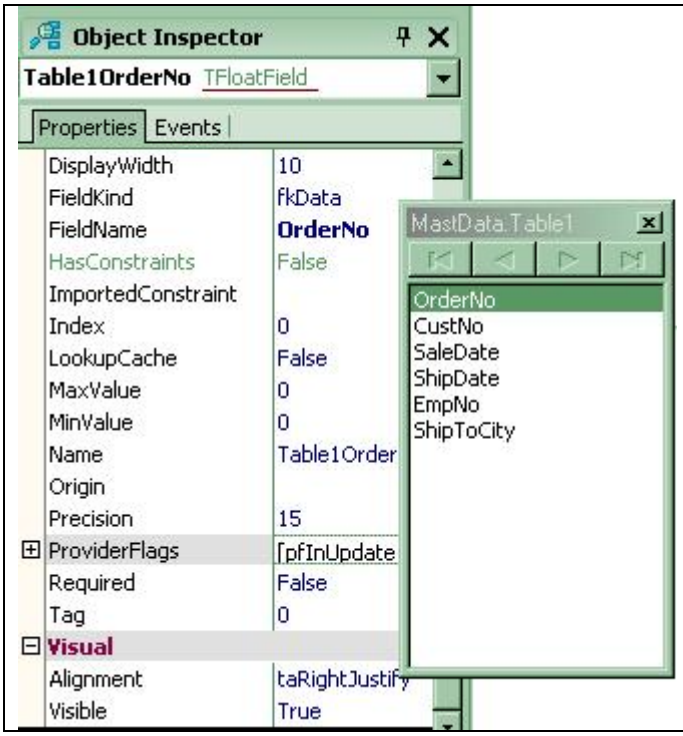
Для того чтобы определить один или несколько компонентов **TField** следует:

1. Выбрать набор данных (компонент TTable или TQuery), нажать правую кнопку мыши и в появившемся меню выбрать пункт «**Fields Editor**» (редактор полей). Откроется окно с шапкой, содержащей имя таблицы.



2. Вновь нажать правую кнопку мыши и в появившемся меню выбрать
   - «**Add Fields**» для добавления компонента **TField** для существующего поля в наборе данных,
   - «**New Fields**» для добавления виртуального поля,
   - «**Add all Fields**» для добавления компонентов **TField** для всех полей в наборе данных.

3. Если выбрать пункт «**Add Fields**» откроется окно «**Add Fields**», в котором будут представлены все поля из набора данных. Следует выбрать необходимые поля и нажать кнопку «**OK**».



4. Для каждого из указанных полей будет создан компонент **TField**.
Если необходимо изменить свойства конкретного поля или написать обработчик для какого-либо события, необходимо в редакторе полей выбрать нужное поле и, используя инспектор объектов, установить значение в свойство или определить обработчик события.

Типы полей

Поля в таблицах баз данных различаются по типу. Соответственно, по типу различаются и компоненты **TField**. Класс **TField** есть родительский тип, определяющий базовые свойства и методы для своих потомков, типизированных полей. Иерархия компонентов- полей такова смотри в файле справки.

Покажем общие для всех этих типов свойства, методы и события.

Обращение к полям и их значениям

Следует различать обращение к полю и обращение к его значению. Рассмотрим способы обращения к полям.

К полю можно обратиться, несколькими способами:

1. Если полю соответствует компонент **TField**, то свойство *Name* данного компонента задает имя поля. По умолчанию имя компонента **TField** создается как результат сцепления имени компонента TTable или TQuery и собственно имени поля в таблице базы данных. Например, для поля FIO, работа с которым происходит через компонент **Table2**, по умолчанию будет создано имя **Table2FIO**. Тогда использовать поле можно следующим образом:

```
Table2FIO.AsString  : =  'Иванов';  // или
Table2FIO.Value  :=  'Иванов';
```

2. Используя метод **FieldByName ('ИмяПоля')** набора данных,
function FieldByName(const FieldName: string): TField;

```
Table2.FieldByName('FIO').AsString  :=  'Иванов';  //или
Table2.FieldByName('FIO').Value:=  'Иванов';
```

3. Используя свойство **Fields[индекс]** набора данных,
property Fields[Index: Integer]: TField;

Индекс является порядковым номером поля в определении ТБД. Отсчет идет от 0. Например, пусть поле Name определено в ТБД третьим по счету. Тогда его индекс равен 2 и использовать поле можно следующим образом:

Tablel.Fields[2].AsString := 'Иванов'; // или
Tablel.Fields[2].Value:= 'Иванов';

4. Используя свойство набора данных property **FieldValues[const FieldName: string]:** Variant;

Это свойство позволяет обращаться к полю через его имя, указываемое как содержимое параметра FieldName:

Table2.FieldValues['FIO'] := 'Иванов';

Поскольку свойство FieldValues принимается для набора данных по умолчанию, его имя при обращении к полю можно опускать:

Table2['Name']:= 'Иванов';

Более предпочтительным считается обращение к полю через его имя или через метод **FieldByName**, поскольку в этом случае мы обращаемся к конкретному полю по его имени. Следовательно, к несуществующему полю обратиться нельзя. Если обращаться к полю не по имени, а по индексу это может привести к обращению к несуществующему полю или полю другого типа, что вызовет ошибку.

Свойство **Index** компонента **TField** содержит порядковый номер поля:

- в таблице базы данных, если для компонента **TTable**, ассоциированного с данной таблицей, не создано ни одного компонента **TField**;
- в списке компонентов **TField**, относящихся к данному набору данных – для компонента **TTable**, если для него определен, хотя бы один компонент **TField**, и всегда - для компонента **TQuery**.

Порядок следования полей важен, когда содержимое набора данных визуализируется посредством компонента **TDBGrid**. В этом случае номер столбца в компоненте **TDBGrid** соответствует номеру поля в списке **TField**. И наоборот, если изменить порядок столбца в компоненте **TDBGrid** (например, "переместив" столбец на другое место), это приведет к изменению индекса и Fields[l] до перетаскивания будет относиться к другому полю, нежели Fields[l] после перетаскивания столбцов, поскольку свойство **Index** перетаскиваемого и некоторых других полей изменится.

Это относится как к случаю, когда **TField** для набора данных определены, так и к случаю, когда используются все поля таблицы базы данных. В последнем случае "перемещение" столбца в компоненте **TDBGrid** на новое место не изменит физического порядка следования полей в структуре таблицы базы данных; однако с логической точки зрения его индекс (порядковый номер) для данного набора данных изменится.

Обращение к значению поля

К значению поля можно обратиться при помощи свойств **Value** и **AsNNN**.
Свойство property **Value**: Variant возвращает значения следующих типов:

```
property Value:   Variant;     // Все компоненты
property Value:   string;      //TStringField,   TBlobField
property Value:   Longint;     //TAutoIncField,  TIntegerField, TSmallintField,TWordField
property Value:   Double;      //TBCDField,      TCurrencyField, TFloatField
property Value:   Boolean;     //TBooleanField
property Value:   TDateTime;   //TDateField,     TDateTimeField, TTimeField
```

Аналогичные значения возвращает свойство набора данных **FieldValues**.

Обращение к значению поля через свойство **AsNNN**. Существуют следующие свойства приведения типов полей:

property **AsBoolean**: Boolean;
property **AsCurrency**: Currency;
property **AsDateTime**: TDateTime;
property **AsFloat**: Double;
property **Aslnteger**: Integer;
property **AsString**: String;
property **AsVariant**: Variant;

Каждое из перечисленных свойств приводит значение поля к соответствующему типу данных, означенному в названии свойства. Например, если Table2Number - компонент TIntegerField (поле, хранящее целочисленные значения), для приведения его к типу String можно воспользоваться свойством

Editl.Text:= Table2Number.AsString;

Однако, тип поля должен быть совместимым с типом данных, к которому приводится значение поля. Например, если Table2Summa - компонент TFloatField (поле, хранящее вещественные значения), попытка привести его к несовместимому типу Boolean,

IF Table2Summa.AsBoolean THEN …

приведет к ошибке компиляции.

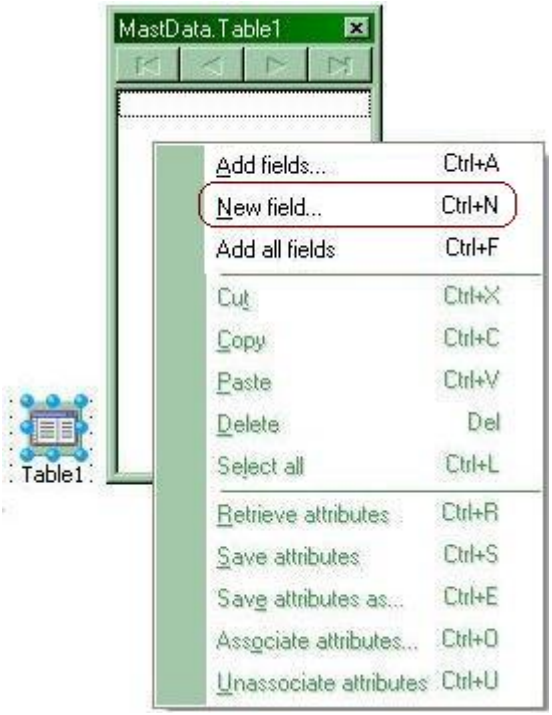
property AsBoolean: Boolean;	Числовые значения приводятся к типу **Boolean** если содержат 0 (False) или 1 (True). Символьные значения - если содержат в качестве первого символа "Y", "y", "T" или "t" (или "Yes" или "True"), и False во всех иных случаях.
property AsDateTime: TDateTime;	Свойство предназначено для приведения к типу **TDateTime** значений TDateField, TDateTimeField и TTimeField. Для подобного приведения можно использовать свойство **Value**, а также для приведения к типу TDateTime строковых значений, находящихся в соответствующем формате.
property AsFloat: Double;	Свойство служит для приведения к типу **Double** значений полей TFloatField, TBCDField и TCurrencyField, AsFloat. Для подобного приведения можно использовать свойство **Value**.
property Aslnteger: Longint;	Свойство служит для приведения к типу **Longint** полей типа TIntegerField, TSmallintField и TWordField. Для подобного приведения можно использовать свойство **Value**. Для полей типа TStringField преобразование к Longint выполняется, если оно возможно.
property AsCurrency: Currency;	Свойство служит для приведения к типу **Currency**.
property AsString: string;	Свойство служит для приведения к типу **String.**
property AsVariant: Variant;	Свойство служит для приведения к типу **Variant**.

1. К любому из ранее созданных приложений добавить возможность обращения к полям текущей записи различными вариантами (4). Для этого использовать кнопки (**Button**) при нажатии на которые значение поля будет отображено через компонент **Label.**

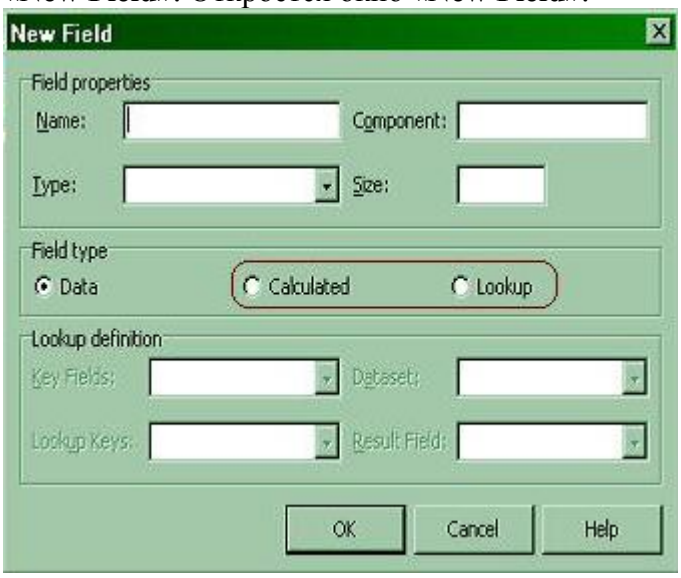
2. Для ранее созданного приложения придумать пример, который для текущей записи выполнит приведение типов.

Создание новых полей

Используя редактор полей можно создавать новые (виртуальные) поля в наборах данных.

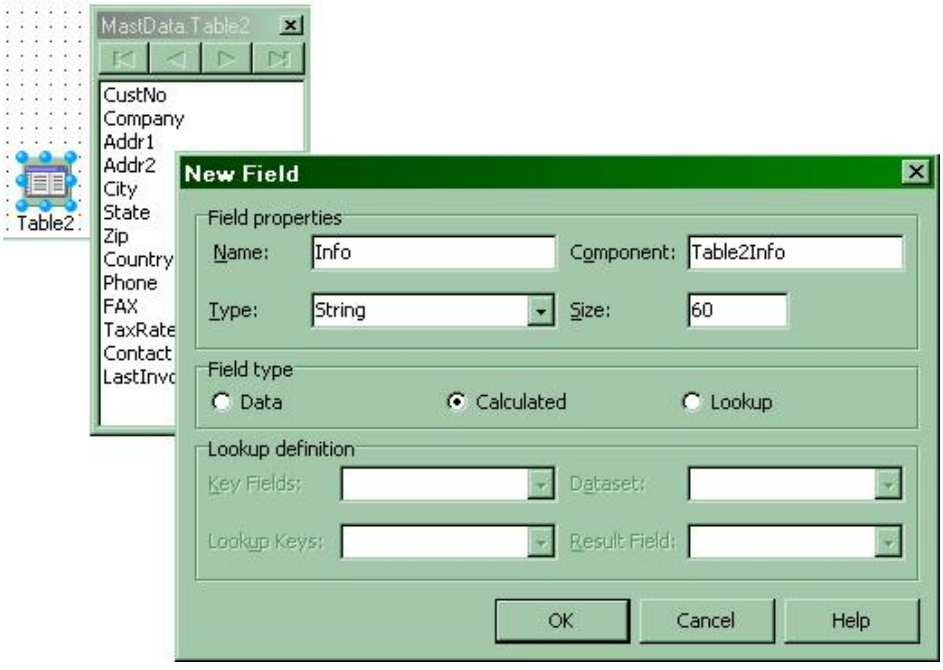


Для этого следует в окне редактора полей нажать правую кнопку мыши и выбрать из меню пункт «**New Field**». Откроется окно «**New Field**».

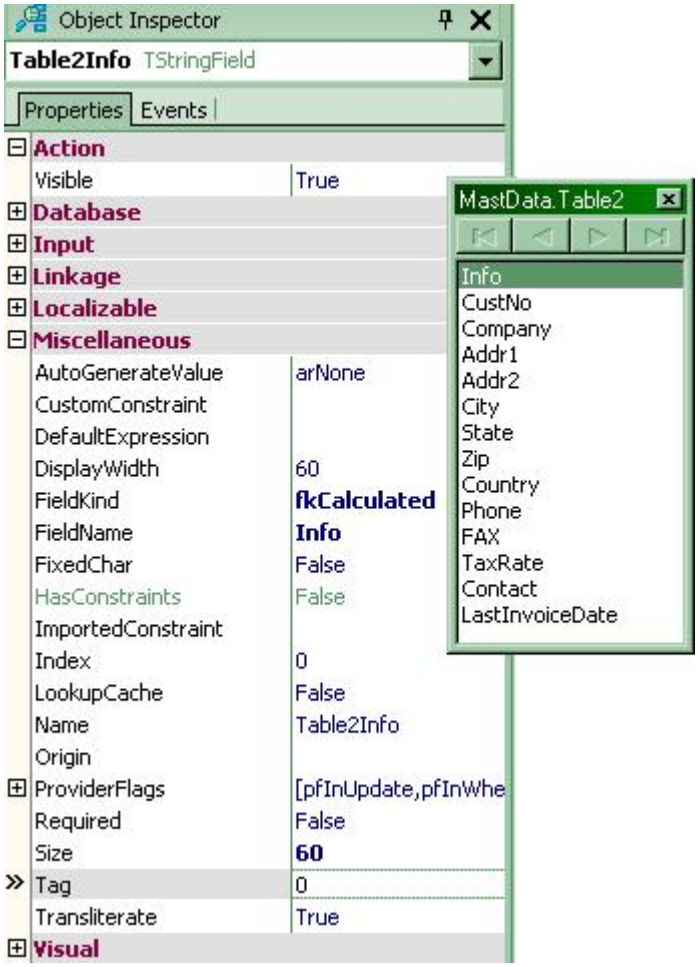


Вы можете создать вычисляемое поле, значение которого вычисляется по значениям других полей (**Calculated**), или создавать поле выбора данных (**Lookup**).

Создание вычисляемых полей



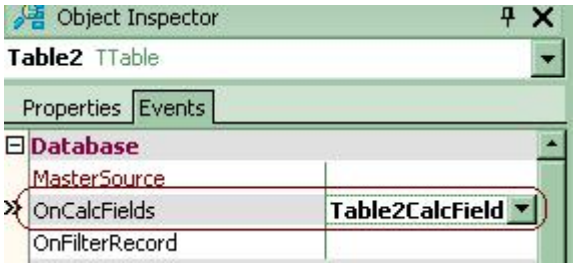
Чтобы создать вычисляемое поле, значение которого вычисляется или формируется по значениям других полей, в окне диалога «**New Field**» необходимо указать имя поля, его тип и для строковых полей длину, установить «**Field type**» в значение **Calculated**.



Для нового поля будет создан компонент **TField**, доступ к которому можно осуществлять из редактора полей.

Имя нового компонента **TField** создается как результат сцепления имени набора данных (компонента **TTable**) **Table2** и имени нового поля **Info**.

Для компонента набора данных (**Table2**), которому принадлежит вычисляемое поле, необходимо определить обработчик события **OnCalcFields**.

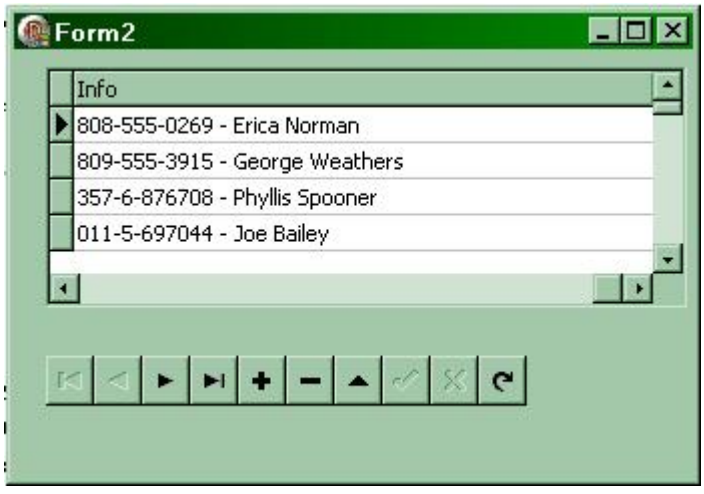


Процедура-обработчик события **OnCalcFields** содержит реализацию алгоритма вычисления значения вычисляемого поля или группы полей. Необходимо помнить, что в этом обработчике значение может быть присвоено только **вычисляемому полю**.

В новое поле поместим информацию о телефоне и персоне для контактов:

```
procedure TDataModule3.Table2CalcFields(DataSet: TDataSet);
begin
Table2Info.AsString:=Table2Phone.Value+' - '+ Table2Contact.AsString;
end;
```

Запустив приложение, увидим, какая информация попадет в созданное вычислимое поле **Info**.



Событие **OnCalcFields** возникает всякий раз, когда курсор (указатель записи) Перемещается в наборе данных от записи к записи (например, после выполнения методов **Next**, **Last** и т.д., или при движении по записям в TDBGrid вручную). Это событие возникает и при инициализации наборе данных (после открытия), а также после фильтрации записей в наборе данных.

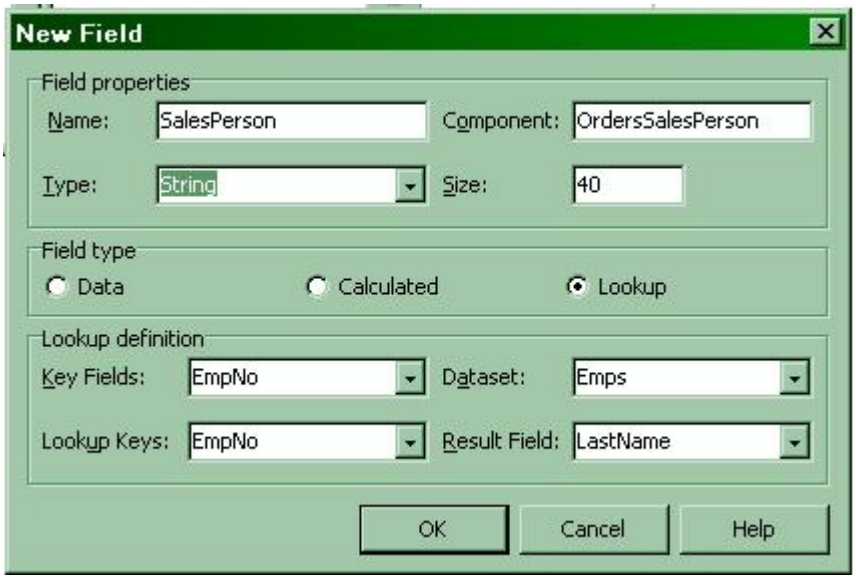
Кроме того, если свойство набора данных **AutoCalcFields** установлено в **True** событие **OnCalcFields** наступает также и при модификации значений невычисляемых полей в режимах **dslnsert** и **dsEdit** данного набора данных или набора данных, с ним связанного (когда установлены ограничения целостности для таблицы).

