Лабораторная работа 1

НАЧАЛО РАБОТЫ С БАЗАМИ ДАННЫХ

Цель работы: ознакомиться с работой СУБД **Microsoft Access** и основами разработки простых приложений БД в среде **Delphi**; разработать простейшее приложение для работы с базой данных; разработать приложение с использованием компонента **TADOQuery**; разработать приложение с установкой связи **Master–Detail** между наборами данных.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Microsoft Access позволяет управлять всеми сведениями из одного файла базы данных*. В этом файле используются следующие объекты:

– таблицы для сохранения данных;

– запросы для поиска и извлечения только требуемых данных;

– формы для просмотра, добавления и изменения данных в таблицах;

– отчеты для анализа и печати данных в определенном формате;

– страницы доступа к данным для просмотра, обновления и анализа данных из БД через Интернет или интрасеть.

При организации работы с базой данных посредством технологии **ADO** (ActiveX Data Objects), которая является частью технологии **MDAC** (Microsoft Data Access Components), применяются соединения с различными провайдерами **OLE DB**, входящими в состав **MDAC**. Среди них наиболее часто используются:

– **Microsoft Jet OLE DB**, обеспечивающий взаимодействие с базами данных **Microsoft Access** (97–2003) в формате **mdb**;

– **Microsoft OLE DB Provider for ODBC**, обеспечивающий взаимодействие с базами данных на основе использовании установленных в системе **ODBC** (Open Database Connectivity) драйверов;

– Microsoft **OLE DB Provider for SQL Server**, обеспечивающий взаимодействие с базами данных **Microsoft SQL Server** и **Microsoft Database Engine** (MSDE);

---

* Более подробно об этом см.: Бекаревич Ю., Пушкина Н. Самоучитель Access 2010. СПб. : БХВ-Петербург, 2011 ; Кошелев В. Е. Access 2007. Эффективное использование. М. : Бином-Пресс, 2007 ; Сурядный А. С. Microsoft Access 2010. Лучший самоучитель. СПб. : Астрель ВКТ, 2012.

– **Microsoft OLE DB Provider for Oracle**, обеспечиващий взаимодействие с базами данных **Oracle**.

Помимо перечисленных выше существует множество других провайдеров **OLE DB** для **MDAC**, которые предоставляет как корпорация **Microsoft**, так и независимые производители.

При установке **Microsoft Office 2007** или **2010** в операционной системе (ОС) инсталируется **Microsoft Office 12.0 Access Database Engine OLED DB Provider**, который позволяет работать с базами данных **Microsoft Access**, в том числе и с современным форматом **accdb**.

Для ознакомления с возможностями СУБД **Microsoft Access** и основами разработки клиентских приложений БД в среде **Delphi** разработаем приложение по учету поступающих на склад товаров. В ходе лабораторной работы будет поэтапно показано, как можно создать базу данных и реализовать простые возможности по управлению данными в среде разработки **RAD Studio**.

1.1. Постановка задачи

Создадим приложение, предназначенное для учета поступающих на склад товаров **GoodsInventory**. База данных состоит из четырех таблиц: справочников **Товары**, **Тип товаров**, **Поставщик** и операционной таблицы **Приход товаров** (табл. 1.1).

Таблица 1.1

Структура таблиц базы данных

Название поля	Тип поля	Размер (формат) поля	Ключ или индекс	Комментарий (описание)
*Товары (**GoodsCatalog**)*				
IDgc	Счетчик	Длинное целое	Ключ	Код товара
gcName	Текстовый	50	Индекс	Наименование товара
IDtg	Числовой	Длинное целое	Индекс	Код типа товара
gcMeasure	Текстовый	20	–	Единица измерения
gcCost	Денежный	Денежный	–	Цена за единицу измерения
*Тип товаров (**TypeGoods**)*				
IDtg	Счетчик	Длинное целое	Ключ	Код типа товара
tgName	Текстовый	50	–	Наименование типа товара