 3. Используя любую таблице из базы данных DBDEMOS создать приложение, в котором создать вычислимое поле.

Создание полей выбора данных (Lookup-полей)

Поля выбора данных одного набора данных содержат значения из другого набора данных, связанного по ключу с набором данных, к которому принадлежит поле выбора данных. Поле выбора данных всегда доступно только для чтения и не может быть одновременно полем выбора данных и вычисляемым полем.

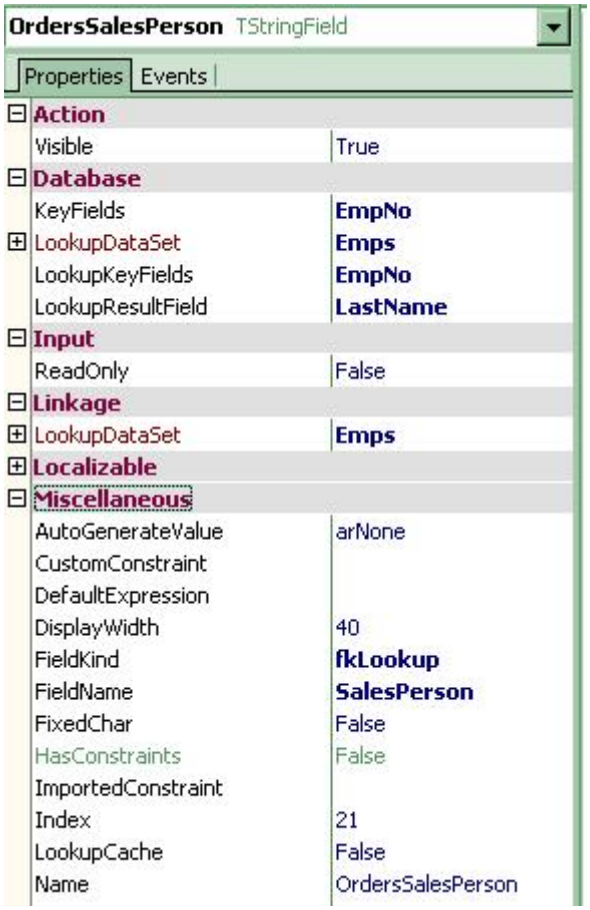
Отношение между наборами данных - «один-ко-многим» и реже «один-к-одному», т.е. на один вариант значения в наборе данных-источнике должно приходиться одно или несколько связанных значений в наборе данных, к которому принадлежит поле выбора.



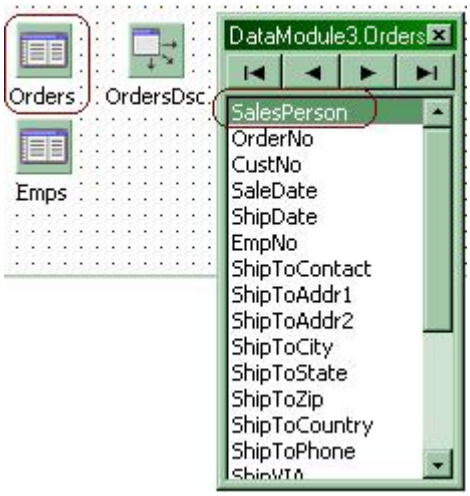
Чтобы создать поле **выбора данных (**типа **Lookup)**, значение которого берется из другого набора данных, в окне диалога «**New Field**» необходимо указать имя поля, его тип и для строковых полей длину, установить «**Field type**» в значение **Lookup**.
Устанавливаем значения свойств в разделе «**Lookup definition**».

DataSet - имя набора данных -источника значений для поля выбора данных;
Key Fields - поля набора данных владельца поля выбора данных. По этим полям набор данных владелец соединяется с источником значений поля выбора данных. Если имеется несколько полей связи, они перечисляются через точку с запятой;
Lookup Fields - индексные поля набора данных источника значений для поля выбора. По значениям этих индексных полей устанавливается связь набора-источника со значениями полей набора данных владельца поля выбора (они указаны в параметре Key Fields). Если в индексе имеется несколько полей, они перечисляются через точку с запятой;
Result Field - поле набора данных-источника, возвращаемое в качестве результата. Необходимо следить, чтобы тип вновь создаваемого поля и поля результата совпадали.

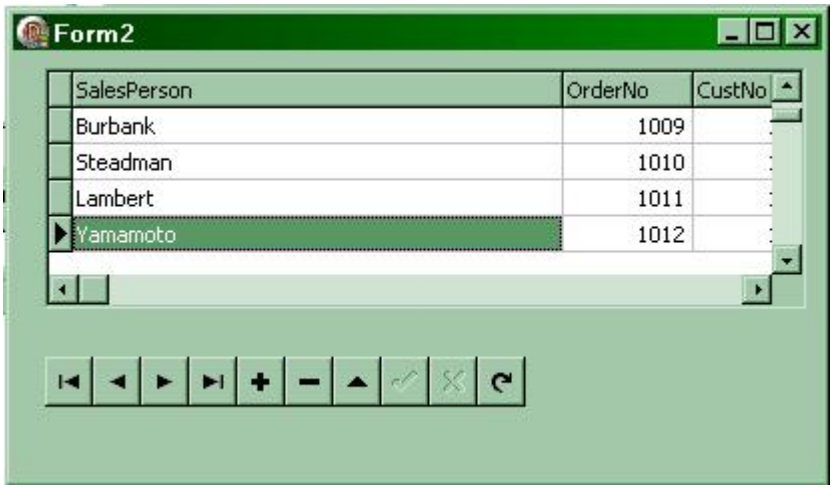
В инспекторе объектов будут установлены соответствующие свойства



Указанным выше параметрам редактора полей соответствуют следующие свойства компонента **TField**
property **LookupDataSet**: TDataSet;
property **KeyFields**: string;
property **LookupKeyFields**: string;
property **LookupResultField**: string;



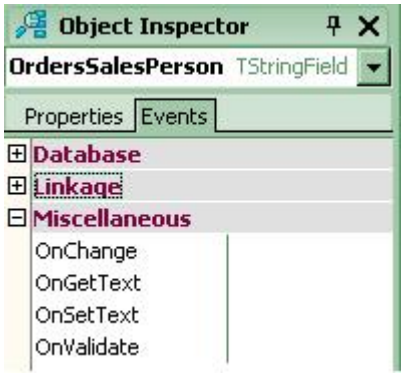
Разберем пример, приведенный выше. Имеются два набора данных **Orders** (заказы, orders.db) и **Emps** (Сотрудники, employee.db). Поле **EmpNo** в наборе данных **Orders** ссылается на ключевое поле **(EmpNo**) набора данных **Emps.** Было создано поле выбора данных **SalesPerson**, которое по полю **EmpNo** связывается с набором данных **Orders** и возвращает значение поля **LastName**.



Значения поля выбора данных **SalesPerson** набора данных **Orders** в работающем приложении выбираются из набора данных **Emps.**

4. Используя связанные таблицы из базы данных DBDEMOS создать приложение, в котором создать поле выбора данных.

События и свойства компонента TField



При работе с компонентом TField могут возникнуть события **OnSetText, OnValidate, OnChange** и **OnGetText.**

Событие **OnGetText** позволяет определить алгоритм форматирования значения в поле перед тем, как поле будет показано в визуальных компонентах, работающих с данными - TDBGrid, TDBEdit и т.д.

Параметры процедуры обработчика события:

Text - отформатированное значение, показываемое в столбце компонента TDBGrid, в TDBEdit или других визуальных компонентах, связанных с таблицей;

DisplayText - позволяет определить, произошло событие **OnGetText** при показе значения (значений) поля (**True**) или при модификации пользователем значения поля (**False**).

Например, при показе содержимого поля **Company** значение следует заключать в кавычки (хотя оно хранится без кавычек). Если пользователь захочет изменить значение поля в какой-либо записи, нужно показывать содержимое поля, представленное заглавными буквами.

```
procedure TForml.Table2Company GetText(Sender: TField; var Text: OpenString;
DisplayText: Boolean);
begin
IF DisplayText THEN
Text :=””’+ Table2Company.AsString ”””
ELSE Text := AnsiUpperCase(Table2Company.AsString);
end; // ”’’ одиночная кавычка двойная кавычка одиночная кавычка
```

Когда происходит показ некоторой записи из Table2 в TDBGrid, для нее возникает событие **OnGetText** и вызывается приведенный выше обработчик с **DisplayText = True**. В результате в компоненте TDBGrid весь столбец, соответствующий полю **Table2Company**, будет содержать значения в кавычках. Затем, если пользователь захочет в какой-либо записи изменить значение данного поля, оно будет выдано ему для редактирования заглавными буквами.
Если для поля определен обработчик события **OnGetText**, игнорируются режимы форматирования, определенные в свойствах **Display Format** и **Edit Mask** данного поля.

Свойство **property DisplayFormat: string**;
применяется для форматирования при показе полей типа TDateField, TDateTimeField, TIntegerField, TSmallintField, TTimeField, TWordField.

Для форматирования полей числового типа и типа даты могут применяться стандартные процедуры и функции **Str**, **DateTimeToStr**, **FloatToTextFmt**.

Свойство **property EditMask: string;**
служит для контроля правильности вводимых в поле значений. Ограничения накладываются при помощи формата. Если некоторый введенный символ не удовлетворяет маске, он не воспринимается. Для строковых полей значение данного свойства может использоваться для форматирования не только входных, но и выходных значений вместе со свойством DisplayText.

Свойство **го property EditMaskPtr: string;**
возвращает значение маски редактирования. Поскольку свойство доступно только на чтение, его следует использовать вместо **EditMask** в тех случаях, когда маска должна быть только прочитана. В итоге мы защищаемся от случайных изменений маски.

Свойство **property EditFormat: string;**
применяется для форматирования значений полей типа TIntegerField, TSmallintField, TWordField перед их редактированием. Форматирование выполняется функцией **FloatToTextFmt**.

Свойство **property Text: string;**
содержит строковое изображение значения поля в том виде, в котором но показывается в визуальном компоненте, когда набор данных находится в режиме редактирования (**dsEdit**) .

Свойство **DisplayText** содержит строковое изображение значения поля, когда набор данных находится не в режиме редактирования.

Проверка введенного в поле значения

Для проверки введенного в поле значения могут быть использованы свойство **IsNull** и обработчики событий **OnSetText, OnValidate**, **OnChange**.

Свойство **property IsNull:** Boolean;
возвращает **True**, если поле содержит пустое значение.

Проверить введенное в поле значение на его соответствие некоторым ограничениям или условиям можно в обработчике события **OnValidate**. Это событие наступает при изменении значения поля либо вручную, либо программно. Событие наступает до выполнения метода **Post**, который запоминает изменения в базе данных. Таким образом, если полю присвоено неверное значение, выполнение метода **Post** можно предотвратить, выполнив метод **Abort** или возбудив исключительную ситуацию **raise Exception.Create**. Этот подход контроля правильности значений называется ориентированным на поля**.**

Существует и другой подход, ориентированный на записи. Он состоит в том, что в структуре таблицы при ее определении описываются ограничения на значения, которые может принимать данное поле. В этом случае контроль правильности ведется автоматически.

Пример обработчика события **OnValidate**. Запретим полю **Table2Company** содержать символ «#». Обратите внимание на то, как формируется имя обработчика события для компонента **TField**, к имени компонента добавляется имя события без начального «**On**».

```
procedure TForml.Table2Company Validate(Sender: TField);
begin
IF POS('#',Table2Company.AsString) > 0 THEN
begin
  ShowMessage('Обнаружен символ #!');
  Abort;
  end;
end;
```

Или

```
procedure TForml. Table2Company Validate (Sender: TField);
begin
IF POS ('#',Table2Company.AsString) > 0 THEN
raise Exception.Create( 'Неверное значение');
end;
```

Если символ «#» содержится в значении, присвоенном полю, метод **Abort** или принудительно возбужденная исключительная ситуация не позволят выполниться методу **Post** и запись с неверным полем не будет физически записана в базу данных. Набор данных не изменится.

Для проверки введенного значения может быть использовано и другое событие, **OnSetText**. Подобно событию **OnValidate**, оно возникает при изменении значения поля. Однако на момент события новое значение полю не присвоено и, если этого не сделать программно в обработчике события, сделано никогда не будет.

Пусть в поле **Table2PRICE** типа TFloatField было изменено значение с 10 на 360. Если новое значение не должно превышать 150. Тогда обработчик события будет выглядеть так:

```
procedure TForml.Table2PRICE SetText(Sender: TField; const Text: String);
var Tmp : Real;
begin
Tmp :.= StrToFloat(Text) ;
IF Tmp > 150 THEN
ShowMessage('Ошибочное значение')
ELSE
Table2PRiCE.Value := Tmp;
end;
```

Для этого примера обработчик события **OnValidate** будет выглядеть так:

```
procedure TForml.Table2PRICE Validate(Sender: TField);
begin
    IF Table2PRICE.Value > 150 THEN
    begin ShowMessage('Ошибочное значение'); Abort;
    end;
end;
```

Как видно, особенность события **OnSetText** состоит в том, что в обработчик события передается константа-параметр **Text**, содержащая в текстовом виде новое значение, назначенное полю, в то время как действительное значение поля остается без изменения.

Третье событие, **OnChange**, может быть использовано для тех же целей, что и **OnValidate**:

```
procedure TForml.Table2PRICEChange(Sender: TField);
begin
IF Table2PRICE.Value > 100 THEN5
raise Exception.Create('Ошибочное значение');
end;
```

Порядок вызова обработчиков событий OnSetText, OnValidate, OnChange

При изменении значения поля вызываются обработчики событий в следующей последовательности:

1. **OnSetText**;
2. **OnValidate**;
3. **OnChange**.

Это важно, когда действия по проверке правильности нового значения поля сосредоточены не в одном обработчике, а распределены в обработчиках разных событий.

Относительно события **OnSetText** известно, что если полю присвоено ошибочное значение, то нет нужды выполнять метод **Abort** или **возбуждать исключительную ситуацию** для предотвращения занесения этой записи в базу данных (поскольку новое значение в поле в этом случае еще не занесено). Однако если поле удовлетворяет допустимым критериям, в него программно нужно записать введенное пользователем новое значение (передаваемое в обработчик как параметр const **Text**: String).

В обработчиках событий **OnValidate** и **OnChange**, наоборот, в этом случае необходимо выполнять метод **Abort** или **возбуждать исключительную ситуацию** для предотвращения занесения этой записи в базу данных (поскольку новое значение в поле в этом случае уже занесено).

Однако следует помнить, что событие **OnChange** возникает только после события **OnValidate**. Поэтому, обработчик события **OnChange** может быть и не вызван, если обработчик **OnValidate** выполняет метод **Abort** или **возбуждает исключительную ситуацию.**

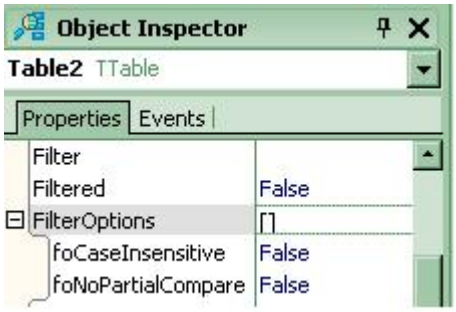
 5. Используя таблицы из базы данных DBDEMOS создать приложение, в котором для компонентов **TField** написать обработчики событий **OnSetText, OnValidate, OnChange** и **OnGetText.**

Лабораторная работа № 4

Фильтрация и поиск записей в наборах данных

Delphi позволяет проводить фильтрацию и поиск записей по индексированным полям и по любым полям (индексированные или неиндексированные).

Фильтрация записей по любым полям



Свойство **Filter** позволяет задать критерий фильтрации. В этом случае набор данных будет отфильтрован, как только его свойство **Filtered** станет равным **True**. Синтаксис описания критерия похож на синтаксис секции WHERE SQL-запроса с тем исключением, что имена переменных программы указывать нельзя, можно указывать имена полей и литералы (явно заданные значения); можно использовать обычные операции отношения и логические операторы **AND**, **NOT** и **OR**, например:

([Doljnost] = 'профессор') AND ([TabNum] > 6001)

Строку критерия фильтрации можно ввести во время выполнения программы или на этапе конструирования формы.

Пусть на форме находятся компоненты Table2 (TTable), Edit2 (TEdit), CheckBox2 (TCheckBox) и другие для визуализации набора данных.

Обработчик события **OnChecked** компонента CheckBox1 позволяет установить критерий фильтрации для набора данных Table2, считывая значения для фильтра из поля редактирования:

```
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
Table2.Filter := Edit2.Text;
Table2.Filtered := CheckBox2.Checked;
end;
```

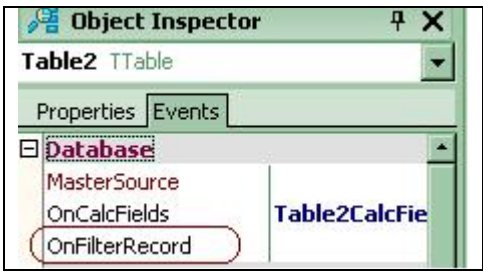
Если в наборе данных имеются поля **Doljnost** (должность) и **TabNum** (табельный номер) и в поле Edit2 во время выполнения программы ввести строку

([Doljnost] = 'профессор') AND ([TabNum] > 6001)

Тогда набор данных будет отфильтрован, если поставить галочку в компоненте CheckBox2.

С помощью свойства
type TFilterOption = (foCaseInsensitive, foNoPartialCompare);
property **FilterOptions**: TFilterOptions;
программист может определить дополнительные условия фильтрации строковых полей:
foCaseInsensitive - фильтрация производится без учета разницы в высоте букв;
foNoPartialCompare - поиск производится на точное соответствие.

 1. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором создать пример фильтрации записей с помощью Свойства **Filter**.



Событие **OnFilterRecord** возникает при установке значения **True** в свойство **Filtered**. Обработчик события имеет два параметра: имя фильтруемого набора данных и переменную **Accept**, в которую программа должна поместить **True**, если текущая запись удовлетворяет критерию фильтрации.

В отличие от критерия в строке **Filtered**, ограниченного рамками синтаксиса условного выражения, критерий, реализуемый в обработчике события **OnFilterRecord**, определяется синтаксисом Object Pascal и может реализовывать сложные алгоритмы фильтрации. Однако следует помнить о том, что в обработчике **OnFilterRecord** последовательно перебираются все записи набора данных. Это делает использование обработчика **OnFilterRecord** предпочтительным для фильтрации небольших объемов записей и сильно ограничивает его применение при больших объемах.
Всякий раз, когда приложение обрабатывает событие **OnFilterRecord**, набор данных переводится из состояния **dsBrowse** в состояние **dsFilter**. Это предотвращает модификацию набора данных во время фильтрации. После завершения текущего вызова обработчика события **OnFilterRecord** набор данных переводится в состояние **dsBrowse**.
Чтобы создать набор данных из тех записей таблицы «**Сотрудники**», в которых поле **Doljnost** содержит значение «доцент», можно использовать такой обработчик:

```
procedure TForml.Table2FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
Accept := DataSet ['Doljnost' ] = 'профессор';
end;
```

Чтобы отфильтровать таблицу «**Сотрудники**» по условию «Отобрать всех сотрудников, у кого табельные номера (поле TabNum) больше значения, вводимого пользователем в поле редактирования Edit1, и в поле FIO есть подстрока символов, вводимых пользователем в Edit2»:

```
procedure TForml.Table2FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
Accept := (DataSet['TabNum'] > Edit1.Text)) and (Pos(Edit2.Text,DataSet['FIO']) > 0);
end;
```

Если в строке **Filter** и в обработчике события **OnFilterRecord** заданы разные критерии фильтрации, выполняются оба.

 2. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором создать пример фильтрации записей с помощью события **OnFilterRecord**.

Методы **FindFirst**, **FindLast**, **FindNext**, **FindPrior** позволяют перемещаться в неотфильтрованном наборе данных (**Filtered = False**) между записями, удовлетворяющими условию фильтрации. Условие фильтрации задается событием **OnFilterRecord** и/или свойством **Filter**. Действие методов таково: они ненадолго переводят набор данных в отфильтрованное состояние (**Filtered = True**) без визуализации этой фильтрации в TDBGrid или другом подобном визуальном компоненте, находят соответствующую запись и переводят набор данных обратно в неотфильтрованное состояние (**Filtered = False**). Если искомая запись найдена, методы возвращают **True**, в противном случае - **False**.
В отличие от этих методов, методы **First**, **Last**, **Next** и **Prior** учитывают критерий фильтрации, только если набор данных находится в отфильтрованном состоянии (свойство Filtred содержит True).

 3. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать навигацию в неотфильтрованном наборе данных между записями, удовлетворяющими фильтру.

Поиск записей в наборах данных по любым полям

Метод Locate

Метод **Locate** ищет первую запись, удовлетворяющую критерию поиска, и если такая запись найдена, делает ее текущей. В этом случае в качестве результата возвращается **True**. Если запись не найдена, возвращается **False**.
function **Locate**(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean;

Список **KeyFields** указывает поле или несколько полей, по которым ведется поиск. В случае нескольких поисковых полей их названия разделяются точкой с запятой. Критерии поиска задаются в вариантном массиве **KeyValues** так, что i-е значение в **KeyValues** ставится в соответствие i-му полю в **KeyFields**. Options позволяет указать необязательные значения режимов поиска:
loCaseInsensitive - поиск ведется без учета высоты букв, т.е. если в KeyValues указано «принтер», а в некоторой записи в данном поле встретилось «Принтер» или «ПРИНТЕР», запись считается удовлетворяющей условию поиска;
loPartialKey - запись считается удовлетворяющей условию поиска, если она содержит часть поискового контекста; например, удовлетворяющими контексту «Ма» будут признаны записи со значениями в искомом поле «Машин», «Макаров» и т.д.
Например,

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver', 'P', '408-431-1000']),
[loPartialKey]);
```

или

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;
```

Locate производит поиск по любому полю; поле или поля, по которым производится поиск, могут не только не входить в текущий индекс, но и не быть индексными вообще.