Окончание табл. 1.1

Название поля	Тип поля	Размер (формат) поля	Ключ или индекс	Комментарий (описание)
*Поставщики (**SuppliersCatalog**)*				
IDsc	Счетчик	Длинное целое	Ключ	Код поставщика
scName	Текстовый	40	Индекс	Название фирмы
scAddress	Текстовый	80	–	Адрес поставщика
scPhone	Текстовый	20	–	Телефон поставщика
scEmail	Текстовый	30	–	E-mail поставщика
*Приход товаров (**IncomingGoods**)*				
IDig	Счетчик	Длинное целое	Ключ	Номер прихода
igDate	Дата/время	Краткий формат даты	Индекс	Дата поставки
IDgc	Числовой	Длинное целое	Индекс	Код товара
igAmount	Числовой	Длинное целое	–	Количество товара
IDsc	Числовой	Длинное целое	–	Код поставщика

Для базы данных установим связи между этими таблицами (рис. 1.1).

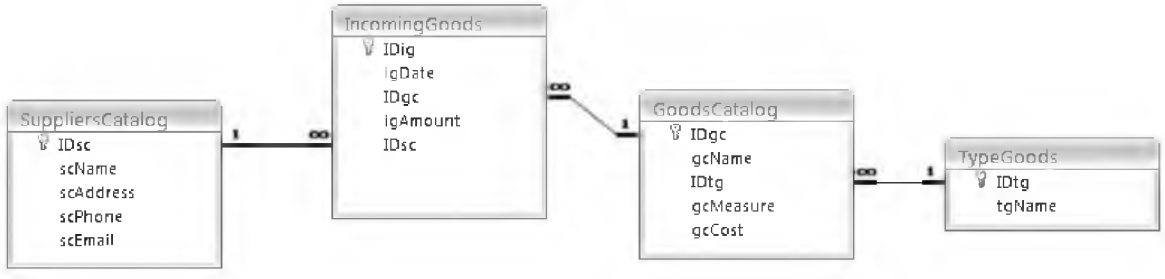


Рис. 1.1. Логическая схема базы данных

Чтобы решить поставленную задачу, необходимо реализовать в СУБД **Microsoft Access** базу данных. Перейдем к поэтапному рассмотрению процесса создания базы данных.

1.2. Создание базы данных в Microsoft Access

Система управления базами данных **Microsoft Access** предназначена для создания баз данных в среде операционной системы

Microsoft Windows, обеспечивая удобное и надежное управление данными, которые хранятся в таблицах.

Суть создания базы данных в **Microsoft Access** состоит в том, что сначала нужно разработать структуру базы данных, т. е. определить структуру таблиц и установить между ними связи, а затем заполнить эти таблицы.

Для создания новой пустой базы данных щелкнем на пиктограмме **Пустая база данных** в разделе **Новая пустая база данных**. Справа в окне приложения появится текстовое поле с именем файла **База данных1.accdb** и пиктограмма папки для сохранения файла в требуемый каталог. По умолчанию будут указаны имена файла, каталога и диска для сохранения файла, которые можно изменять. Окно задания файла новой базы данных представлено на рис. 1.2.

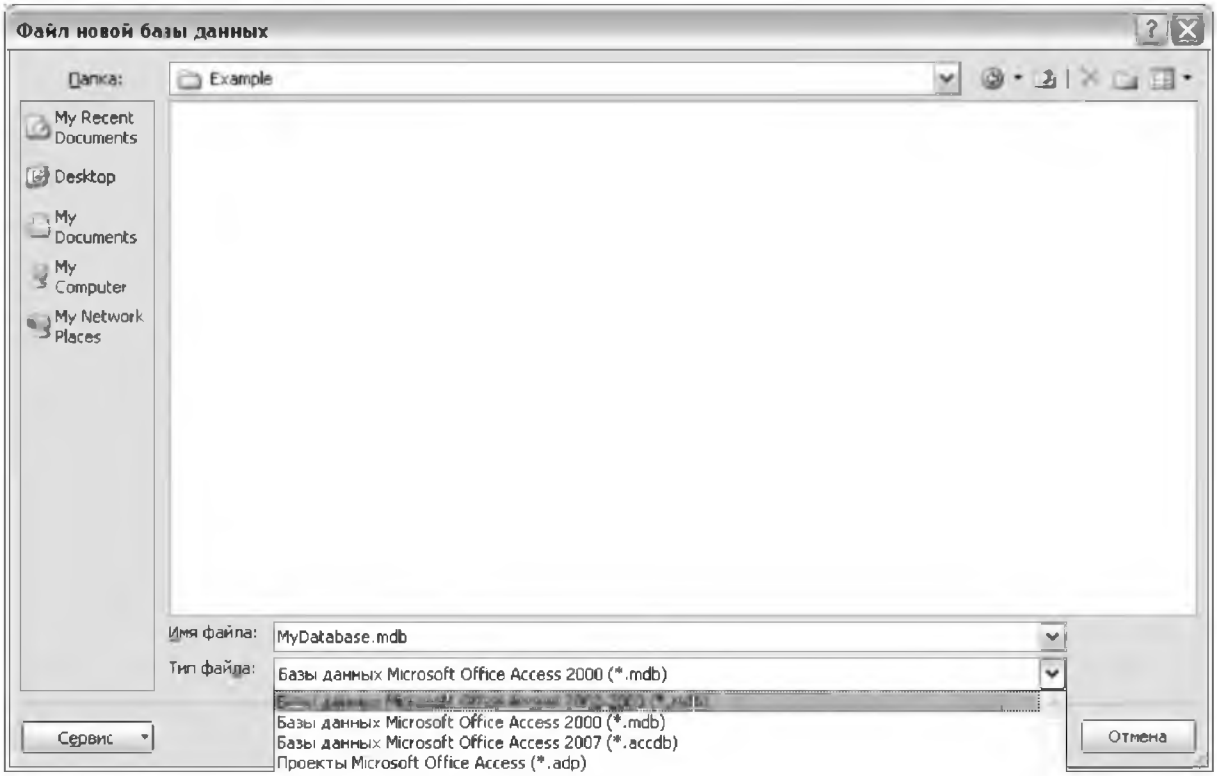


Рис. 1.2. Окно задания файла новой базы данных

Примечание. Начиная с СУБД **Access 2007** для новых баз данных по умолчанию используется формат файла **Access 2007** (**.accdb**), но файл новой базы данных при необходимости можно сохранить и в формате предыдущих версий **Access** (**.mdb**).

После выбора директории, диска, имени файла и формата БД щелкнем на кнопке **Создать**. Файл базы данных с выбранным именем

будет сохранен в указанной папке, после чего откроется окно **Microsoft Access** (рис. 1.3). В этом приложении отображается окно базы данных с назначенным именем и активной вкладкой **Режим таблицы**, а также новая пустая таблица с именем **Таблица 1**.