В случае если поля поиска входят в какой-либо индекс, **Locate** использует этот индекс при поиске. Если искомые поля входят в несколько индексов, трудно сказать, какой из них будет использован. При поиске по полям, не входящим ни в один индекс, применяются фильтры BDE.

 4. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать поиск записей в наборах данных с использованием метода **Locate.**

Метод Lookup

Метод **Lookup** находит запись, удовлетворяющую условию, но не делает ее текущей, а возвращает значения некоторых полей этой записи. Тип результата -**Variant** или вариантный массив. Независимо от успеха поиска записи, указатель текущей записи в НД не изменяется.

function **Lookup**(const **KeyFields**: String; const **KeyValues**: Variant; const **ResultFields**: String): **Variant**;

В отличие от **Locate**, **Lookup** осуществляет поиск только на точное соответствие критерия поиска и значения полей записи. В **KeyFields** указывается список полей, по которым необходимо осуществить поиск. При наличии в этом списке более чем одного поля соседние поля разделяются точкой с запятой. **KeyValues** указывает поисковые значения полей, список которых содержится в **KeyFields**.

Если имеется несколько поисковых полей, каждому i-му полю в списке **KeyFields** ставится в соответствие i-ое значение в списке **KeyValues**. При наличии одного поля его поисковое значение можно указывать в качестве **KeyValues** непосредственно; в случае нескольких полей - их необходимо приводить к типу вариантного массива при помощи **VarArrayOf**.

В качестве поисковых полей можно указывать поля как входящие в какой-либо индекс, так и не входящие в него; тип текущего индекса не имеет значения. Если поисковые поля входят в какие-либо индексы, их использование производится автоматически; в противном случае используются фильтры BDE.

Результат можно проверить с помощью функции **VarType**, которая возвращает одно из следующихзначений:

Значение, возвращаемое VarType	**Содержимое**
varEmpty	пустое (неназначенное) значение
varNull	значение Null
varSmallint	тип Smallint
varInteger	тип Integer
varSingle	тип Single
varDouble	тип double
varCurrency	тип Currency
varDate	тип TDateTime
varVariant	variant
varShortInt	тип ShortInt
varByte	тип Byte
varWord	тип Word
varInt64	тип Int64
varString	Ссылка на динамически выделенную строку

Если в результате поиска запись не найдена, метод **Lookup** возвращает **Null**, что можно проверить с помощью оператора
if **VarType**(LookupResults) = **varNull** then ...

В противном случае **Lookup** возвращает из этой записи значения полей, список которых содержит **ResultFields**. При этом размерность результата зависит от того, сколько результирующих полей указано в **ResultFields**:

одно поле - результатом будет значение соответствующего типа или Null, если поле в найденной записи содержит пустое значение;

несколько полей - результатом будет вариантный массив, число элементов в котором меньше или равно числу результирующих полей (некоторые поля найденной записи могут содержать пустые значения).

```
var
  LookupResults: Variant;
begin
with CustTable do
```

```
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
'Company; Addr1; Addr2; State; Zip');
end;
```

или

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
  C: Integer;
  A: String;
begin
  V := Table1.Lookup('Company;State',  VarArrayOf(['Blue Sports', 'OR']),
    'CustNo;Addr1');
if not (VarType(V) in [varNull]) then begin
    C := V[0];
    A := V[1];
    ShowMessage(IntToStr(C) + #10 + A);
end
else
    ShowMessage('Search unsuccessful!');
end;
```

Пример, когда метод **Lookup** возвращает одно результирующее поле (результат - значение типа Variant).

```
procedure TFormX. ButtonOneClick(Sender: TObject);
var
LookupResults: Variant;
begin
LookupResults := Table2.Lookup('FIO', Edit2.Text, 'UchStepen'); // поиск

// Проверяем,   содержит ли результат пустое значение или Null:
case VarType(LookupResults) of
varEmpty : Label2.Caption := 'Пустой результат';
varNull  : Label2.Caption := 'Запись не найдена';
else
          Label2.Caption := LookupResults;
 end;   //case
end;
```

Пример, когда метод **Lookup** возвращает несколько результирующих полей (результат - вариантный массив). Если переменная типа **Variant** является вариантным массивом, функция **VarIsArray** возвращает **True**, верхнюю и нижнюю границы массива можно определить при помощи функций **VarArrayLowBound** и **VarArrayHighBound**, а тип каждого элемента - с помощью функции **VarType**.

```
procedure TFormX.ButtonManyClick(Sender: TObject);
var
LookupResults: Variant;
begin
// Ищем  запись:
LookupResults := Table2.Lookup('FIO',Edit2.Text, 'TabNum;Doljnost;UchStepen');
```

```
// Проверяем:  результат  - вариантный массив?
 if VarIsArray(LookupResults) then
begin
Label1.Caption := LookupResults[0];
if LookupResults[1] <> Null then
Label2.Caption := LookupResults[1];
if LookupResults[2] <> Null then
Label3.Caption := LookupResults[2];
end
else
// результат - не вариантный массив,   а  единичное значение
case VarType(LookupResults) of
          varEmpty: Label1.Caption := 'Пустой результат';
          varNull : Label1.Caption := 'Запись не найдена';
end; //case
 end;
```

Если запись не найдена, VarType(LookupResults) возвращает значение varNull. Если поиск по какой-либо причине не был произведен, VarType (LookupResults) возвращает значение varEmpty. Если какое-либо из полей, чьи значения возвращаются в результате поиска в вариантном массиве, содержит пустое значение, соответствующий элемент вариантного массива также будет содержать пустое значение (Null). В этом случае обращение к нему возбудит исключительную ситуацию, поэтому нужна предварительная проверка.

 5. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать поиск записей в наборе данных с использованием метода **Lookup**.

Лабораторная работа № 5

Фильтрация записей в наборах данных по индексированным полям

Для фильтрации записей компонент **TTable** имеет следующие методы:
SetRangeStart - устанавливает нижнюю границу фильтра;
EditRangeEnd - устанавливает верхнюю границу фильтра;
ApplyRange - осуществляет фильтрацию записей в TTable;
SetRange - имеет тот же эффект, что и последовательное выполнение методов **SetRangeStart**, **SetRangeEnd** и **ApplyRang**e. В качестве параметра используются массивы констант, каждый из которых содержит значения ключевых полей.
Заметим, что фильтрация методами ApplyRange/SetRange должна проводиться по ключевым полям. По умолчанию берется текущий индекс, определяемый свойством **IndexName** или свойством **IndexFieldNames** компонента **TTable**. Если значения этих свойств не установлены, по умолчанию используется главный индекс таблицы базы данных. Поэтому, если нужно использовать индекс, отличный от главного, необходимо явно переустановить значение свойства **IndexName** (имя текущего индекса) или **IndexFieldNames** (список полей текущего индекса).

Метод **SetRange**
procedure **SetRange**(const StartValues, EndValues: array of const);
показывает в наборе данных только те записи, индексные поля которых лежат в диапазоне [StartValues..EndValues].

Пусть на форме находятся компоненты Table2 (TTable), Edit2 (TEdit), CheckBox2 (TCheckBox) и другие для визуализации набора данных. Пусть текущий индекс построен по полю **NumGr** (номер группы).

Обработчик события **OnClick** компонента CheckBox2 позволяет установить критерий фильтрации для набора данных Table2, считывая значения для фильтра из поля редактирования Edit2:

Отфильтруем записи таким образом, чтобы показывались записи только с определенным номером группы. Если CheckBox2 отмечен (CheckBox2.Checked = True), то производится фильтрация по номеру группы, введенному в Edit2, в противном случае показываются все записи из таблицы.

```
procedure TFormA.CheckBox2Click(Sender: TObject);
var
NumGrTmp: Integer;
begin
if CheckBox2.Checked then
begin .
  NumGrTmp := StrToInt(Edit2.Text);
  //фильтрация записей в НД
  with Table2 do
  begin
  CancelRange;
  SetRange([NumGrTmp],[NumGrTmp]);
  end; {with}
end
else
//отмена фильтрации
  Table2.CancelRange;
end;
```

В отфильтрованном наборе данных показываются только те записи, индексное поле текущего индекса у которых (поле **NumGr**) имеет значение, лежащее в заданном диапазоне.

Если бы мы захотели, чтобы в наборе данных фильтровались записи из нескольких групп, то нам следовало бы добавить в форму второй компонент Edit3, в котором вводился бы номер конечной группы - в то время как в Edit2 вводился бы номер начальной группы:

```
procedure TFormA.CheckBox2Click(Sender: TObject);
var
NumGrTmp1,NumGrTmp2: Integer;
begin
if CheckBox2.Checked then
begin
NumGrTmp1 := StrToInt(Edit2.Text);
NumGrTmp2 := StrToInt(Edit3.Text);
//фильтрация записей
with Table2 do
begin
CancelRange;
SetRange([NumGrTmpl] , [NumGrTmp2]);
end; //with
end
else
```

```
//отмена фильтрации
Table2.CancelRange;
end;
```

Методы **SetRangeStart**, **SetRangeEnd**, **ApplyRange** являются альтернативой методу **SetRange**, который объединяет в себе функциональность трех указанных методов.

В частности, рассмотренная в предыдущем примере фильтрация по начальному и конечному номеру группы может быть реализована следующим образом:

```
procedure TFormA.CheckBox2Click(Sender: TObject);
var
NumGrTmp1,NumGrTmp2: Integer;
begin
if CheckBox2.Checked then
begin
with Table2 do
begin
CancelRange;
SetRangeStart;
  Table2.FieldByName('NumGr').Aslnteger:= NumGrTmp1;
SetRangeEnd;
  Table2.FieldByName('NumGr').Aslnteger:= NumGrTmp2;
ApplyRange;
end;  //with
 end
else
Table2.CancelRange;
end;
```

Метод **CancelRange** служит для отмены предыдущих условий фильтрации. Если предыдущую фильтрацию не отменить, возможно, следующие фильтрации принесут не такой результат, которого вы ожидаете.

Изменим пример, представленный выше. Сначала удалим из него вызов метода **CancelRange**. Пусть сначала набор данных сортируется по наименованию товара (поле Tovar). Разместим в форме группу зависимых переключателей **RadioGroup2**, позволяющую переключать текущие индексы компонента Table2:

```
procedure TFormA.RadioGroup2Click(Sender: TObject);
begin
with RadioGroup2 do
begin
case ItemIndex of

0: Table2.IndexFieldNames := 'Tovar'; // Текущий индекс - по полю 'Tovar'
                                      // (наименование товара)
1: Table2.IndexFieldNames := 'NumGr'; // Текущий индекс - по полю 'NumGr'
                                      // (номер группы товара)
end;  //case
 end;  //with
end;
```

Когда пользователь хочет произвести фильтрацию по номеру группы, он нажимает кнопку «Фильтровать» (**Button2**), для которой реализован следующий обработчик нажатия:

```
procedure TFormA.Button2Click(Sender:   TObject);
var
NumGrTmpl,NumGrTmp2:   Integer;
begin
with Table2  do
begin
// Отмечаем строку текущего выбора  в  RadioGroup2:
RadioGroup2.ItemIndex   :=   1;
// Смена  текущего индекса:
IndexFieldNames   :=   'NumGr';
SetRange([NumGrTmpl],[NumGrTmp2]);
end;    //with
end;
```

Чтобы исключить ошибок при фильтрации при переключении индексов, следует перед новой фильтрацией (по какому бы то ни было индексу) отменять результаты предыдущей фильтрации методом **CancelRange**:

```
procedure TFormA.Button2Click(Sender: TObject);
var
NumGrTmp1,NumGrTmp2: Integer;
begin
with Table2 do
begin
CancelRange;
RadioGroup2.ItemIndex := 1;
// Смена   текущего индекса:
IndexFieldNames := 'NumGr';
SetRange([NumGrTmp1],[NumGrTmp2]);
end; //with
end;
```

Методы **EditRangeStart**, **EditRangeEnd** предназначены для смены условий фильтрации, установленных ранее с использованием соответственно методов **SetRangeStart** и **SetRangeEnd**. Сама фильтрация в этом случае выполняется методом **ApplyRange**. Преимущества их использования ясны не всегда.

```
procedure TFormA.Button2Click(Sender: TObject);
var
NumGrTmpl,NumGrTmp2: Integer;
const
Num: Integer = 0;
begin
inc (Num).;
with Table2 do
begin
if Num = 1 then
begin
SetRangeStart;
FieldByName('NumGr').Aslnteger:= NumGrTmp1;
SetRangeEnd;
FieldByName('NumGr').Aslnteger := NumGrTmp2;
ApplyRange;
```

```
end
else begin
EditRangeStart;
FieldByName('NumGr').Aslnteger:= NumGrTmp1;
EditRangeEnd;
FieldByName('NumGr').Aslnteger := NumGrTmp2;
ApplyRange;
end
end; //with
end;
```

Однако результат будет ошибочным. Указанный код будет правильно работать только в случае, когда индекс по **NumGr** является принятым по умолчанию и в процессе работы не изменяется (представим, что в показанном выше примере мы удалили переключатели RadioGroup1 для выбора текущего индекса). Однако в этом случае правильно работает и такой код

```
procedure TFormA.Button2Click(Sender: TObject);
var
NumGrTmpl,NumGrTmp2: Integer;
begin
with Table2 do
begin
SetRangeStart;
FieldByName('NumGr').Aslnteger:= NumGrTmpl;
SetRangeEnd;
FieldByName('NumGr').Aslnteger := NumGrTmp2;
ApplyRange;
end; //with
end;
```

 1. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором создать примеры фильтрации записей в наборе данных с использованием методов

SetRange
SetRangeStart, SetRangeEnd, ApplyRange
EditRangeStart, EditRangeEnd, ApplyRange

Свойство **KeyExclusive** применяется для фильтрации записей в компоненте **TTable** с использованием методов **SetRangeStart**, **SetRangeEnd** и **EditRangeStart**, **EditRangeEnd**.

Свойство **KeyExclusive** влияет на включение в отфильтрованный набор данных записей, у которых индексные поля содержат граничные значения диапазона фильтрации. **KeyExclusive** включается и отключается отдельно для начального и конечного условия фильтрации.

Если в свойство **KeyExclusive** для данной границы диапазона фильтрации (верхней или нижней) установлено значение **False**, записи, содержащие в индексном поле (полях) значение, указанное в качестве данной границы диапазона, включаются в отфильтрованный набор данных, в противном случае не включаются. По умолчанию применяется значение **False**.

```
…
with Table2 do
begin
CancelRange;
SetRangeStart;
KeyExclusive := True;
```

```
FieldByName('NumGr').AsInteger:= NumGrTmpl;
SetRangeEnd;
FieldByName('NumGr').AsInteger  := NumGrTmp2;
ApplyRange;
end;   //with
```

Фильтрация **по составному индексу** (индекс состоит из нескольких полей) осуществляется

- при использовании метода **SetRange**. Значения полей должны перечисляться через запятую внутри квадратных скобок.
- при использовании **SetRangeStart**, **SetRangeEnd** и т.д. Значение каждого поля должно устанавливаться явно.

```
// Набор данных отсортирован по индексу, состоящему из полей NumGr и Tovar
procedure  TFormA.CheckBox2Click(Sender:  TObject);
NumGrTmp:  Integer;
TovarTmp:  String;
begin
if CheckBox2.Checked then
begin
NumGrTmp  :=  StrToInt(Edit1.Text);
TovarTmp  :=  Edit2.Text;
//фильтрация  записей в  НД
with  Table2  do
begin
CancelRange;
SetRange([NumGrTmp,TovarTmp],[NumGrTmp,'яя ' ]) ;
end;   //
 end
else
//отмена  фильтрации
Table2.CancelRange;
end;
```

Реализацию выборки обеспечивает оператор

SetRange([NumGrTmp,TovarTmp],[NumGrTmp,'яя']);

Если требуется показывать в наборе данных все записи группы, начинающиеся со значения, введенного в поле **Edit2**, то в качестве значения товара в конечном условии фильтрации нужно объявить максимально возможное значение, которое только может встретиться в качестве названия товара. Поскольку строчные буквы имеют большие коды, чем заглавные, и название товара не может начинаться с 'яя', эти символы вполне могут использоваться как верхний ограничитель наименования товара.

Если введено значение для группы товара в поле **Edit1** и не введено наименование товара в поле **Edit2,** то в отфильтрованный набор данных попадут все товары данной группы.

Если введено значение для группы товара в поле **Edit1** и введено наименование товара в поле **Edit2,** то в отфильтрованный набор данных попадут товары данной группы, у которых наименование больше или равно значению, введенному в поле **Edit2**.

 2. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать фильтрацию по составному индексу.

Для случаев сортировки по символьным полям полезно выполнять фильтрацию **по частичному соответствию** индексного поля (полей) условиям фильтрации.

```
//таблица отсортирована по наименованию товара Tovar
procedure TFormA.CheckBox2Click(Sender: TObject);
begin
if CheckBox2.Checked then
begin
with Table2 do
begin
CancelRange;
SetRange([Edit2.Text],['яя']); end; //with
end
else
Table2.CancelRange;
end;
```

В примере в наборе данных будут показаны все записи с названием товара, равным или большим значению, указанному в поле **Edit2**.

Если изменить вызов метода **SetRange** на следующий:

SetRange([Edit2.Text],[Edit2.Text + 'яя'])>

то отфильтрованный набор данных попадут только записи, начинающиеся с введенного в поле **Edit2** фрагмента названия товара.

 3. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать фильтрацию по частичному соответствию индексных полей.

В качестве условий фильтрации могут быть заданы не все поля текущего индекса, а только ведущее поле или группа ведущих полей (**фильтрация по части составного индекса**). В частности, для предыдущего примера можно указать в качестве текущего индекс **Tovar;NumGr**. Тогда при применении метода

SetRange([Edit2.Text],[Edit2.Text + 'яя']);

в квадратных скобках в качестве начального и конечного условия фильтрации указаны не два значения поля, а одно, фильтрацию следует проводить на предмет соответствия ведущего поля индекса (в нашем случае **Tovar**) заданному поисковым значением (в нашем случае начальное значение - Edit2.Text; конечное значение – Edit2 Text + 'яя').

 4. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать фильтрацию по части составного индекса.

Отметим, что рассмотренный механизм фильтрации позволяет отфильтровывать только те записи, у которых значения ключевых полей больше или равны нижней границе и меньше или равны верхней границе фильтрации. Если условие фильтрации сложное, то вместо работы с компонентом **TTable** создавайте запросы и работайте с компонентом **TQuery**.

Лабораторная работа № 6

Поиск записей в наборах данных по индексированным полям

Для поиска записей в наборах данных в компоненте **TTable** по индексированным полям применяются следующие методы:

FindKey- ищет запись, точно удовлетворяющую условиям в списке значений. Существует метод **GoToKey**, выполняющий аналогичные действия.

FindNearest - ищет запись, наиболее полно удовлетворяющую условиям в списке значений. Существует метод **GoToNearest**, выполняющий аналогичные действия.

Установка значений для поиска

Состав полей, используемых для идентификации нужной записи при поиске в наборе данных, определяется текущим индексом. Поэтому в качестве текущего нужно установить индекс, построенный по полям, значения которых и будут отыскиваться. Для установки текущего индекса используют свойства **IndexFieldNames** или **IndexName**.

Для индексов, в состав которых входит более одного поля, должны указываться значения всех полей, входящих в индекс, или значения полей старших уровней вложенности.

Точный поиск

Для точного поиска (поиска на точное соответствие) применяется метод **FindKey**. Он пытается отыскать в наборе данных запись, у которой индексные поля соответствуют значениям, указанным в параметре обращения. Если такая запись найдена, метод **FindKey** возвращает **True** и указатель текущей записи в наборе данных (курсор наборе данных) устанавливается на эту запись, т.е. она становится текущей. Если найдена группа записей, отвечающая условию, текущей становится логически первая из них. Если запись не найдена, курсор набора данных не перемещается и метод возвращает **False**.

Пример. Пусть имеется набор данных с полями **NumGr** (номер группы), **NN** (номенклатурный номер товара), **Tovar** (наименование товара);

NumGr	**NN**	**Tovar**
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Предположим, что поисковое значение NumGr вводится в поле **Edit1**, а NN – в поле **Edit2**. Тогда обработчик нажатия клавиши поиска FindButton может выглядеть так (для простоты в обработчике не контролируется правильность ввода в компонентах Edit1, Edit2):