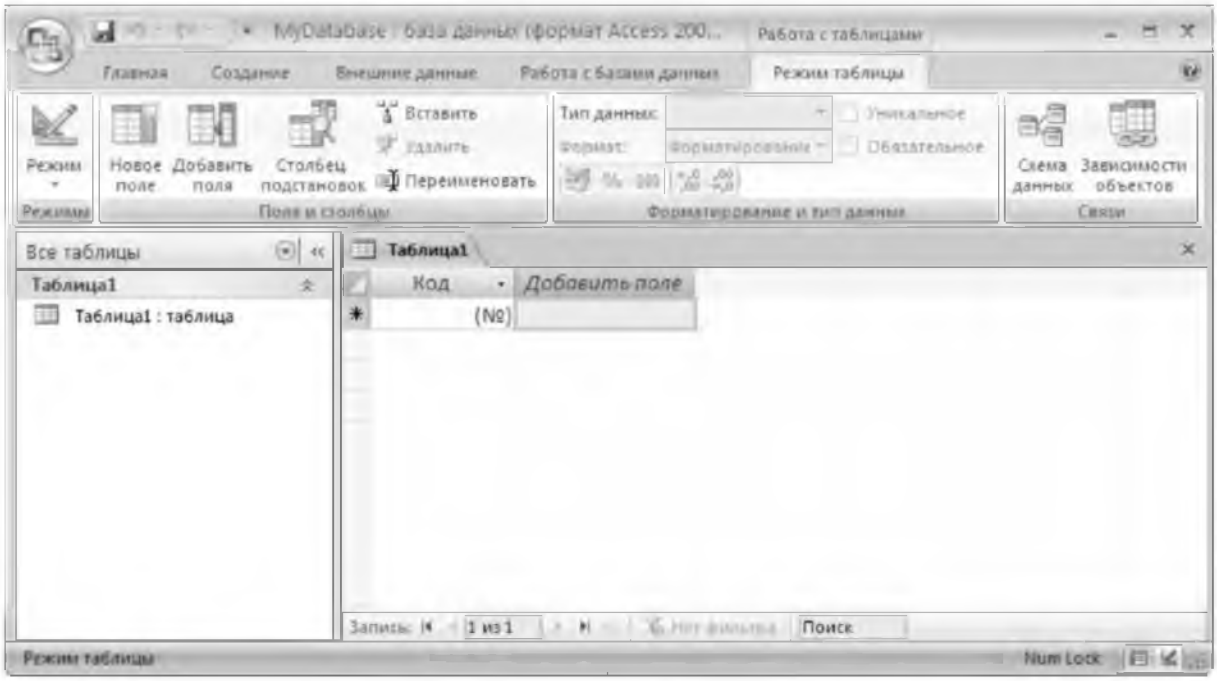


Рис. 1.3. Основное окно **Microsoft Access**
после создания новой базы данных

Сохраненную базу данных в формате файла **Access** можно сохранить и в других форматах, щелкнув на кнопке **Office** в окне БД и выбрав команду **Сохранить как**. В появившемся диалоговом окне нужно выбрать требуемый формат.

Рассмотрим окно приложения **Microsoft Access** (см. рис. 1.3) более подробно.

В верхней части окна расположены кнопка **Office**, панель быстрого доступа с пиктограммами (**Сохранить**, **Отменить**), строка заголовка и кнопки изменения размеров окна.

Ниже расположен элемент управления **Лента**, который состоит из вкладок **Главная**, **Создание**, **Внешние данные**, **Работа с базами данных** и других вкладок, которые появляются в зависимости от режима работы. В приложении **Access** используются контекстные инструменты, которые отображаются при работе с определенным объектом. Так, например, при работе с таблицей будут доступны контекстные инструменты для объекта **Таблица** под названием **Работа с таблицами**.

Под лентой слева расположена область переходов, а справа – окно редактирования, в котором отображается редактируемый объект. В области переходов находятся все объекты **Microsoft Access** (таблицы, формы, запросы, отчеты и т. д.). В списке **Все объекты Access** можно выбрать требуемый объект. При двойном щелчке на имени объекта в области переходов этот объект будет отображаться в окне редактирования.

Внизу окна расположена строка состояния и кнопки режимов просмотра.

После создания базы данных мы можем приступить к добавлению в нее таблиц и настройке связей между ними.

1.2.1. Создание таблиц

В **Microsoft Access** таблицу можно создать с помощью формирования новой базы данных, вставки таблицы в существующую базу данных, а также импорта или создания ссылки на таблицу из другого источника данных, например такого, как книга **Microsoft Office Excel**, документ **Microsoft Office Word**, текстовый файл или другая база данных. **MS Office Access 2007** поддерживает типы данных, которые представлены в табл. 1.2.

Таблица 1.2

Типы данных, используемые для полей в MS Office Access 2007

Тип данных	Объект хранения	Размер
Текстовый	Алфавитно-цифровые знаки	До 255 знаков
Поле **MEMO**	Алфавитно-цифровые знаки (более 255 знаков) или форматированный текст	До 1 Гбайта знаков или 2 Гбайт памяти (2 байта на знак), из которых в элементе управления можно отобразить 65 535 знаков
Числовой	Числовые значения (целые или дробные)	1, 2, 4 и 8 байт или 16 байт (если используется для кода репликации)
Дата/время	Даты и время	8 байт
Денежный	Денежные значения	8 байт
Счетчик	Уникальное числовое значение, которое **MS Office Access 2007** автоматически вводит при добавлении записи	4 байта или 16 байт, если используется для кода репликации
Логический	Логические значения	1 бит (8 бит = 1 байт)
Поле объекта	**OLE**-объекты или дру-	До 1 Гбайта

OLE	гие двоичные данные	

Окончание табл. 1.2

Тип данных	Объект хранения	Размер
Вложение	Рисунки, изображения, двоичные файлы, файлы **Microsoft Office**	Для сжатых вложений – 2 Гбайта. Для несжатых вложений – примерно 700 кбайт в зависимости от степени возможного сжатия вложения
Гиперссылка	Гиперссылки	До 1 Гбайта знаков или 2 Гбайт памяти (2 байта на знак), из которых в элементе управления можно отобразить 65 535 знаков
Мастер подстановок	Фактически типом данных не является, а вызывает мастер подстановок	На основе таблицы или запроса – размер привязанного столбца; на основе значения – размер текстового поля, содержащего значение

При создании новой базы данных в нее автоматически вставляется новая пустая таблица с именем **Таблица1**. Для ее сохранения необходимо вызвать контекстное меню и нажать **Сохранить**. Откроется окно **Сохранение**, в котором нужно указать имя таблицы и нажать кнопку **OK**. В нашем случае укажем имя таблицы **GoodsCatalog**.

Для добавления в базу данных новых таблиц на вкладке **Создание** (рис. 1.4) можно выбрать один из возможных способов создания таблиц, например **Таблица** или **Конструктор таблиц**.

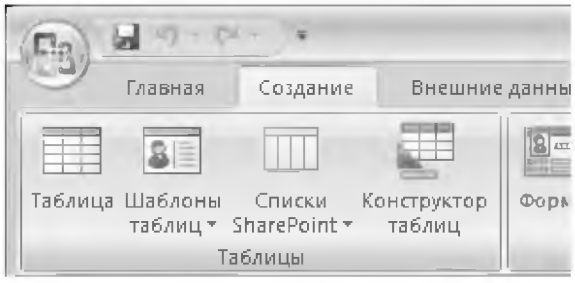


Рис. 1.4. Режимы создания таблиц

Для детальной настройки структуры таблицы воспользуемся режимом **Конструктор**. Для этого щелкнем на пиктограмме **Режим** и выберем режим **Конструктор** (рис. 1.5). Если таблица не была сохранена, то при переходе в режим **Конструктор** откроется окно

Сохранение. При сохранении укажем имя **GoodsCatalog**, которое соответствует таблице **Товары**.

Теперь создадим структуру для таблицы.

В первую строку колонки **Имя поля** введем код товара (**IDgc**) и нажмем клавишу **Enter**. Курсор переместится в колонку **Тип данных**, где уже будет установлено значение **Счетчик**, поскольку мы перешли туда из режима **Таблица**. При создании таблицы посредством команды **Конструктор таблиц** на вкладке **Создание** по умолчанию будет назначен тип данных **Текстовый**.