```
procedure TFormA.FindButtonClick (Sender: TObject) ;
var
GrTmp, NNTmp: Integer;
begin
GrTmp:=StrToInt(Editl.Text) ;
NNTmp:=StrToInt(Edit2.Text) ;
// поиск записи
if not Table2.FindKey([GrTmp,NNTmp]) then
ShowMessage('Нет товара с такой группой и номером!');
end;
```

Пусть в Edit1 введено 12 и в Edit2 - 4. Тогда указатель переместится на запись, у которой значение NumGr равно 12 и значение поля NN равно 4:

NumGr	**NN**	**Tovar**
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир // Перемещение указателя текущей записи
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Для точного поиска в Delphi имеется группа методов **SetKey, EditKey, GotoKey**, которые должны выполняться вместе, и которые по функциональности аналогичны методу **FindKey**. Использование их менее удобно: сначала набор данных переводится в состояние **dsSetKey** (методом **SetKey** или, если он уже применялся для данного индекса, **EditKey**), затем присваиваются поисковые значения полям и выполняется метод **GotoKey**. После этого набор данных переходит в состояние **dsBrowse**. Результат выполнения аналогичен результату, возвращаемому методом **FindKey**. Например, код

```
if not  Table2.FindKey([GrTmp,NNTmp])   then
    ShowMessage('Нет  товара  с  такой группой и номером!');
```

эквивалентен более громоздкому коду с применением *GotoKey:*

```
Table2.SetKey; Table2NumGr.Value := GrTmp;
Table2NN.Value := NNTmp;
if not Table2.GotoKey then
                ShowMessage('Нет товара с такой группой и номером!');
```

 1. . Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать точный поиск записей наборах данных по индексированным полям с использованием методов **FindKey, SetKey, EditKey, GotoKey** (индекс по одному полю, индекс по нескольким полям).

Неточный поиск

Неточный поиск (поиск на неточное, приблизительное соответствие) осуществляется методом **FindNearest**. Он пытается отыскать в наборе данных запись, у которой индексные поля соответствуют указанным значениям. Если такая запись найдена, указатель текущей записи в наборе данных перемещается на нее или на следующую за ней запись, в зависимости от значения свойства **KeyExclusive**. Если **KeyExclusive = False** (по умолчанию), указатель текущей записи перемещается на нее. Если **KeyExclusive = True**, указатель текущей записи перемещается на следующую запись.

Если запись не найдена, указатель текущей записи всегда перемещается на ближайшую запись с большим значением индекса.

В приводимых ниже примерах подразумевается KeyExclusive = False.

Предположим, что поисковое значение NumGr вводится в поле Editl, а NN - в поле Edit2. Тогда обработчик нажатия кнопки для поиска **FindButton** может выглядеть так:

```
procedure  TFormA.FindButtonClick(Sender:   TObject);
var
```

```
GrTmp, NNTmp: Integer;
begin
GrTmp := StrToInt(Editl.Text);
NNTmp := StrToInt(Edit2.Text);

// поиск записей
Table2.FindNearest([GrTmp,NNTmp]);
end;
```

Если в поле **Edit1** введено значение 12 и в поле **Edit2** введено значение 4. Тогда указатель переместится на запись «**Кефир**»:

NumGr	NN	Tovar
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Если в поле **Edit1** введено значение 12 и в поле введено значение **Edit2** – 0, то указатель переместится на запись с большим значением индекса. Наращивание индекса ведется по внутреннему полю (если оно есть) и только затем - по полю более высокого приоритета. В нашем случае внутреннее поле в индексе - поле NN и будет выбрана запись «**Вода минеральная**»:

NumGr	NN	Tovar
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Если в поле **Edit1** введено значение 50, в поле **Edit2** введено значение 0 (или что-либо другое, для данного состояния набора данных это неважно), записи с той же группой (50) и большим значением номенклатурного номера нет. Поэтому указатель записи перемещается на запись с большим номером группы:

NumGr	NN	Tovar
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Заметим, что теоретически можно опускать в условиях поиска значение внешнего поля индекса (в нашем случае поля **NumGr**). Например, можно задать в поле **Edit1** значение 0, а в поле **Edit2** - 1. Однако, вопреки ожиданиям, курсор набора данных не встанет на первую запись с номенклатурным номером 1 для первой попавшейся группы, а встанет на первую запись с группой, превышающей 0, т.е. на логически первую запись в наборе данных при сортировке по полям **NumGr, NN**:

NumGr	NN	Tovar
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Существует более громоздкая альтернатива методу **FindNearest** - выполнение группы методов **SetKey, EditKey, GoToNearest** и заполнение полей поисковыми значениями Выполнение метода

Table2.**FindNearest**([GrTmp,NNTmp]);

может быть заменено эквивалентным по последствиям кодом

Table2.**SetKey**;
Table2NumGr.Value := GrTmp;
Table2NN.Value := NNTmp;
Table2.**GotoNearest**;

Если нужно осуществить поиск по индексу, отличному от текущего индекса, необходимо:

- сохранить список текущих индексных полей в строковой переменной;
- заменить список текущих индексных полей набора данных на необходимый;
- осуществить поиск;
- восстановить список текущих индексных полей набора данных из строковой переменной.

Пусть текущая сортировка в наборе данных осуществляется по имени товара (т.е. индекс построен по полю Tovar, Table2.**IndexFieldNames** = 'Tovar') и текущей является вторая логическая запись:

NumGr	NN	Tovar
12	1	Вода минеральная
12	26	Груши
12	4	Кефир
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая
1	10	Макароны

Напишем обработчик события нажатия кнопки **FindButton**:

```
procedure TFormA.FindButtonClick(Sender: TObject);
var
  GrTmp, NNTmp: LongInt;
  OldIndexFieldNames: String;
  begin
  OldIndexFieldNames := Table2.IndexFieldNames;
  Table2.IndexFieldNames := 'NumGr;NN';
  //…
  Table2.FindNearest([GrTmp, NNTmp] ) ;

    Table2.IndexFieldNames := OldIndexFieldNames;
```

end;

Если в поле **Edit1** ввести значение 100, а в поле **Edit2** ввести значение 0, то после нажатия кнопки **FindButton** указатель переместится на запись **Конфеты «Ласточка»**:

NumGr	NN	Tovar
12	1	Вода минеральная
12	26	Груши
12	4	Кефир
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая
1	10	Макароны

Перед выполнением последней строки обработчика

Table2.IndexFieldNames := OldIndexFieldNames;

набор данных будет иметь вид:

NumGr	NN	Tovar
1	10	Макароны
12	1	Вода минеральная
12	4	Кефир
12	26	Груши
100	1	Конфеты «Ласточка»
100	10	Крупа гречневая

Восстановление исходного индекса в последней строке приведет к изменению логического следования записей в наборе данных, но текущая запись останется **прежней.** Это происходит из-за того, что действует правило: простое изменение сортировки в наборе данных, если оно не сопровождалось изменением условий фильтрации записей, не влечет за собой изменения местоположения курсора НД.

 2. Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать неточный поиск записей в наборах данных по индексированным полям с использованием методов:

FindNearest

SetKey, EditKey, GoToNearest

Инкрементальный локатор

Под локатором будем понимать механизм поиска (точного или приблизительного) записей в наборе данных с последующим позиционированием на них курсора компонента **TTable**. Для реализации локатора обычно применяется один или несколько компонентов **TEdit** для ввода условий поиска и кнопка **TButton**, обработчик события нажатия которой и реализует поиск.

Описанные выше обработчики события нажатия кнопки FindButton реализуют локаторы. Однако вне рассмотрения остался еще один режим: по вводу каждого символа в TEdit курсор переходит на запись, ближе всего лежащую к искомой. Чем больше введено символов, тем ближе курсор к искомой записи. Такой локатор называется **инкрементальным.**

Пусть необходимо реализовать инкрементальный локатор для уточняющего поиска записи по названию товара. Если описанный ранее набор данных отсортирован по индексному полю **Tovar** и текущая запись в нем - логически первая, тогда набор данных имеет следующий вид:

Tovar	NumGr	NN
Вода минеральная	12	1
Груши	12	26
Кефир	12	4
Конфеты «Ласточка»	100	1
Крупа гречневая	100	10
Макароны	1	10

Будем вводить значения для поиска в поле **Edit3**. Напишем обработчик события, возникающего при любом изменении значения в **Edit3** (**OnChange**):

```
procedure TFormA.Edit3Change(Sender:   TObject);
begin
    Table2.FindNearest ( [Edit3.Text] ) ;
end;
```

Если нужно отыскать запись с наименованием товара «Крупа гречневая», то при использовании описанного механизма инкрементального локатора необязательно вводить это название полностью. Курсор будет, приближаться к искомой записи по мере ввода символов в поле **Edit3**. Введем в **Edit3** символ «К». Тогда

Table2.**FindNearest** ([Edit3.Text]) ;

означает

Table2.FindNearest(['К']);

В результате курсор переместится на 1-ю запись, имеющую в поле **Tovar** значение, большее строки «К»:

Tovar	NumGr	NN
Вода минеральная	12	1
Груши	12	26
Кефир	12	4
Конфеты «Ласточка»	100	1
Крупа гречневая	100	10
Макароны	1	10

Вводим в поле **Edit3** следующий символ, «р» (Edit3.Text = Кр). В результате курсор переместится на 1-ю запись, имеющую в поле **Tovar** значение, большее или равное «Кр»:

Tovar	NumGr	NN
Вода минеральная	12	1
Груши	12	26
Кефир	12	4
Конфеты «Ласточка»	100	1
Крупа гречневая	100	10
Макароны	1	10

Это и есть искомая запись. Заметим, что применение инкрементальных локаторов возможно не только для символьных полей, но и для числовых.

 3. Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать **Инкрементальный локатор.**

Поиск по части текущего индекса

Можно осуществлять поиск по части индексных полей. Для этого необходимо с помощью свойства **KeyFieldCount** указать, сколько начальных полей индекса будут использоваться при поиске.

Установка значения свойства **KeyFieldCount** актуальна только в случае использования методов **GotoKey** и **GotoNearest**.

Установим текущий индекс по полям **Doljnost** и **FIO**. Организуем поиск по Doljnost и по Doljnost + FIO. Условия для поиска будем вводить в поля **Edit1** (должность) и **Edit2** (ФИО).

По кнопке **OneFieldFindButton** будем выполнять поиск по одному полю, а по кнопке **FullKeyFindButton** будем выполнять поиск по всему индексу (по полям **Doljnost** и **FIO**).

```
procedure TFormA .OneFieldFindButtonClick (Sender: TObject);
begin
// поиск по одному полю индекса
with Table2 do
begin
   SetKey;
KeyFieldCount := 1;
      Table2Doljnost.Value := Edit1.Text;
      GoToNearest;
   end; // with
end;

procedure TForml.FullKeyFindButtonClick(Sender: TObject);
begin
// Поиск  по двум полям индекса
with Table2 do
begin
SetKey;
KeyFieldCount := 2;
Table2Doljnost.Value := Editl.Text;
Table2FIO.Value := Edit2.Text;
   GoToNearest;
end; // with
end;
```

Поля по которым осуществляется поиск должны быть в полном индексе непрерывным, т.е. при индексе из 4 полей можно осуществить поиск по 1-му, 2-му, 3-му полям одновременно и нельзя по 1-му, 4-му, 6-му полям.

При использовании методов **FindKey** и **FindNearest** необходимости в использовании **KeyFieldCount** нет. Для поиска по частичному соответствию достаточно указать в списке часть полей данного индекса:

```
// Поиск по одному полю из двух: Table2.FindNearest([Editl.Text]);
// Поиск по двум полям из двух:
Table2.FindNearest([Editl.Text, Edit2.Text]);
```

 4. Используя таблицы из базы данных DBDEMOS создать приложение, в котором организовать **Поиск по части текущего индекса.**

Лабораторная работа № 7

Работа с запросами. Статические запросы. Параметрические запросы.

Компонент набора данных **TQuery** находится на странице «**BDE**» палитры компонентов «**Tool Palette**».

Компонент TQuery

Компонент **TQuery (**невизуальный, реализован классом **TQuery,** является потомком класса **TDBDataSet).** Этот компонент позволяет определить набор данных на основе нескольких таблиц с помощью **SQL**- запроса. Использование этого компонента связано со знанием языка **SQL**. Компоненты **TQuery** и **TTable** имеют много общих признаков. Для показа данных из компонента **TQuery** в визуальных компонентах используется компонент **TDatasource**.

Необходимая для работы база данных задается свойством **DatabaseName**, в котором можно указать зарегистрированный в BDE псевдоним базы данных или локальное имя базы данных компонента **TDatabase.** Запрос, на основе которого создается набор данных, определяется свойством **SQL** (типа TStrings). Объект **TStrings** представляет собой список строк. Тип данных TStrings имеет методы добавления строк, их загрузки из текстового файла и обмена данными с другим объектом TStrings.

При помощи методов **Open** (для запросов на выборку) или **ExecSQL** (для запросов действий) запрос открывается или запускается на выполнение. Метод **Close** закрывает набор данных. О состоянии набора данных можно судить по значению свойства **Active** (**True** – набор данных открыт).

Свойство **Local** определяет расположение таблиц (**True** – локальные таблицы, **False** – таблицы на **SQL**-сервере). Используется только для чтения значений.

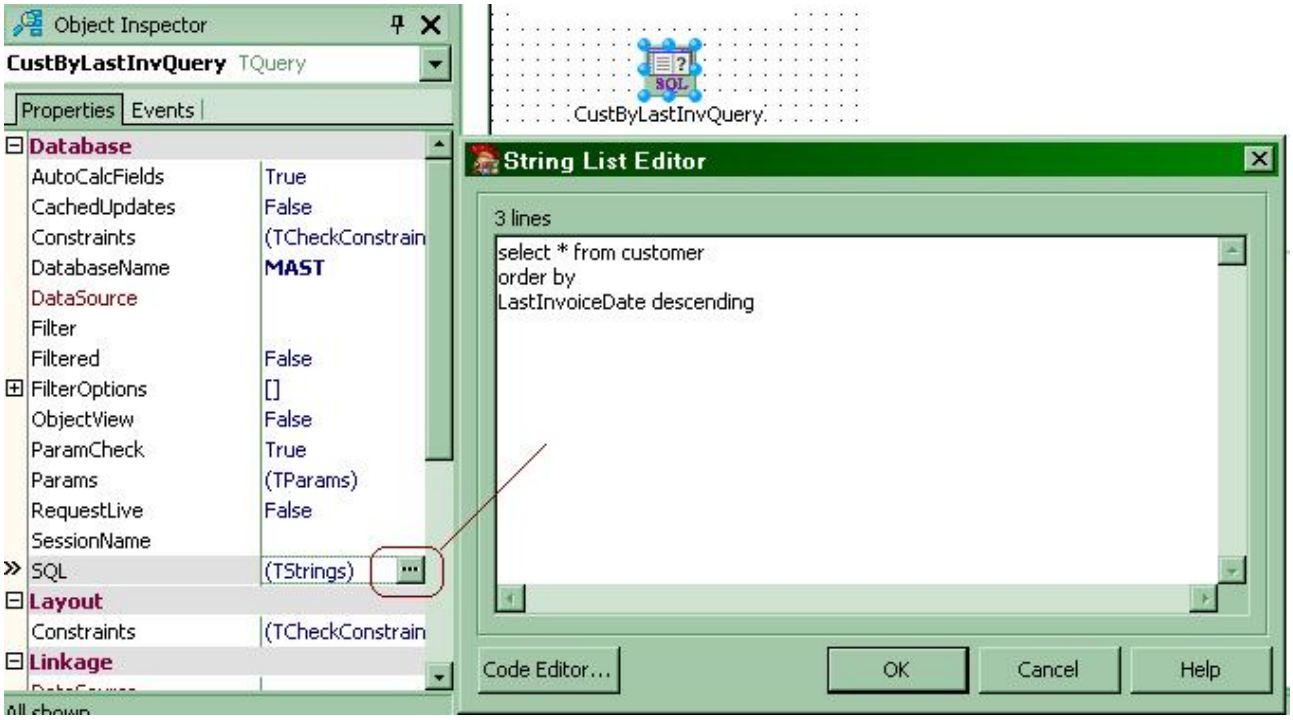
Свойство **RequestLive** определяет возможность изменять набор данных, полученный в результате выполнения запроса (по умолчанию имеет значение **False**). Однако на запросы, которые можно изменять наложено много ограничений: запрос должен быть основан на одной таблице, запрос не должен использовать сортировку и агрегатные функции. У остальных запросов результат доступен только для чтения независимо от значения данного свойства.

Свойство **UniDirectional** определяет направление перемещения курсора по набору данных, полученному в результате выполнения запроса.

Запросы могут быть статическими или изменяемыми (динамические). Статические запросы не изменяются во время выполнения программы. Изменяемые запросы могут быть параметрическими (т.е. меняться в зависимости от значения введенных параметров) или формируемыми.

Свойство **Params** содержит список параметров динамического запроса. Если запрос параметрический, то используя свойство **DataSource** (типа TDataSource) можно определить источник данных, значения полей которого используются как параметры для параметрического запроса. Метод **Prepare** посылает запрос в BDE для проверки синтаксиса и оптимизации. Рекомендуется выполнять для динамических запросов.

Чтобы работать с компонентом **TQuery** его следует поместить в модуль данных или на форму. Используя инспектор объектов указать через свойство **DatabaseName** базу данных, с которой будет работать приложение. Далее в строке свойства **SQL** нажать на кнопку с тремя точками «…». Откроется окно «**String List Editor**» для ввода **SQL**-строки запроса. По кнопке «**OK**» сохраните введенную строку в свойстве **SQL.**



Для визуализации данных из запроса (компонент **TQuery)** в приложении используются те же компоненты, что и для визуализации данных из таблицы (компонент **TTable**). Не забудьте открыть набор данных **TQuery.**

1. Изучение приложения **mastapp.**

1.1.Открыть приложение mastapp (файл проекта **mastapp.bdsproj**) из папки

c:\Program Files\Borland**BDS\4.0\Demos\DelphiWin32\VCLWin32\Db\MastApp**

1.2. В режиме редактирования (конструктора) открыть форму **MastData** (меню «**View**»-> «**Forms**», или с панели инструментов «**View Forms**» (Shift F12))

1.3. Открыть окно инспектора объектов (меню «**View**»->«**Object Inspector**»)

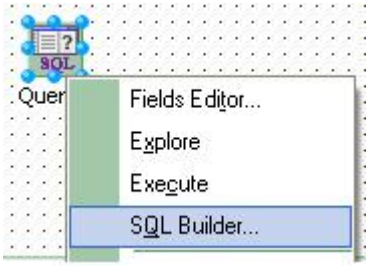
1.4. Исследовать свойства (Properties) и события (Events) объектов (компонентов), находящихся на форме. (**TDatabase, TDataSource, TQuery**). Обратить внимание на связи между этими объектами. Проверить используют ли компоненты **TQuery** редакторы полей для создания виртуальных полей.

Выполнение статических запросов

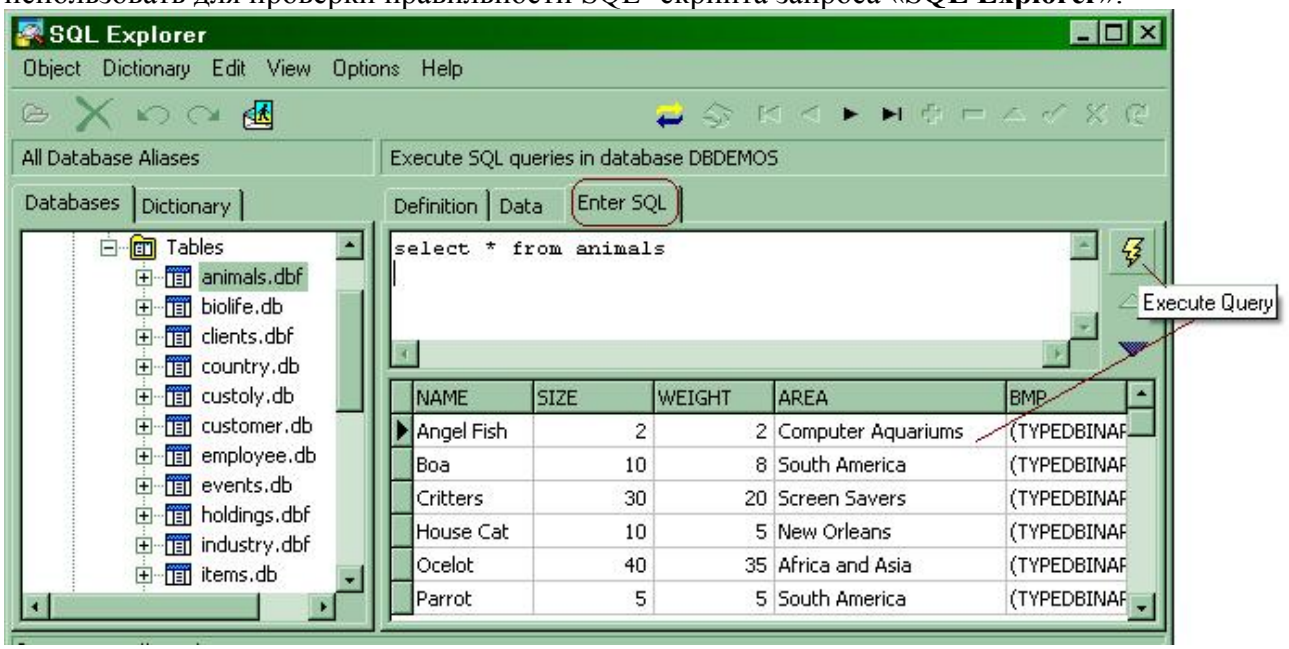
2. Создать главную форму «**Main**», на которую будем добавлять кнопки. Эти кнопки будут открывать формы, работающие с компонентом **TQuery** (запрос)**.**

1. На форму «**Main**» добавить кнопку «**Связь нескольких таблиц**».
2. Создать форму «**Chld1**», в которой расположить компоненты **TQuery, TDataSource, TDBGrid, TDBNavigator.**
3.

Если вы работаете в Delphi7 и более ранних версиях, а также в CodeGear Delphi, то по правой кнопке мыши на компоненте **TQuery** можно выбрать пункт меню «**SQL Builder**» для построения запроса.



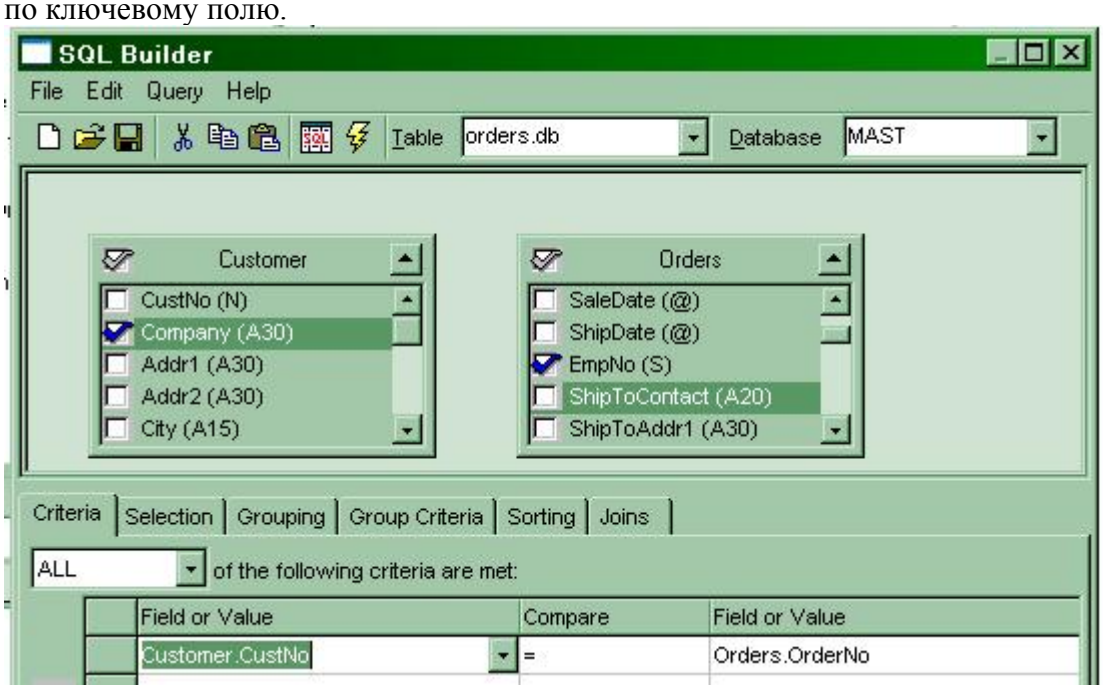
В BDS запрос придется писать самим (если «**SQL Builder**» не установлен), и использовать для проверки правильности SQL- скрипта запроса «**SQL Explorer**».



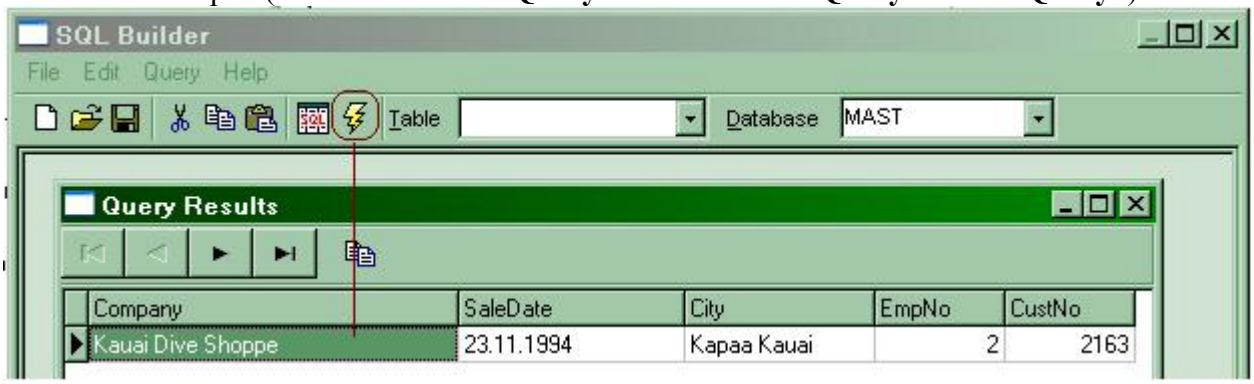
Работать будем с таблицами **customer.db**, **orders.db**, имеющими Alias **DBDEMOS**.

4. Из таблицы **customer.db** выбрать поля **Company**, **Country**, **Addr1**, **State**. Из таблицы **orders.db** выбрать поля **OrderNo**, **Saledate**, **PaymentMethod**, **ItemsTotal**, **TaxRate**, **AmountPaid**.

5. Если «**SQL Builder**» установлен, то перейти на закладку «**Criteria**» и установить связь по ключевому полю.

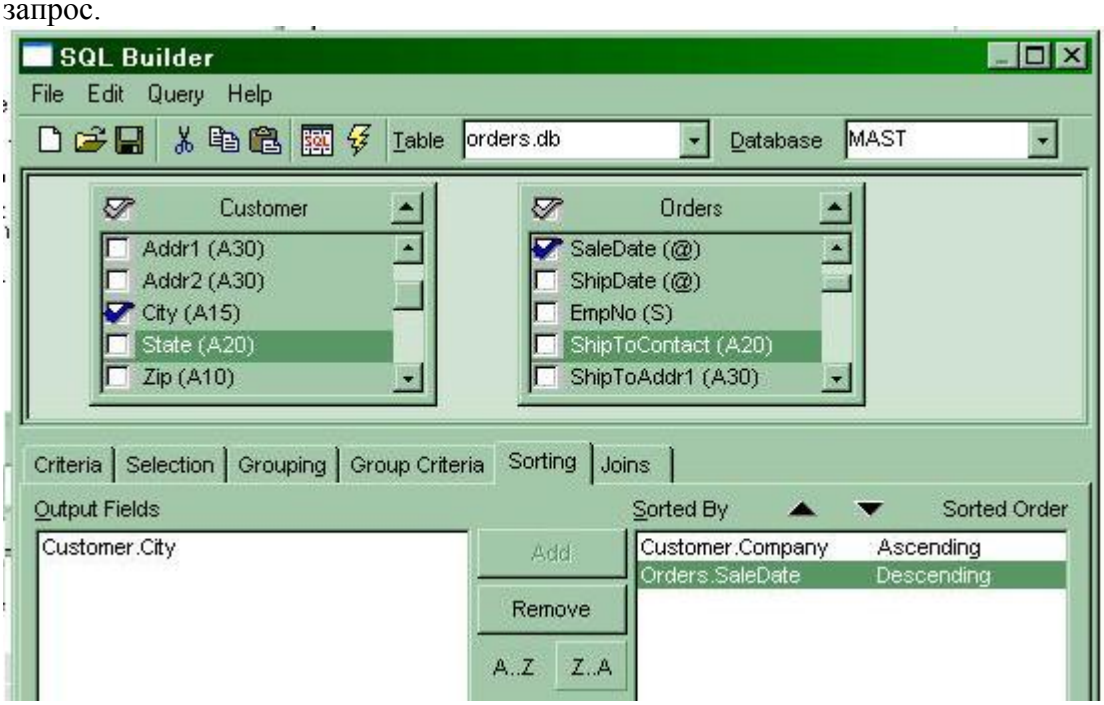


Выполнить запрос (кнопка «**Execute Query**» или из меню «**Query**»->«**RunQuery**»).

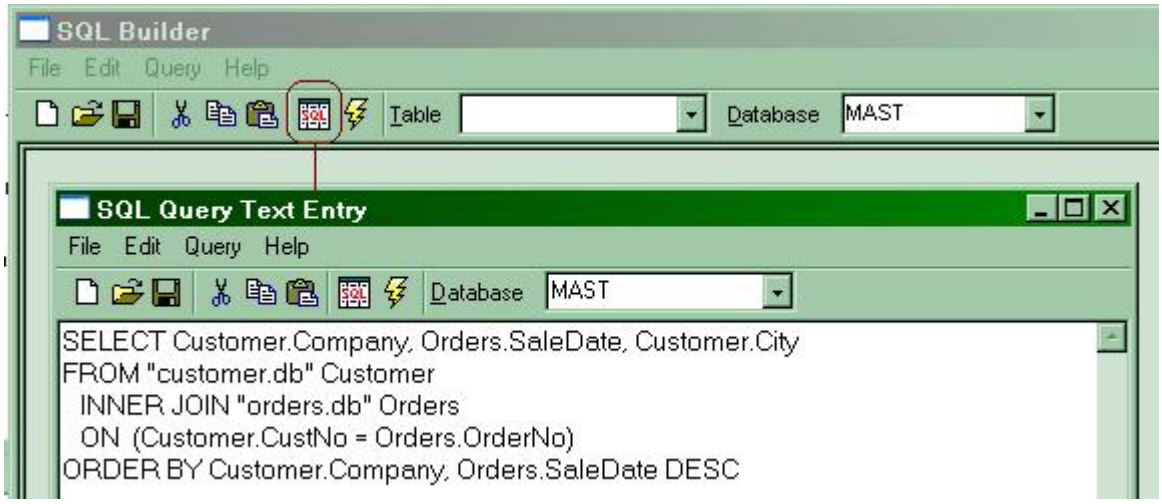


Объединение таблиц в запросе можно организовать, если перейти на закладку «**Joins**» и установить связь по ключевым полям.

6. Если «**SQL Builder**» установлен, то перейти на закладку «**Sorting**» и отсортировать данные по полю **Saledate** по возрастанию, а по полю **Company** по убыванию. Выполнить запрос.



7. Посмотреть текст SQL запроса (кнопка «**Show and Edit SQL**» или из меню «**Query**»->«**Show SQL**»). Определить по тексту запроса какие ключевые слова из запроса отвечают за все действия, которые вы проделали над таблицами.



8. Закрыть «**SQL Builder**», сохранив изменения в запросе. Текст запроса запишется в свойство **SQL** компонента **TQuery.** Имя базы данных **(**Alias) попадет в свойство **DatabaseName**.
9. Связать все компоненты на форме между собой (аналогично как с компонентом **TTable**).
10. Чтобы открыть запрос нужно
    Свойство **Active** компонента **TQuery** установить в **true**

Или

    Использовать метод **Open** компонента **TQuery** (ИмяЗапроса.Open**)**

Чтобы закрыть запрос нужно
    Свойство **Active** компонента **TQuery** установить в **false**

Или

    Использовать метод **Close** компонента **TQuery** (ИмяЗапроса**.**Close)

11. По кнопке «**Связь нескольких таблиц**» вызвать форму «**Chld1**».
12. Откомпилировать и запустить программу.

Мы создали статический запрос. Данные из запроса в процессе выполнения программы изменяться не могут.

Возможные проблемы:

1. В форме не получается сослаться на невизуальные компоненты из модуля данных.

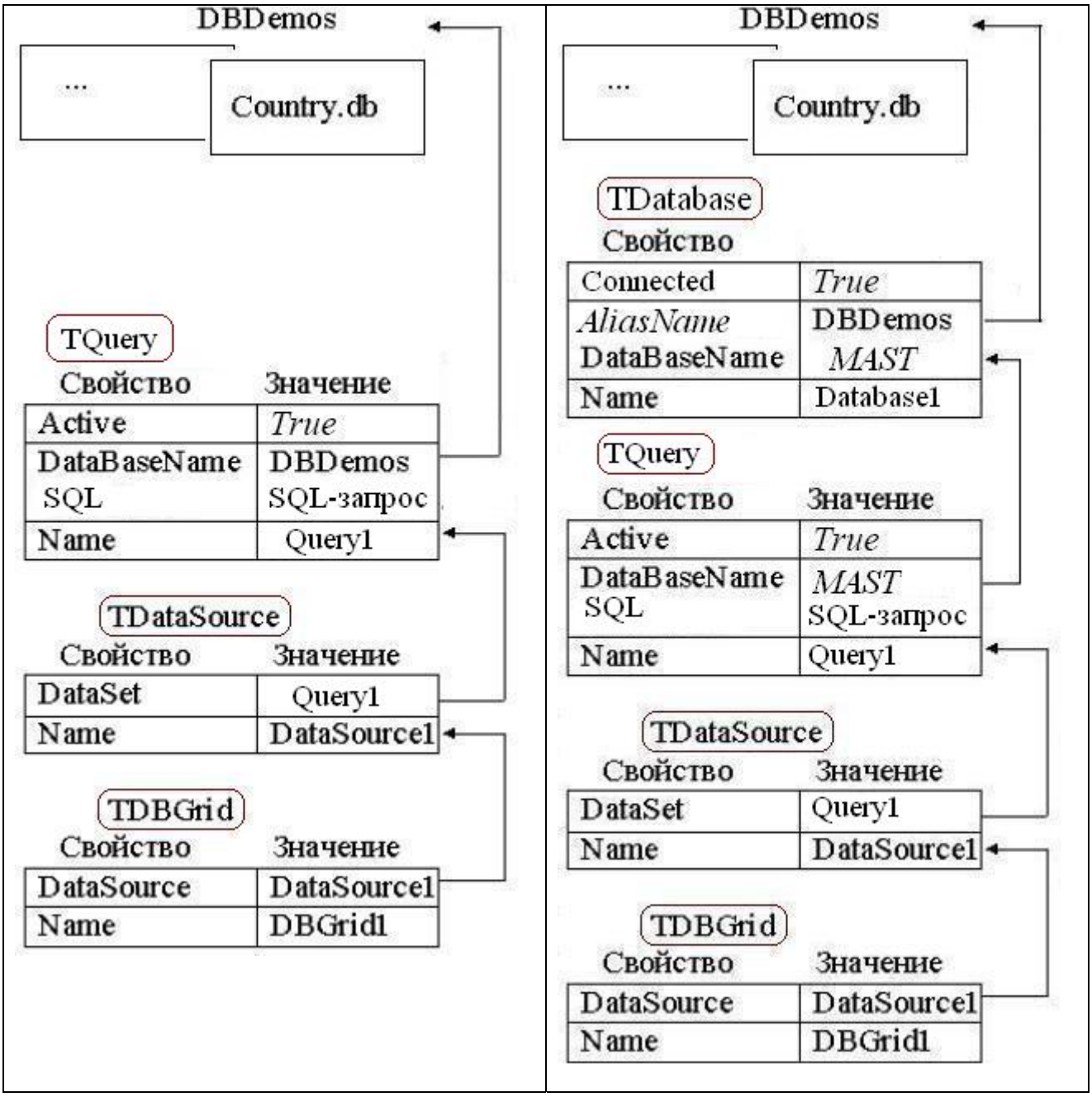
(Чтобы из формы **Form1** были видны компоненты модуля данных, надо с закладки «**Design**» формы **Form1** перейти на закладку «**Code**» и в секции «**interface**» в строку «**uses**» добавить имя, указанное после «**unit**» в коде модуля данных (Например, DataMod, если в модуле данных указано unit DataMod;.).

2. Нет отображения данных из таблицы на форме.

- Вы неверно связали компоненты меджу собой.
- Не установили свойство **Active** компонента **TQuery** в **true** (или не открыли запрос методом **Open** (ИмяЗапроса.Open**)**, например, на событии **OnActivate** формы).

3. В компонентах TBDEdit, TDBImage,…и других предназначенных для визуализации данных, находящихся в «Data Controls» нет отображения данных.

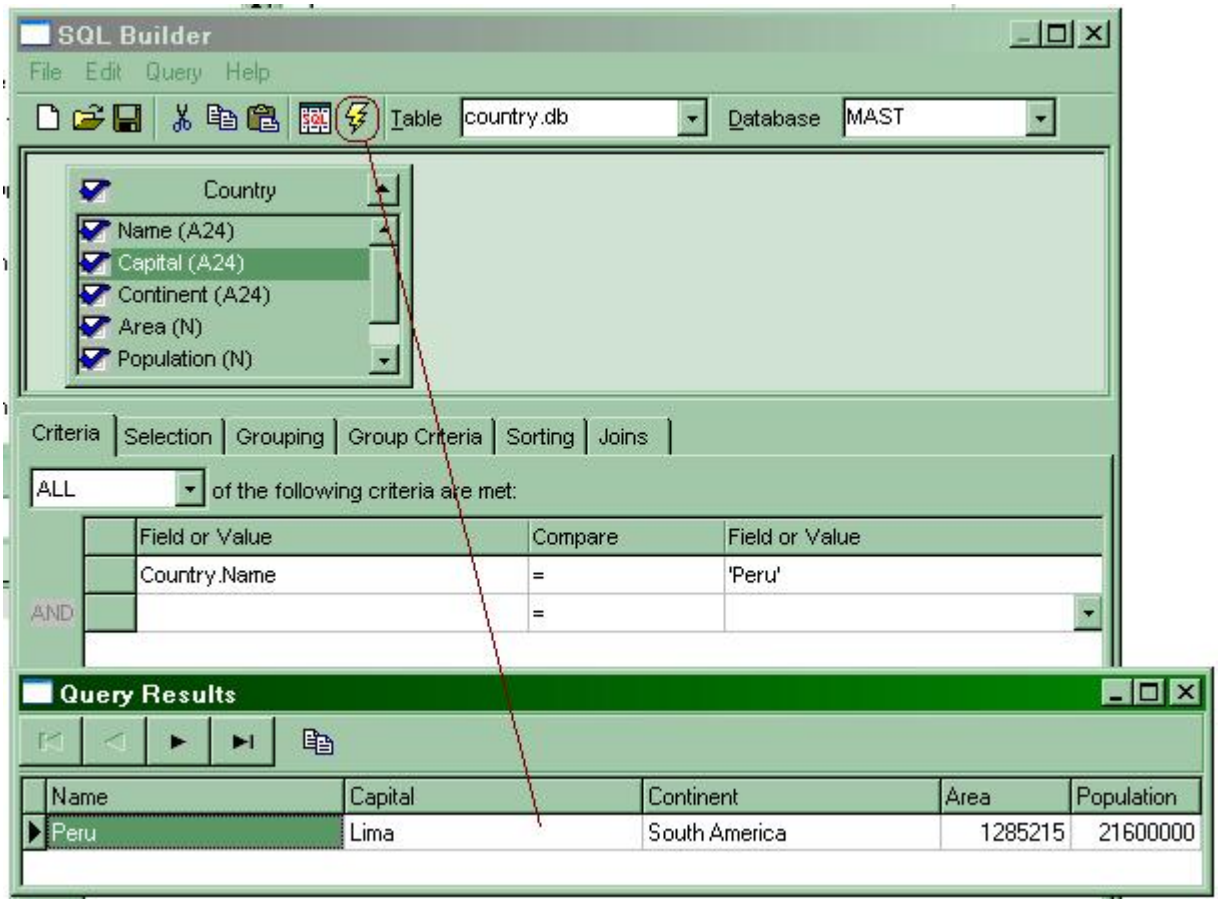
- Забыли связать эти компоненты с соответствующим полем из отображаемого запроса.



Изменяемые запросы. Параметрические запросы.

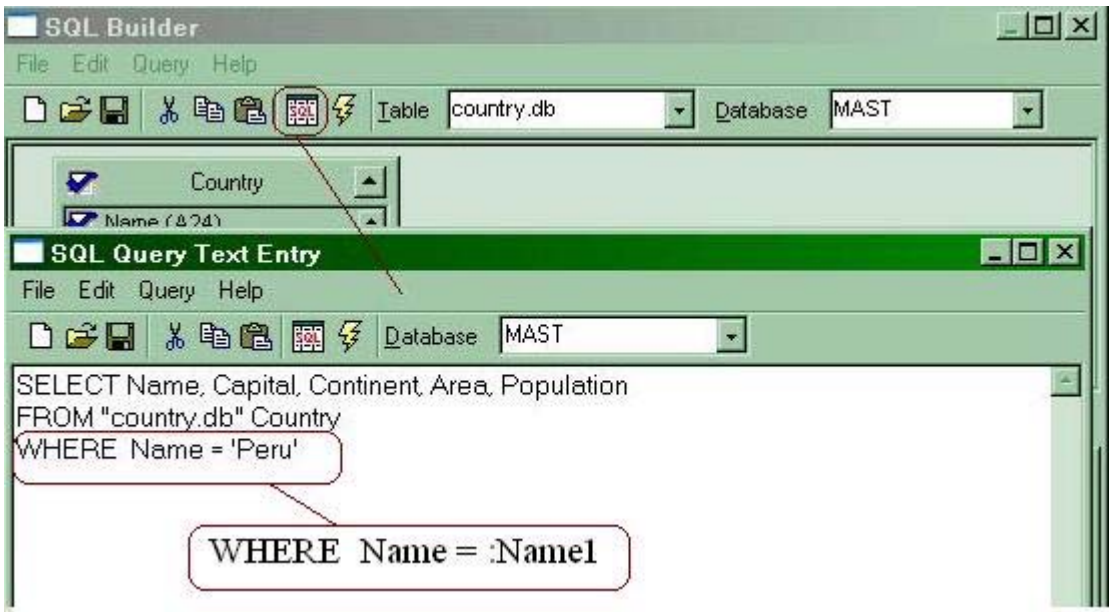
3. Создание параметрического запроса.

1. Создать форму «**Chld2**», в которой расположить компоненты **TQuery, TDataSource, TDBGrid, TDBNavigator.**
2. Выбрать компонент **TQuery** . Работать будем с таблицей **country.db**, имеющей Alias **DBDEMOS**. Выбрать все поля.
3. Если «**SQL Builder**» установлен, то перейти на закладку «**Criteria**» и установить для поля **Name** значение **Peru**. Выполнить запрос.



4. Переключиться в режим просмотра текста SQL запроса. Вместо строки ‘Peru’ написать **:Name1** (так мы зададим имя параметра, значение которого будем подставлять во время выполнения программы).

Для определения параметра в запросе используется двоеточие **(:)**, за которым следует имя параметра. Пример SQL-запроса с использованием динамического параметра: select * from EMPLOYEE where DEPT_NO = **:Dept_no**

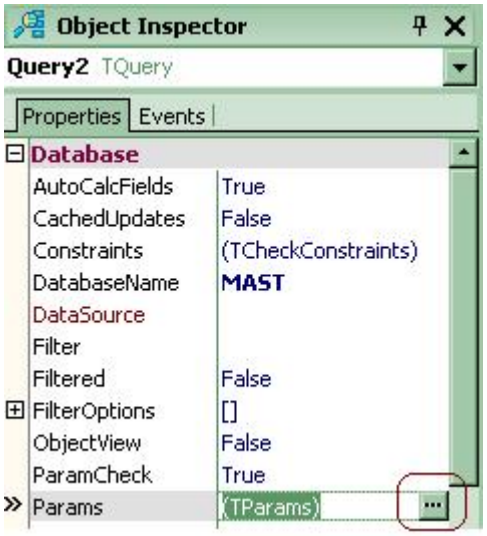


5. Закрыть «**SQL Builder**», сохранив изменения в запросе. Текст запроса запишется в свойство **SQL** компонента **TQuery.** Имя базы данных **(**Alias) попадет в свойство **DatabaseName**.

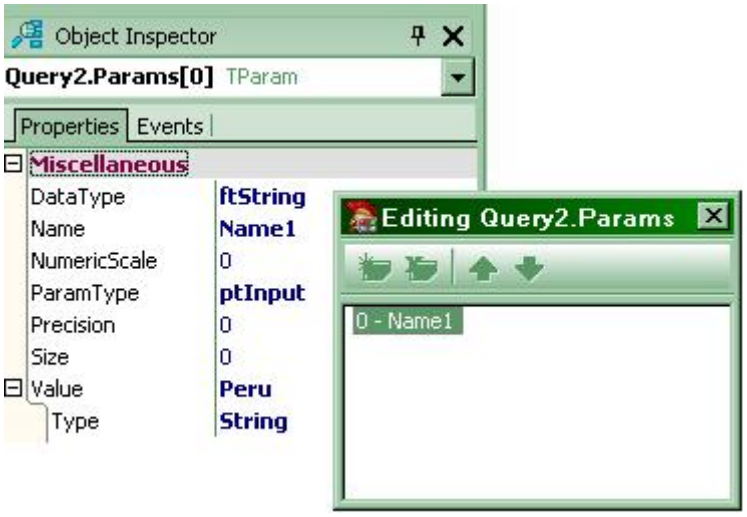
6. Проверить, что в свойство **SQL** компонента **TQuery** попал параметр.

7. В свойстве **Params** компонента **TQuery** задать (выбрать) типы параметров и начальные значения (например, Peru).

1.



2.



8. Связать между собой компоненты на форме «**Chld2**»
9. На форму «**Main**» добавить кнопку «**Параметрический запрос**». По этой кнопке вызвать форму «**Chld2**». Запустить программу.
10. На форму «**Chld2**» добавить поле для ввода страны и кнопку «**Показать**» для открытия запроса с новым значением параметра.
11. В обработчике события кнопки **OnClick** необходимо выполнить действия:
    - закрыть запрос (метод **Close** компонента **TQuery**),
    - задать значения параметров,
    - открыть запрос с новыми значениями параметров (метод **Open** компонента **TQuery**).

Обратиться к конкретному параметру можно:
1) указав индекс параметра в свойстве **Params** (Params[n])
Номер параметра можно посмотреть в окне редактора параметров

2) с помощью метода **ParamByName(Имя_параметра)** компонента **TQuery**
Имена можно посмотреть в окне редактора параметров

Для установки значения конкретного параметра использовать одно из свойств **AsXXXX** (**AsString**, **AsInteger**,…) объекта TParam или через свойство **Value**