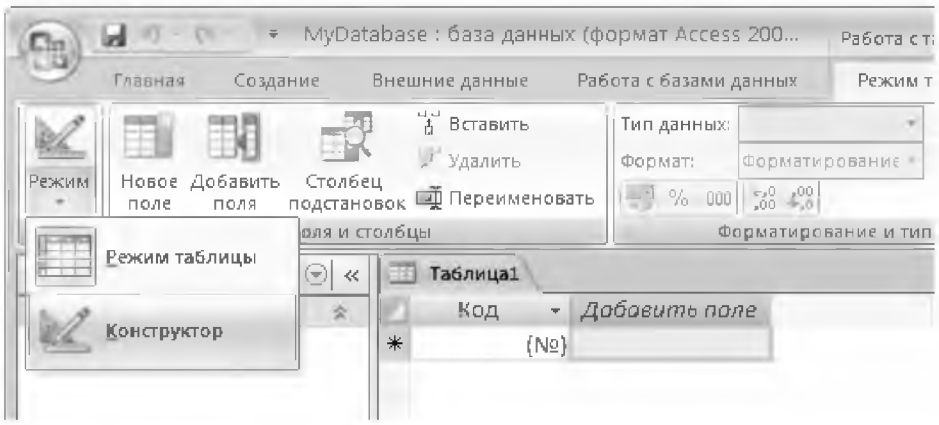


Рис. 1.5. Смена режимов работы

Первой строке таблицы (**IDgc**-поле кода товара) при трансформации из режима **Таблица** по умолчанию назначается поле первичного ключа. В случае если таблица создавалась сразу в режиме **Конструктор**, для **IDgc**-поля необходимо назначить флаг ключевого поля. Для этого в контекстном меню выберем пункт **Ключевое поле**. Для первичного ключа в свойствах поля установим значение индексированного поля **Да (Совпадения не допускаются)**. Далее заполним вторую строку (второе поле таблицы): имя поля – **gcName**, тип данных – **Текстовый**; третью строку: имя поля – **IDtg**, тип данных – **Числовой**; четвертую строку: имя поля – **gcMeasure**, тип данных – **Текстовый**. При этом для имени полей **gcName** и **gcMeasure** в разделе **Свойства поля** установим параметр **Размер поля**, равным 50 и 20 соответственно. Пример настройки структуры таблицы **GoodsCatalog** представлен на рис. 1.6.

Для полей основных числовых типов полей **Числовой** и **Счетчик** характерны дополнительные свойства, такие как **Размер поля** (**FieldSize**) и **Формат** (**Format**).

Описание возможных значений свойства для этих типов приведено в табл. 1.3.