```
ИмяQuery.Params[i].AsString:=…;
```
Или
```
ИмяQuery.ParamByName('имя_параметра').Value:=…;
```

Номера и имена параметров видны в окне редактора параметров.

Метод **Prepare** посылает запрос в BDE для проверки синтаксиса и оптимизации. Рекомендуется выполнять для динамических запросов.

```
Query1.close;     //Деактивируем запрос
if not Query1.Prepared
then
```

```
Query1.Prepare; //Убедимся что запрос подготовлен
//Берем значение, введенное пользователем и заменяем им параметр.
if edit1.text <> '' //Проверяем на предмет пустого ввода
then
        Query1.ParamByName('DEPT_NO').AsString := edit1.text
else
begin
        Query1.ParamByName('DEPT_NO').AsInteger := 0;
        edit1.text := '0';
end;
Query1.Open;    //Выполняем запрос и открываем набор данных
```

12. Запустить программу.

Лабораторная работа № 8

Работа с запросами. Изменяемые запросы. Формируемые запросы.

Свойство **SQL** компонента **TQuery** имеет тип **TStrings** и поэтому содержимое свойства **SQL** может формироваться программно методами **Add** (добавить элемент), **Delete** (удалить элемент), **Clear** (очистить список) и прочими.

 1. Продолжим работу с формой, которую создали при работе с параметрическими запросами. Программно будем формировать **SQL**-операторы, с помощью которых будем работать с данными, находящимися в таблицах. Строки сформированных **SQL**-операторов будем отображать в компоненте **TListBox.**

1. На форму «**Chld2**» добавить 5 компонентов **TEdit**, расположив их под каждой колонкой компонента **TDBGrid**.
2. На форму «**Chld2**» добавить компонент **TListBox**, в котором будем будем отображать формированные строки **SQL**- операторов.
3. На форму «**Chld2**» добавить компонент **TQuery** через который будем работать (добавлять записи) с таблицей country.db.
4. На форму «**Chld2**» добавить кнопку «**Добавить**», по которой будем запускать программно сформированный запрос.
5. В обработчике события **OnClick** кнопки «**Добавить**», сформировать строку **SQL**-запроса. По этому запросу будем добавлять записи в таблицу. Значения для полей брать из компонентов **TEdit.**

Формат **SQL**-операторов для изменения данных
[...] означает необязательные параметры
{вариант1| вариант2 |…} означает один из вариантов

Добавление записей
INSERT INTO <*объект*> [**(***столбец1* [,*столбец2* ...])]
{VALUES (<*значение1*> [, < *значение2*> ...]) | <*оператор* SELECT>};

Оператор SELECT возвращающий ноль или более строк, где число столбцов в каждой строке такое же, как число элементов, которые должны быть вставлены.

Следующая инструкция добавляет строку в таблицу, присваивает значения двум столбцам:

INSERT INTO EMPLOYEE (EMP_NO, PROJ_ID) VALUES (52, "DGPII");

Если бы таблица EMPLOYEE состояла их двух столбцов, то можно написать

INSERT INTO EMPLOYEE VALUES (52, "DGPII");

Следующая инструкция определяет значения, чтобы вставить в таблицу, используя инструкцию SELECT:

INSERT INTO PROJECTS
SELECT * FROM NEW_PROJECTS WHERE NEW_PROJECTS.START_DATE > "6-JUN-1994";

Изменение записей
UPDATE <*объект*>
SET *столбец1* = < *значение1*> [,*столбец2* = < *значение2*> ...]
[WHERE <*условие поиска*>;

Следующая инструкция изменяет столбцы для всех строк таблицы:

UPDATE CITIES
SET POPULATION = POPULATION * 1.03;

Следующая инструкция использует предложение WHERE, чтобы ограничить модификацию столбцов подмножеством строк:

UPDATE COUNTRY
SET CURRENCY = "USDollar"
WHERE COUNTRY = "USA";

Удаление записей
DELETE FROM <*объект*> [WHERE < *условие поиска* >];

Следующая инструкция удаляет все строки из таблицы:

DELETE FROM EMPLOYEE_PROJECT;

Следующая инструкция удаляет строку для служащего #141:

DELETE FROM SALARY_HISTORY
WHERE EMP_NO = 141;
Обратите внимание: EMP_NO это PRIMARY KEY для таблицы EMPLOYEE и поэтому гарантирует уникальную идентификацию строки.

6. Сформированную строку **SQL**-запроса отобразить в компонент **TListBox.**
Свойство **Items (**тип **TStrings)** компонент **TListBox** отвечает за строку, которая появляется в **TListBox**
7. Запустить запрос на добавление записи в таблицу. Не забыть отобразить возможные ошибки при работе с базой данных (тип ошибки **EDBEngineError**) и прочие ошибки.

В случае использование INSERT, UPDATE , DELETE набор данных не возвращается. Компонент **TQuery** с одним из перечисленных **SQL**-операторов следует открывать, выполняя метод **ExecSQL:**

ИмяЗапроса. ExecSQL;

8. Поле **Name** в таблице **country.db** ключевое (т.е. его значения должны быть уникальны). Перед добавлением записи в таблицу проверять значение этого поля на уникальность (например, создать запрос, в который добавить значение поля **Name** при условии, что это значение равно введенному значению соответствующего **TEdit**. Если результирующий набор данных не пуст, это означает, что добавляемая запись не уникальна).
9. На форму «**Chld2**» добавить кнопку «**Удалить**», по которой будем запускать программно сформированный запрос на удаление записи.
10. В обработчике события **OnClick** кнопки «**Удалить**», сформировать строку **SQL**-запроса. По этому запросу будем удалять запись, выбранную в данный момент в компоненте **TDBGrid** . Сформированную строку отображать в компоненте **TListBox** .
11. Подкорректировать процедуры, написанные ранее, чтобы ваше приложение работало красиво:
- чтобы при добавлении, удалении, обновлении записей в таблице происходило обновление компонента **TDBGrid;**
- добавить во все процедуры обработку ошибок

```
try
  запуск запроса
except
   on EDBEngineError do
                    обработка ошибки или сообщение об ошибке;
   on тип ошибки do
     ……
   else
        сообщение о прочей ошибке;
end;
```

12. На форму «**Chld2**» добавить кнопку «**Изменить**», по которой изменять запись, выбранную в данный момент компоненте **TDBGrid** . Сформированную строку отображать в компоненте **TListBox** . Изменяемые значения указывать в компонентах **TEdit** . Условия для обновления определять по ключевому слову таблицы **country.db**.

В запросах можно подсчитывать агрегированные значения данных (минимум, максимум, среднее, счетчик повторений).

Агрегированные значения вычисляются с помощью агрегированных функций **Avg** (возвращает арифметическое среднее значений поля), **Count** (возвращает количество записей), **Max** (возвращает максимальное значение поля), **Min** (возвращает минимальное значение поля), **Sum** (возвращает сумму всех значений).

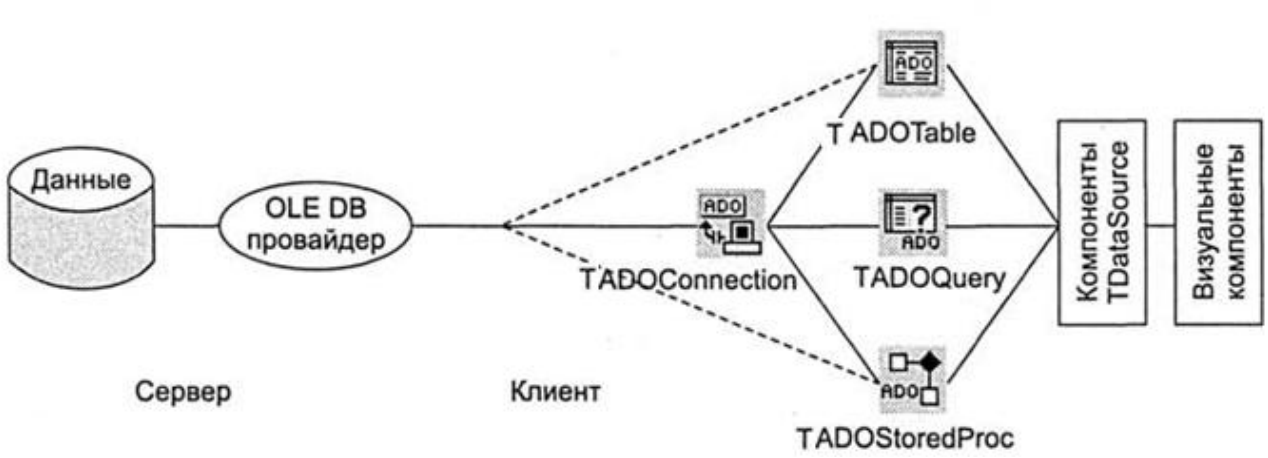
13. Подкорректировать процедуру добавления записей (кнопка «**Добавить**»), чтобы перед добавлением записей выдавалось сообщение о том, какая запись (по счету) будет добавлена в таблицу. После добавления записи в компоненте **TLabel** отобразить максимальное и минимальное значения из поля **Area**.

Лабораторная работа № 9

Использование технологии ADO

ADO (**ActiveX Data Objects** - объекты данных, построенные как объекты **ActiveX**) - это часть архитектуры универсального доступа к данным от **Microsoft**. Технология **ADO** базируется на возможностях **COM**, а именно интерфейсов **OLE DB**. **OLE DB** представляет собой интерфейс системного уровня, обеспечивающий доступ к различным источникам данных, изолируя приложение от вида источника. **ADO** представляет собой высокоуровневый программный интерфейс для доступа к **OLE DB**-интерфейсам. **ADO** содержит набор объектов, используемых для соединения с источником данных, для чтения, добавления, удаления и модификации данных.

При использовании технологии **ADO** приложение взаимодействует с любым источником данных (база данных, электронная таблица, файл) при помощи провайдера данных. Провайдер данных это некоторая надстройка над специальной технологией **OLE DB.** Провайдер «знает» о местоположении хранилища данных и его содержании, умеет обращаться к данным с запросами и интерпретировать возвращаемую служебную информацию и результаты запросов с целью их передачи приложению. Взаимодействие компонентов **ADO** и провайдера осуществляется на основе технологии **ActiveX,** при этом провайдер реализуется как **COM-сервер, а ADO** компоненты как **COM-клиенты**. Ниже представлена схема реализации технологии **ADO** в Delphi.



Базовые объекты **ADO** входят в комплект поставки Delphi. Это объекты: **Connection**, **Recordset**, **Command**, **Parameter**, **Field**, **Error** и **Property**.

Объект **Connection** используется для установления связи с источником данных. С его помощью производится настройка параметров соединения, этот объект обеспечивает механизм транзакций. На него может ссылаться произвольное количество объектов **Command** и **Recordset**. В этом случае **Connection** управляет транзакциями этих объектов. С объектом связан набор объктов **Error**, в котором фиксируются все ошибки, связанные с работой объкта **Connection**.

Объект **Recordset** представляет набор записей, полученных из источника данных. Этот объект может применяться для добавления, удаления, обновления и просмотра наборов записей. С объектом **Recordset** автоматически связывается набор объектов **Field**, в которых описываются все поля наборов данных. При создании объекта автоматически создается и связанный с ним курсор, обеспечивающий просмотр, редактирование и изменение записей.

Объект **Command** содержит команду, которая применяется к источнику данных. Команды могут представлять собой простые **SQL** - операторы или вызовы хранимых процедур. В последнем случае используется набор объектов **Parameter** объекта **Command** для указания информации о индивидуальных параметрах: объеме, типе данных, направлении

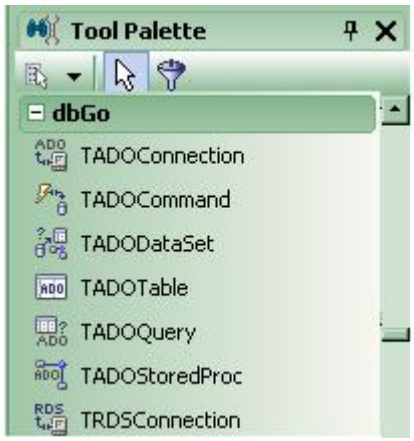
передачи и значении. При обнаружении ошибки с объектом связываестя коллекция объектов **Error**.

Объект **Parameter** определяет единственный параметр, который будет использоваться при выполнении метода **Execute** объекта **Command**, тип параметра, размер и способ применения (входной, выходной, входной и выходной или только для чтения). При необходимости с объектом **Command** можно связать коллекцию объектов **Parameter** для указания множества параметров.

Коллекция объектов **Error** хранит все ошибки, связанные с работой остальных объектов, и прежде всего объектов **Connection**, **Command** и **Recordset**.

Объект **Field** хранит всю необходимую информацию об одном поле набора данных. Поскольку набор данных обычно содержит несколько полей, с объектом **Recordset** связана коллекция объектов **Field**. С любым полем **Field** можно связать произвольную коллекцию объектов **Property**, определяющих индивидуальные характеристики поля.

Объект **Property** может быть связан с любым другим объектом **ADO**, кроме объектов **Connection** и **Error**. Он может хранить как статические, так и динамические свойства. Статических свойств у объекта всего четыре: **Name**, **Type**, **Value** и **Attributes**. Остальные свойства динамические и создаются в ходе выполнения программы.



Компоненты **ADO** в палитре компонентов Delphi (расположены на странице «**dbGo**» «**Tool Palette**» («**ADO**» для старых версий Delphi)) представляют собой надстройки над базовыми объектами.

Компоненты доступа к данным **ADO** могут использовать два варианта подключения к хранилищу данных.

В первом случае компоненты используют свойство **ConnectionString** для прямого обращения к хранилищу данных.

Каждый компонент, обращающийся к хранилищу данных **ADO** самостоятельно, задавая параметры соединения в свойстве **ConnectionString**, открывает собственное соединение. Чем больше приложение содержит компонентов **ADO**, тем больше соединений может быть открыто одновременно. Поэтому целесообразно реализовать механизм соединения **ADO** через компонент **TADOConnection**.

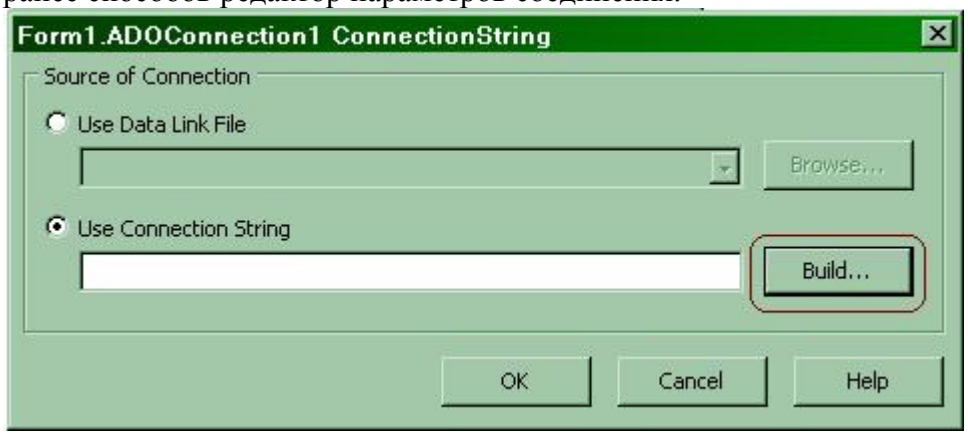
Свойство **ConnectionString** предназначено для хранения информации о соединении. В нем через точку с запятой перечисляются все необходимые параметры, например имя провайдера соединения или удаленного сервера. Отметим, что набор параметров строки подключения изменяется в зависимости от типа провайдера.

Во втором случае используется специальный компонент **TADOConnection**, который обеспечивает расширенное управление соединением и позволяет обращаться к данным нескольким компонентам одновременно.

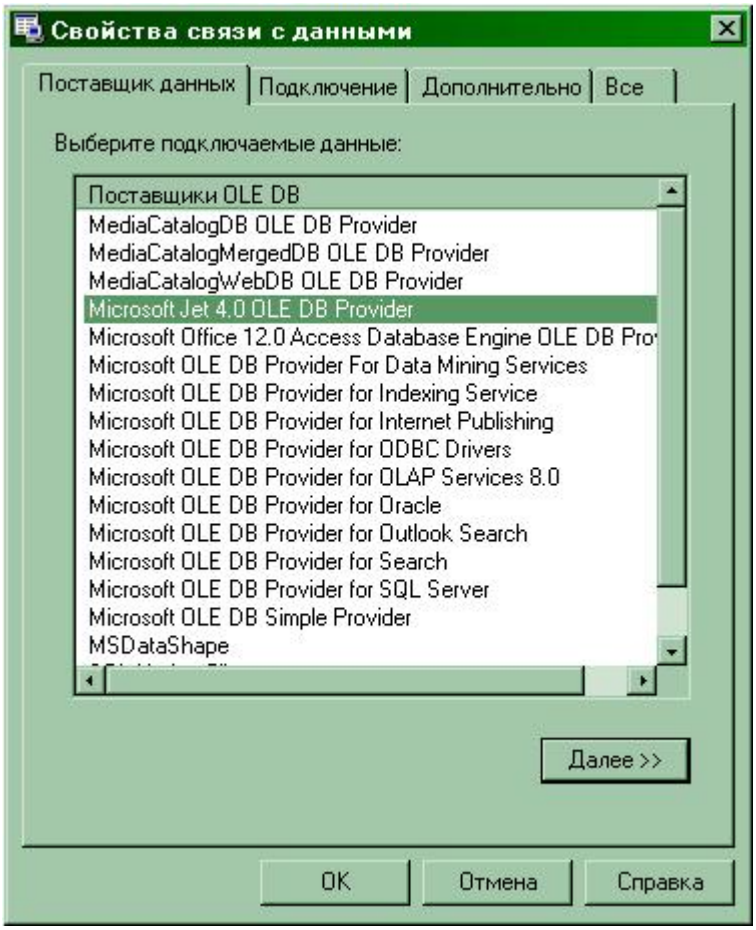
Набор параметров строки подключения может настраиваться как вручную, так и при помощи специального редактора параметров соединения. Этот редактор вызывается двойным щелчком на компоненте **TADOConnection**, перенесенным на форму, или при нажатии правой кнопкой мыши на компоненте **TADOConnection** и выбором из меню «**Edit ConnectionString**», или щелчком на кнопке с тремя точками свойства **ConnectionString** в инспекторе объектов (Object Inspector).

Компонент **TADOConnection** открывает соединение, также заданное свойством **ConnectionString** и предоставляет разработчику дополнительные средства управления соединением. Компоненты, работающие с хранилищем данных **ADO** через данное соединение, подключаются к компоненту **TADOConnection** при помощи свойства **Connection**, которое имеет каждый компонент.

Рассмотрим создание строки подключение **ConnectionString** для базы данных формата **Access**. Будем использовать компонент **TADOConnection.** Вызвать любым из перечисленных ранее способов редактор параметров соединения.



В группе «**Sourse of Connection**» выбрать переключатель «**Use Connection String**» и нажать на кнопку «**Build…**».

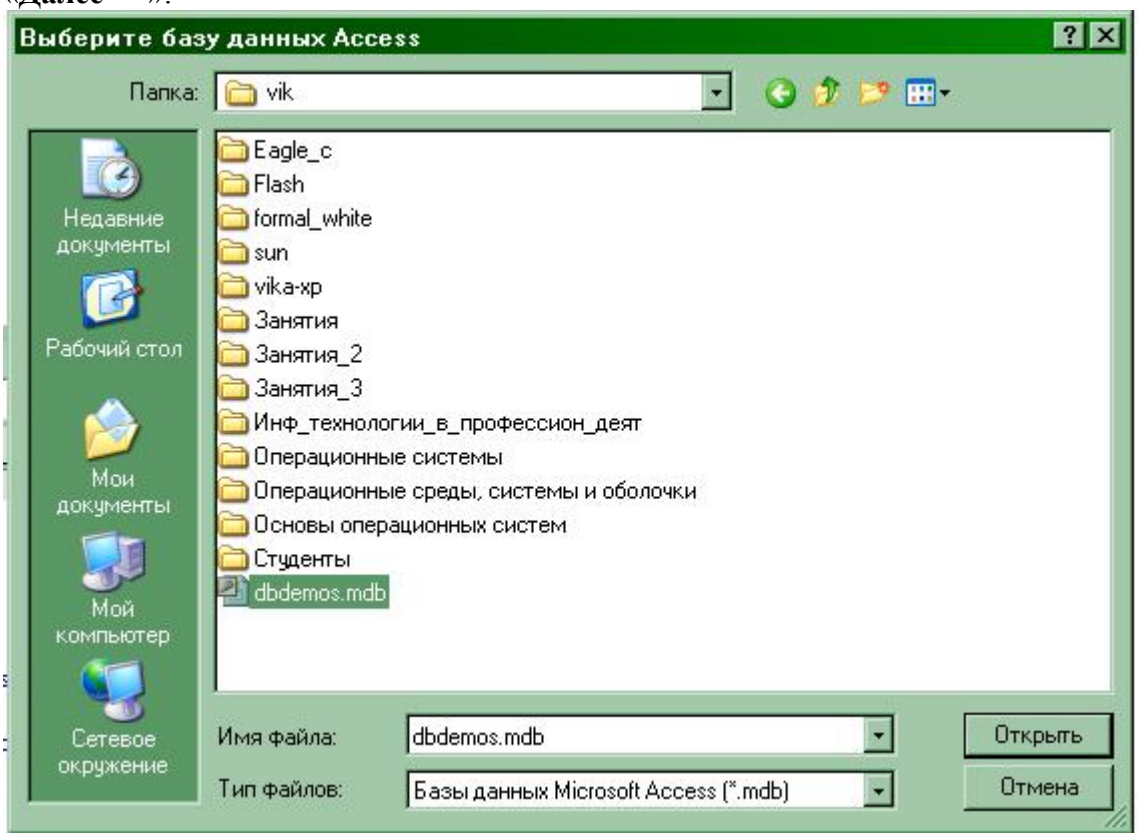


На экране появится диалоговое окно «**Свойства связи с данными**», в котором следует настроить параметры соединения вручную. Окно содержит четыре закладки, позволяющие этап за этапом задать все необходимые параметры.

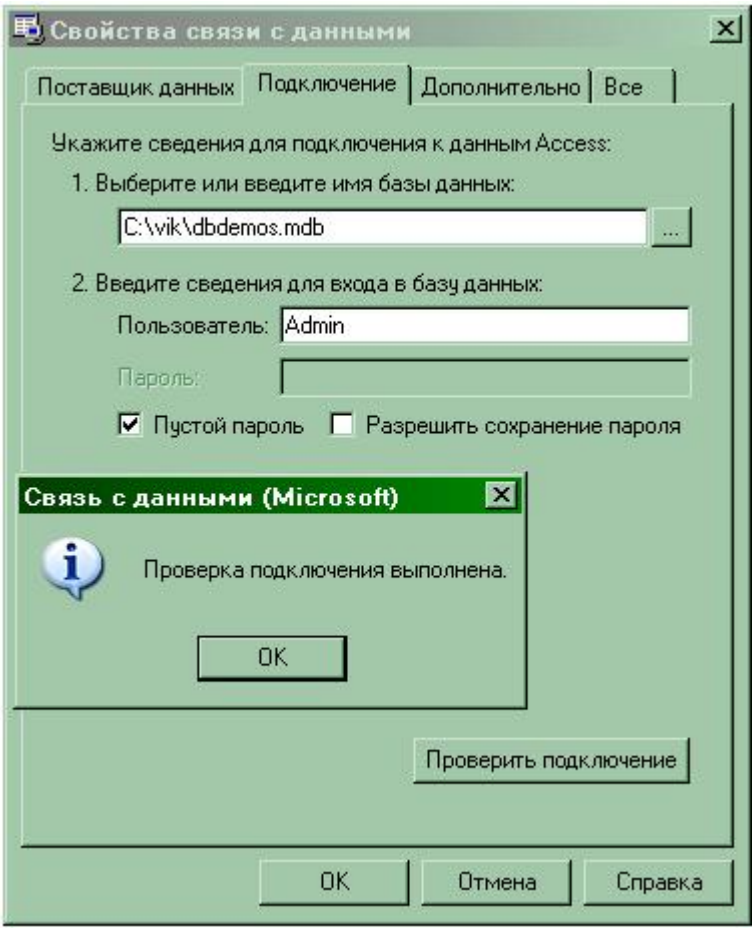
Первая страница (**Поставщик данных**) позволяет выбрать провайдера **OLE DB** для конкретного типа источника данных из числа провайдеров, установленных в системе. Здесь находятся провайдеры не только для серверов баз данных, но и для служб, установленных в операционной системе.

Состав элементов управления следующих страниц зависит от типа источника данных. Далее необходимо задать источник данных (имя сервера, базу данных, файл и т. д.), режим аутентификации пользователя, а также определить имя пользователя и пароль.

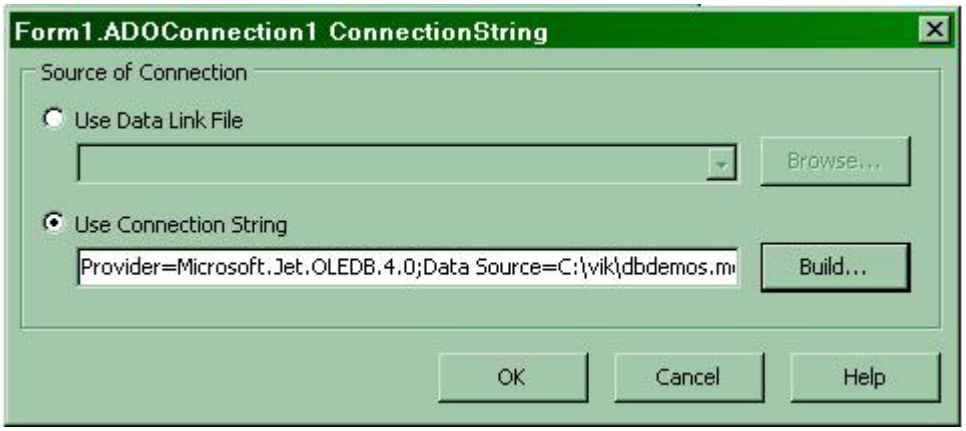
Выбираем поставщика данных **Microsoft.Jet 4.0 OLE DB Provider** и нажимаем на кнопку «**Далее >>**».



Откроется окно для выбора файла базы данных. Находим файл базы данных и нажимаем кнопку «**Открыть**».



Происходит переход на закладку «**Подключение**». Необходимо указать пользователя и пароль. По кнопке «**Проверить подключение**» можно проверить правильно ли Вы подключаетесь к источнику данных. После завершения всех операций по формированию строки подключения и нажатия кнопки «**OK**», сформированная строка появится в свойстве **ConnectionString** компонента **TADOConnection**.



В результате проделанных действий будет сформирован следующая строка **ConnectionString:**
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\vik\dbdemos.mdb;Persist Security Info=False

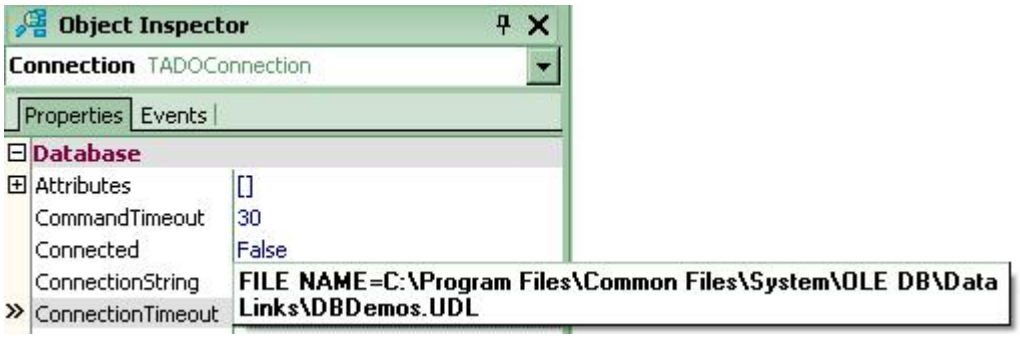
Если использовать строку **ConnectionString** путь к базе будет жестко закодирован внутри исполняемого файла приложения. В результате возможности приложения будут существенно ограничены. Чтобы решить проблему, в **ADO** используются так называемые файлы связи с данными (**Data Link Files**). Файл связи с данными — это строка подключения, оформленная в виде INIфайла. Например, в рамках Delphi устанавливается файл **dbdemos.udl**, в котором содержится следующий текст:

[oledb]
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb

Файл связи с данными может обладать любым расширением, однако рекомендуется использовать расширение **.UDL**. Вы можете создать такой файл при помощи любого текстового редактора. Если в редакторе свойства **ConnectionString** выбрать переключатель «**Use Data Link File**» (использовать файл связи с данными), то в этом свойстве будет автоматически размещена строка 'FILE NAME =', за которой будет указано имя файла связи с данными.

Файлы связи с данными можно разместить в любом месте диска, однако **ADO** использует для хранения таких файлов некоторый стандартный каталог:

C:\Program Files\Common Files\System\OLE DB\Data Links



Использование компонентов dbGo

Далее представлены компоненты со страницы **dbGo**.

Компонент dbGo	**Описание**	**Эквивалент из комплекта BDE**
TADOConnection	Подключение к базе данных	**TDataBase**
TADOCommand	Исполняет команду SQL	Нет эквивалента
TADODataSet	Многоцелевой наследник TDataSet	Нет эквивалента
TADOTable	Инкапсулирует таблицу	**TTable**
TADOQuery	Инкапсулирует SQL SELECT	**TQuery**
TADOStoredProc	Инкапсулирует сохраненную процедуру (stored procedure)	**TStoredProc**
TRDSConnection	Подключение Remote Data Services	Нет эквивалента

Четыре компонента наборов данных (**TADODataSet**, **TADOTable**, **TADOQuery** и **TADOStoredProc**) фактически полностью реализованы общим для них базовым классом **TCustomADODataSet**. Этот компонент несет ответственность за выполнение большинства функций, присущих набору данных. Компоненты **TADOTable**, **TADOQuery** и **TADOStoredProc** предназначены для упрощения адаптации кода, ориентированного на BDE. В качестве основного компонента при разработке новых программ следует считать компонент **TADODataSe**t.

Компонент TADOConnection

Применение компонента **TADOConnection** позволяет:

- всем компонентам доступа к данным **ADO** обращаться к хранилищу данных через одно соединение;
- напрямую задать объект провайдера соединения;
- строку подключения хранить в одном месте;
- выполнять транзакции;

- выполнять команды **ADO**;
- управлять соединением при помощи методов-обработчиков событий.

Компонент **TADOConnection** выполняет роль концентратора соединения с хранилищем данных. Часть свойств этого объекта доступеа через инспектор объектов. Для установления связи нужно с помощью свойства **ConnectionString** сформировать строку соединения с источником данных и затем установить значение **True** в свойство **Connected** или вызвать метод **Open**. Для разрыва связи используется метод **Close** компонента или в его свойство **Connected** устанавливается значение **False**. До и после открытия и закрытия соединения разработчик может использовать соответствующие стандартные методы-обработчики событий: **BeforeConnec**t, BeforeDisconnect, **AfterConnect**, **AfterDisconnec**. Кроме этого, компонент **TADOConnection** имеет дополнительные методы-обработчики.

В свойствах **CommandCount** и **DataSetCount** содержится количество соответствующих объектов, которые обслуживаются данным компонентом **TADOConnection**. В сочетании со свойствами **Commands** и **DataSets** можно получить доступ к любому интересующему объекту.

Чтобы открыть соединение с базой данных всех объектов, обслуживаемых компонентом ADOConnection1, нужно выполнить:

```
var i : Integer;
begin
for i:=0 to (ADOConnection1.DataSetCount-1) do
   ADOConnection1.DataSets[i].Open;
end;
```

Используя методы **GetProcedureNames** и **GetTableNames** можно получить список всех хранимых процедур и таблиц.

```
ADOConnection1.Open;
ADOConnection1.GetTableNames(ListBox1.Items);
```

В результате имена всех таблиц базы данных поместятся в список ListBox1.

Важной особенностью компонента **TADOConnection** является возможность управления с его помощью транзакциями. Для этого в состав компонента добавлены соответствующие методы и события. С помощью метода **BeginTrans** стартует новая транзакция, методы **CommitTrans** и **RollbackTrans** подтверждают или откатывают транзакцию. Разрешается произвольная глубина вложенности транзакций, то есть после старта одной транзакции может немедленно стартовать следующая и т. д. Транзакция, стартующая с помощью компонента **TADOConnection,** разделяется всеми другими связанными с ним компонентами. С помощью свойства **InTransaction** программа может определить, завершилась ли ранее начатая транзакция.

Компонент **TADOConnection** может выполнять команды **ADO** без помощи других компонентов. Для этого используется метод **Execute**:

```
procedure Execute(const CommandText: WideString; var RecordsAffected:
Integer; ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]);
```

или

```
function Execute(const CommandText: WideString; const CommandType:
TCommandType = cmdText; ExecuteOptions: TExecuteOptions = []): RecordSet;
```

Выполнение команды осуществляется процедурой **Execute**, если команда не возвращает набор записей, или одноименной функцией **Execute** если команда возвращает набор записей.

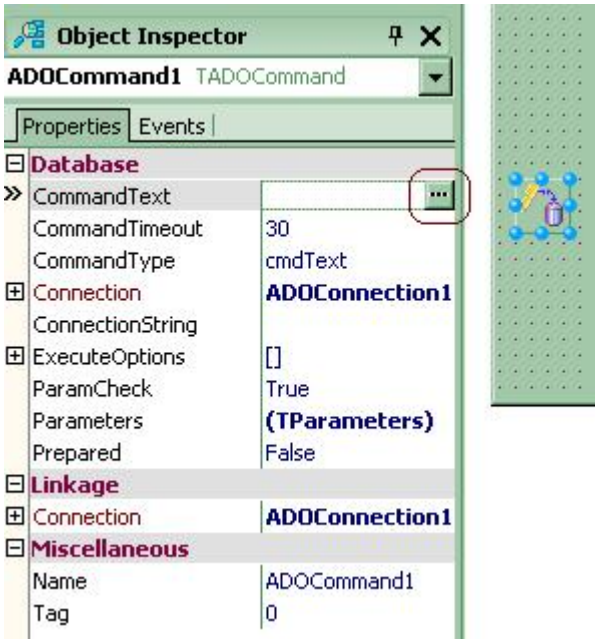
Перечислим некоторые свойства компонента **TADOConnection**.

Свойство	Описание
CommandTimeout	Указывает количество времени в секундах, после которого команда будет отменена, по умолчанию 30 с. Значение 0 приводит к неограниченному ожиданию.
Connected	Указывает активность соединения: True означает соединение открыто, False - закрыто.
ConnectionString	Используется для указания подробной информации о соединениях с источником данных.
ConnectionTimeout	Указывает количество времени в секундах, отведенное на установку соединения, по умолчанию 15 с. Значение 0 приводит к неограниченному ожиданию.
ConnectOptions	Устанавливает тип соединения: **coConnectunspecified** - синхронное соединение всегда ожидает результат последнего запроса; **coAsyncConnect** - асинхронное соединение может выполнять новые запросы, не дожидаясь ответа от предыдущих запросов.
CursorLocation	Указывает местоположение механизма управления курсором. **clUseClient** - клиентский курсор. Позволяет выполнять любые операции с данными, в том числе не поддерживаемые сервером; **clUseServer** - серверный курсор. Реализует только возможности сервера, но обеспечивает быструю обработку больших массивов данных.
DefaultDatabase	Служит для указания базы данных, используемой по умолчанию.
KeepConnection	Определяет реакцию компонента на неиспользуемое соединение. Если через соединение не подключен ни один активный компонент, свойство **KeepConnection** в значении **True** сохраняет соединение открытым. Иначе, после закрытия последнего связанного компонента **ADO**, соединение закрывается.
LoginPromt	Используется для указания того, нужно ли выводить диалоговое окно, предназначенное для ввода идентификатора пользователя и пароля при соединении с базой данных.

Компонент TADOCommand

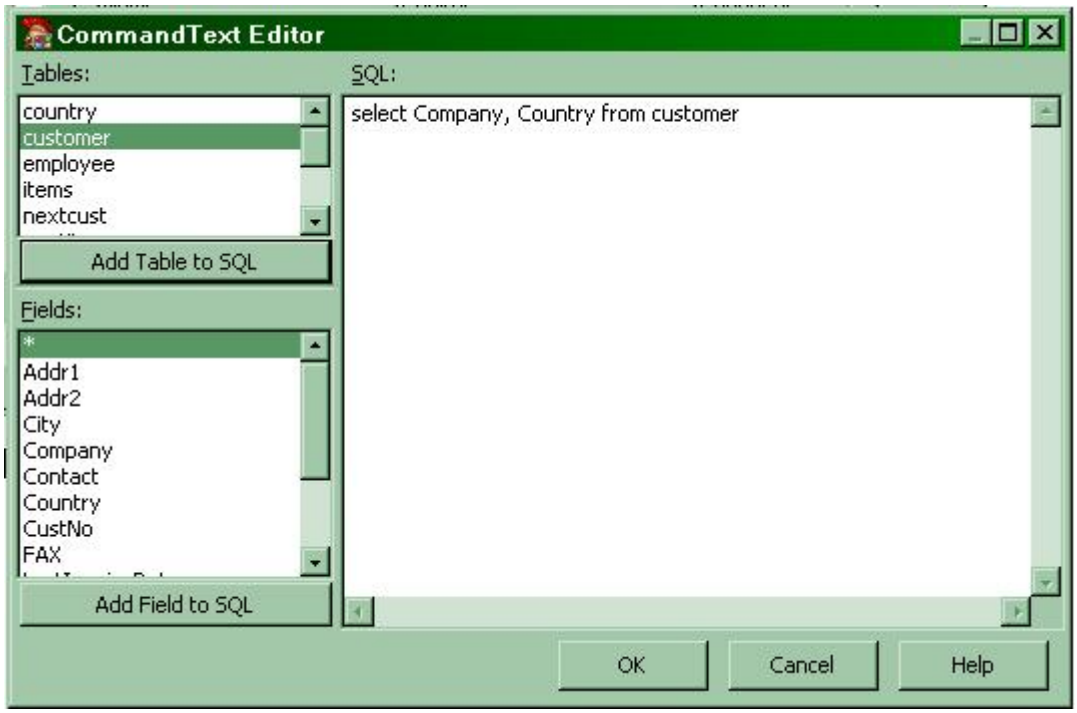
Компонент **TADOCommand** предназначен в основном для выполнение команд, не возвращающих результаты, таких как **SQL**-операторы языка определения данных **DDL** (**Data**

Definition Language). К предложениям **DDL** относятся практически все запросы, которые не начинаются зарезервированным словом **Select**.



В свойстве **CommandText** хранится текст исполняемой команды. За один раз компонент **TADOCommand** способен исполнить только одну команду.

Используя специализированный текстовый редактор компонента **TADOCommand** можно сформировать команду. Этот редактор «**CommandText Editor**» вызывается щелчком мыши на кнопке с многоточием в строке свойства **CommandText** инспектора объектов.



Компонент **TADOCommand** может возвращать результаты. Для этого в него включены три реализации метода **Execute**, две из которых как раз и предназначены для создания набора записей. Использование возвращаемого набора данных возможно с помощью компонента-посредника **TADODataSet**:

ADODataSet1.RecordSet:=ADOCommand1.Execute;

Множество **ExecuteOptions** при этом должно содержать значение **eoExecuteNoRecords**.

Перечислим некоторые свойства компонента **TADOCommand** :

Свойство	**Описание**
CommandText	Указывает, какая команда должна выполняться с помощью метода **Execute**. Это может быть строка, содержащая **SQL**-оператор, имя

	таблицы или хранимой процедуры. Для обеспечения более высокой производительности необходимо указывать тип команды в свойстве **CommandType**.
CommandTimeout	Указывает количество времени в секундах, после которого команда будет отменена, по умолчанию 30 с.
CommandType	Определяет тип команды, заданной в свойстве **CommandText**. Возможные значения: **mdUnknown** - тип заданной команды неизвестен; **cmdText** - текстовое представление команды или хранимой процедуры; **cmdTable** - в свойстве **CommandText** указано имя таблицы, образующей обрабатываемый набор; **cmdStoredProc** - в свойстве **CommandText** указано имя хранимой процедуры, создающей набор данных; **cmdFile** - в свойстве **CommandText** указано имя файла с сохраненным набором записей; **cmdTableDirect** - в свойстве **CommandText** указано имя таблицы.
Connection	Используется для указания компоненты **TADOConnection**, предназначенной для соединения с базой данных.
ConnectionString	Используется для указания подробной информации о соединениях с источником данных. Применяется вместо использования компонента **TADOConnection**.
ExecuteOptions	Используется для установки режимов, влияющих на выполнение команды, указанной в свойстве **CommandText**. Возможные значения: **eoAsyncExecute** - асинхронное выполнение команды; **eoAsyncFetch** - асинхронное выполнение команды на обновление набора данных; **eoAsyncFetch-NonBlocking** - асинхронное выполнение команды на обновление набора данных без установки блокировки; **eoExecuteNoRecords** - команда или хранимая процедура ничего не возвращает; если получены какие-то строки, то они игнорируются;
Parameters	Используется для установки значений параметров, необходимых для выполнения **SQL**-запроса или же хранимой процедуры.

Пример использования компонента **TADOCommand:**

```
// DataModule1 (TDataModule)
// ADOCommand1 (TADOCommand)
// LinkTable, MainTable (TADOTable)
DataModule1.ADOCommand1.CommandText := 'delete from ' +
DataModule1.LinkTable.TableName + ' where '"'+ DataModule1.MainTable.Fields[0].AsString +
'"'=' + DataModule1.LinkTable.TableName+'.'+ DataModule1.MainTable.Fields[0].FieldName;
try
DataModule1. ADOCommand1.Execute;
except
//Произошла ошибка
MessageDlg('Удаление невозможно, поскольку …'+ chr(13)+'
вызывает нарушение …', mtError, [mbOK], 0);
Abort;
end;
```

Компоненты: **TADODataSet** (универсальный набор данных), **TADOTable** (таблица БД), **TADOQuery** (запрос **SQL), TADOStoredProc** (хранимая процедура) инкапсулируют набор данных и адаптированны для работы с хранилищем данных **ADO**.

Для компонентов, инкапсулирующих набор данных, их общим предком является класс **TDataSet**, предоставляющий базовые функции управления набором данных. Компоненты **ADO** обладают обычным набором свойств и методов, а необходимый для доступа к данным через **ADO** механизм наследуют от своего общего предка - класса **TCustomADODataSet**. Кроме этого, класс **TCustomADODataSet** содержит ряд общих для всех потомков свойств и методов.

Методы **SaveToFile** и **LoadFromFile** используются в качестве одного из возможных механизмов обмена данными между разными компьютерами, а также для отложенной обработки данных. Перед вызовами методов **ADO**-набор должен быть закрыт. После успешного вызова **LoadFromFile** набор автоматически открывается в том состоянии, в котором он был сохранен методом **SaveToFile**.

Компонент TADODataSet

Компонент **TADODataSet** используется для выборки данных из одной или нескольких таблиц и доступа к ним посредством ADO. С помощью этого компонента можно получить все данные из таблицы, установить фильтры для того, чтобы выбрать ту информацию, которая отвечает некоторым условиям, выполнять SQL-запросы, запускать системные и определенные пользователем хранимые процедуры, а также сохранять наборы данных в файле и загружать их.

Компонент **TADODataSet** обеспечивает доступ к одной или нескольким таблицам базы данных с помощью запроса типа **Select**. Компонент рассчитан на возвращение набора данных, поэтому его нельзя использовать для выполнения команд, которые не возвращают результирующий набор данных. В компоненте есть свойство **CommandText**, однако в него можно поместить только оператор **Select**. Для выполнения DDL-предложений языка SQL можно использовать метод **Execute** компонента **TADOCommand** или метод **ExecSQL** компонента **TADOQuery**.

Приведем некоторые свойства компонента TADODataset:

Свойство	Описание
Active	Указывает, открыт ли набор данных. Значения свойства изменяют методы **Open** и **Close**.
CommandText	Указывает, какая команда должна выполняться с помощью метода **Execute**. Это может быть строка, содержащая SQL-оператор, имя таблицы или хранимой процедуры. Для обеспечения более высокой производительности необходимо указывать тип команды в свойстве **CommandType**.
CommandTimeout	Указывает количество времени в секундах, после которого команда будет отменена, умолчанию 30 с.
CommandType	Определяет тип команды, заданной в свойстве **CommandText**. Возможные значения: **mdUnknown** - тип заданной команды неизвестен; **cmdText** - текстовое представление команды или хранимой процедуры; **cmdTable** - в свойстве **CommandText** указано имя таблицы, образующей обрабатываемый набор; **cmdStoredProc** - в свойстве **CommandText** указано имя хранимой процедуры, создающей набор данных; **cmdFile** - в свойстве **CommandText** указано имя файла с сохраненным набором записей; **cmdTableDirect** - в свойстве **CommandText** указано имя таблицы.

Connection	Используется для указания компоненты **TADOConnection**, предназначенной для соединения с базой данных.
ConnectionString	Используется для указания подробной информации о соединениях с источником данных. Применяется вместо использования компонента **TADOConnection**.
CursorLocation	Указывает местоположение механизма управления курсором. **clUseClient** - клиентский курсор. Позволяет выполнять любые операции с данными, в том числе не поддерживаемые сервером; **clUseServer** - серверный курсор. Реализует только возможности сервера, но обеспечивает быструю обработку больших массивов данных.
ExecuteOptions	Используется для установки режимов, влияющих на выполнение команды, указанной в свойстве **CommandText**. Возможные значения: **eoAsyncExecute** - асинхронное выполнение команды; **eoAsyncFetch** - асинхронное выполнение команды на обновление набора данных; **eoAsyncFetch-NonBlocking** - асинхронное выполнение команды на обновление набора данных без установки блокировки; **eoExecuteNoRecords** - команда или хранимая процедура ничего не возвращает; если получены какие-то строки, то они игнорируются;
Filter	Используется для указания критерия отбора данных в набор.
Filtered	Используется для активации текущего фильтра.
Sort	Используется для указания порядка сортировки. Содержит список из пар из имен полей и направления сортировки (ASC, DESC).
Parameters	Используется для установки значений параметров, необходимых для выполнения SQL-запроса или же хранимой процедуры.
Prepared	Значение данного свойства нужно установить в **True**, перед вызовом команды, для указания того, что скомпилированная версия данной команды перед выполнением должна быть сохранена для ускорения последующих вызовов.

Компонент TADOTable

Компонент **TADOTable** используется в тех случаях, когда необходимо работать с одной таблицей в базе данных. Основные свойства и методы компонента **TADOTable** аналогичны свойствам компонента **TADODataSet**. Кроме того, этот компонент по своей функциональности аналогичен компоненту **TTable**.

Свойством **TableName** задает имя таблицы БД. Связь компонентов **TADOTable**, работающих с разными таблицами, одна из которых главная, а другая - вспомогательная, осуществляется с помощью свойств **MasterSource** и **MasterFields**. **MasterFields** свойство применяется для указания одного и более полей из главной таблицы, для реализации отношения строк «один ко многим». **MasterSource** (TDataSource) – содержит ссылку на главную таблицу при отношении типа «один ко многим». Если свойство **TableDirect** имеет значение **True**, осуществляется прямой доступ к таблице. В противном случае компонент генерирует соответствующий запрос. Свойство **ReadOnly** позволяет включить или отключить для таблицы режим «**только для чтения**».

Компонент TADOQuery

Компонент **TADOQuery** служит для выполнения **SQL**-запросов, позволяющих осуществить доступ к одной или нескольким таблицам в базе данных. Этот компонент по своей функциональности аналогичен компоненту **TQuery**. Компонента **TADOQuery**

обеспечивает те же функциональные возможности, что и компонент **TADODataSet**, свойство **CommandType** которого имеет значение **cmdText**.

Свойство **SQL** (**TStrings**) используется для указания **SQL**-запроса, который следует выполнить. В операторах могут использоваться параметры, идентификаторы которых помечаются знаком «**:**». Изменение параметров возможно только при закрытом запросе.

Обратиться к параметру можно следующим образом:

ADOQuery1.Parameters[0].Value:= ADOTable1.FieldByName('CustomerID').Value;)

или методом **ParamByName** в коллекции объектов **Parameters** по имени параметра

ADOQuery1.Parameters.ParamByName('IDCustomer').Value :=
ADOTable1.FieldByName('CustomerID').Value;

Изменение параметров возможно только при закрытом запросе.

Медод **ExecSQL** используется для выполнения **SQL**-запроса, хранящегося в свойстве **SQL**.

Компонент TADOStoredProc

Компонент **TADOStoredProc** предназначен для выполнения хранимых процедур базы данных. Компонент **TADOStoredProc** может также считаться частной версией более общего компонента **TADODataSet**, свойство **CommmandType** которого имеет значение **cmdStoredProc**. Компонент **TADOStoredProc** имеет свойство **ProcedureName** (**WideString**), служащее для указания хранимой процедуры для запуска. Параметры процедуры определяются свойством **Parameters** и задаются так же, как и параметры **TADOQuery**. Для выполнения хранимой процедуры используется метод **ExecProc**.

 1. Создать приложение, которое будет работать с таблицами базы данных **Борей.mdb (Nwind.mdb)** с использованием технологии **ADO.**

1.1. Создать собственный проект «**File**»->«**New**»->«**VCL Forms Application**» – «**Delphi for Win32**».

1.2. Создать модуль данных («**File**»->«**New**»->«**Delphi Projects**»-> «**Delphi Files**»->«**Data Module**»), в котором разместить невизуальные компоненты для работы с базой данных (Из палитры компонентов «**dbGo**» и «**Data Access**»).

1.3. В приложении задействовать компоненты **TADOConnection, TADOCommand, TADODataSet, TADOTable, TADOQuery.**

1.4. Для визуализации данных из базы данных использовать компоненты из палитры компонентов «**Data Controls**». Связать визуальные компоненты формы с невизуальными компонентами модуля данных.

1.5. Запустить созданное приложение на выполнение.

Возможные проблемы:

1. В форме не получается сослаться на невизуальные компоненты из модуля данных.

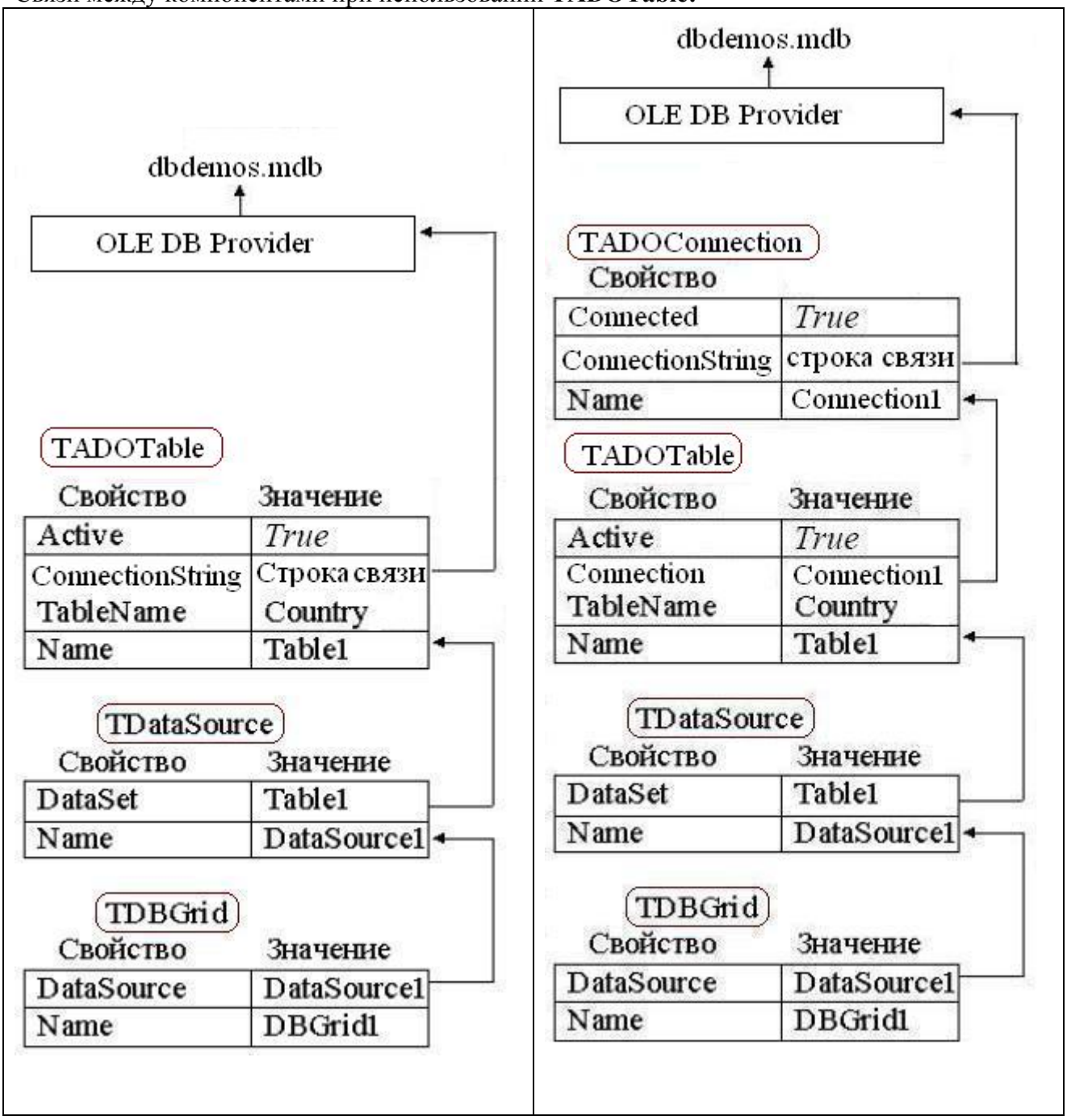
(Чтобы из формы **Form1** были видны компоненты модуля данных, надо с закладки «**Design**» формы **Form1** перейти на закладку «**Code**» и в секции «**interface**» в строку «**uses**» добавить

имя, указанное после «**unit**» в коде модуля данных (Например, DataMod, если в модуле данных указано unit DataMod;.).

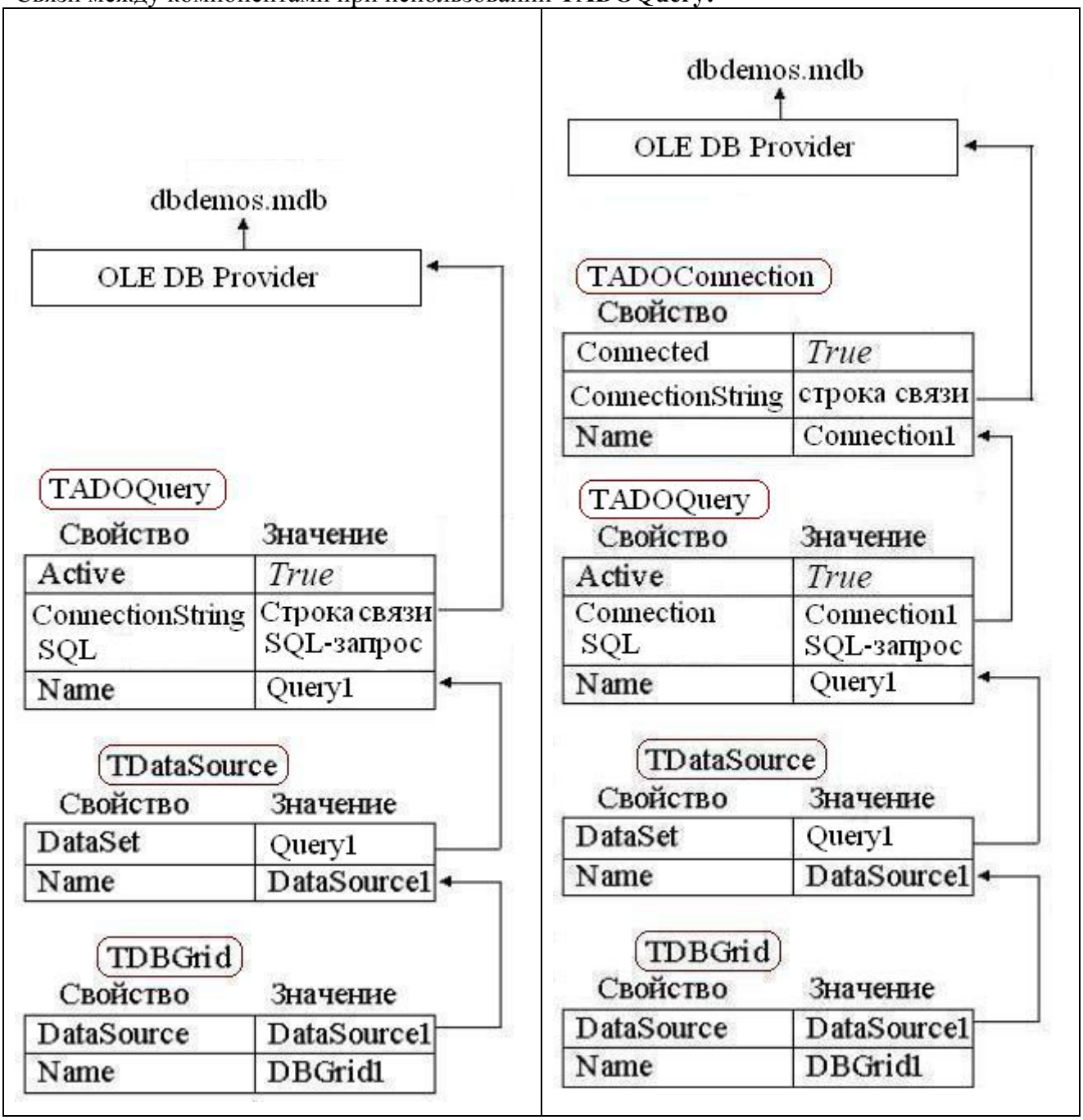
2. Нет отображения данных из таблицы на форме.

- Вы неверно связали компоненты между собой.
- Не установили свойство **Active** компонента **TADOTable** в **true** (или не открыли таблицу методом **Open**, например, на событии **OnActivate** формы).
- Не установили свойство **Active** компонента **TADOQuery** в **true** (или не открыли запрос методом **Open**, например, на событии **OnActivate** формы).

Связи между компонентами при использовании **TADOTable:**



Связи между компонентами при использовании **TADOQuery:**



3. В компонентах TBDEdit, TDBImage,…и других предназначенных для визуализации данных, находящихся в «Data Controls» нет отображения данных.

- Забыли связать эти компоненты с соответствующим полем из отображаемого набора данных.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Фаронов В.В. Программирование баз данных в Delphi 7. Учебный курс. – СПб.: Питер, 2006. – 464с.
2. Федоров А., Елманова Н. ADO в Delphi. СПб.: БХВ-Санкт-Петербург, 2002. 623 с.
3. Александровский А.Д. Delphi 5.0. Разработка корпоративных приложений. – М.: ДМК, 2000. – 512с.
4. Фаронов В. В. Delphi 2005. Разработка приложений для баз данных и Интернета. – СПб.: Питер, 2006. - 608 с.
5. Кэнту Марко Delphi 2005. Для профессионалов. – СПб.: Питер, 2006. – 912с.
6. Сорокин А. В. Разработка баз данных. – СПб.: Питер, 2005/ - 480
7. Архангельский А. Я. Программирование в Delphi для Windows. Версии 2006, 2007, Turbo Delphi. Бином-Пресс, 2007. - 1248 с

СОДЕРЖАНИЕ