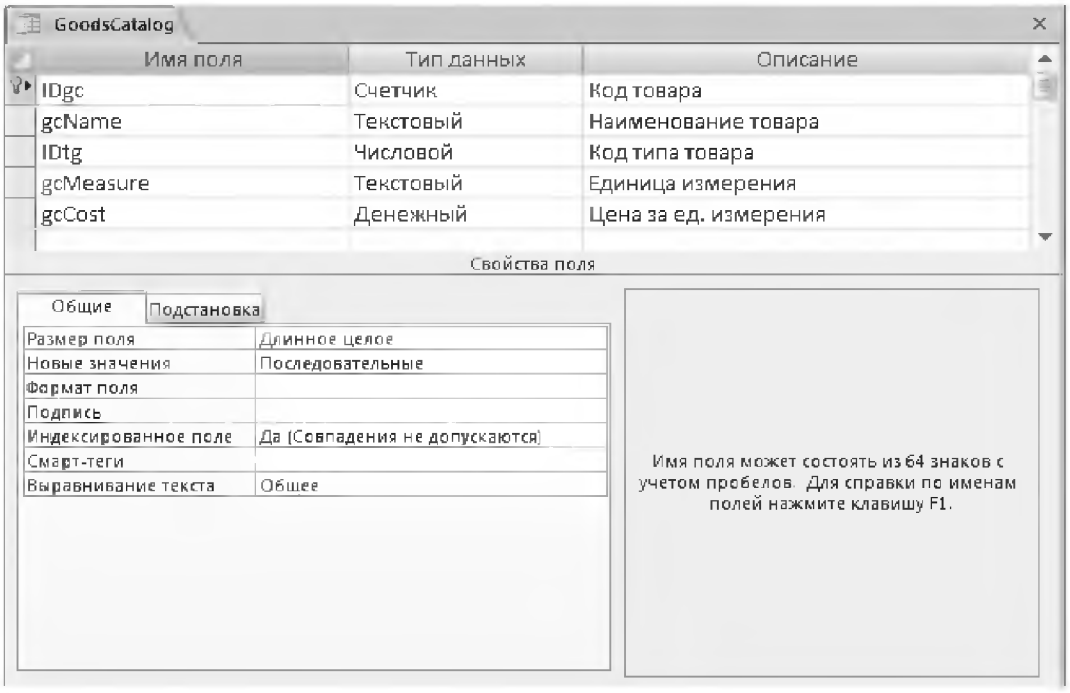


Рис. 1.6. Область редактирования структуры таблицы

Таблица 1.3

Размер поля для числового формата и счетчика

Значение свойства	Описание	Размер
	Тип ***Числовой***	
Байт	Числовые значений в диапазоне от 0 до 255 знаков	1 байт
Целое	Значения в диапазоне от –32 768 до +32 768	2 байта
Длинное целое	Значения в диапазоне от –2 147 483 648 до +2 147 483 647	4 байта
Одинарное с плавающей точкой	Значения с плавающей запятой в диапазоне от $-3{,}4\cdot10^{38}$ до $+3{,}4\cdot10^{38}$, включающих до семи значащих разрядов	4 байта
Двойное с плавающей точкой	Значения с плавающей запятой в диапазоне от $-1{,}797\cdot10^{308}$ до $+1{,}797\cdot10^{308}$, включающие до 15 значащих разрядов	8 байт
Код репликации	Глобальный уникальный идентификатор, необходимый для репликации. При использовании формата файла **.accdb** репликация не поддерживается	16 байт

Действительное	Значения в диапазоне от $-9{,}999 \cdot 10^{27}$ до $+9{,}999 \cdot 10^{27}$	12 байт

Окончание табл. 1.3

Значение свойства	Описание	Размер
	Тип ***Счетчик***	
Длинное целое	Значения в диапазоне от 1 до +2 147 483 648, если для свойства поля **Новые значения** задано значение **Последовательные**, и от –2 147 483 648 до +2 147 483 647, если для свойства поля **Новые значения** задано значение **Случайные**	4 байта
Код репликации	Глобальный уникальный идентификатор, необходимый для репликации. При использовании формата файла **.accdb** репликация не поддерживается	16 байт

Дополнительные сведения о свойствах полей **Формат** для данных числового типа и типа **Дата/время** приведены в табл. 1.4.

Таблица 1.4

Формат поля для чисел и дат

Значение свойства	Описание
	Тип ***Числовой***
Основной	Число отображается в том виде, в каком оно было введено. Например, 3456,789 отображается как **3456,789**
Денежный	При отображении числа используется разделитель групп разрядов и параметры, заданные в компоненте **Язык и региональные стандарты** панели управления для отрицательных значений, обозначения денежной единицы, разделителя целой и дробной части и количества дробных знаков. Например, 3456,789 отображается как **3456,79р**
Евро	При отображении числа используется обозначение денежной единицы евро независимо от того, что указано в компоненте **Язык и региональные стандарты**
Фиксированный	Отображается хотя бы одна цифра и применяются параметры, заданные в компоненте **Язык и региональные стандарты** панели управления для отрицательных значений, обозначения денежной единицы, разделителя целой и дробной части и количества дробных знаков. Например, 3456,789 отображается как **3456,79**
Стандартный	При отображении числа используется разделитель групп разрядов и параметры, заданные в компоненте **Язык и региональные стандарты** панели управления для отрицательных значений, разделителя целой и дробной части и количества дробных знаков. В этом формате обозначение

	денежной единицы не используется. Например, 3456,789 отображается как **3 456,79**

Окончание табл. 1.4

Значение свойства	Описание
Научный	Значение отображается в стандартном экспоненциальном представлении. Например, значение 3 456,789 отображается как **3,46E+03**
Процентный	Значение умножается на 100 и к полученному значению добавляется знак процентов. Применяются параметры, заданные в компоненте **Язык и региональные стандарты** панели управления для отрицательных значений, разделителя целой и дробной части и количества дробных знаков. Например, 0,3456 отображается как **35%**
Тип ***Дата/время***	
Полный формат даты	При отображении значения используется сочетание форматов **Краткий формат даты** и **Длинный формат времени**
Длинный формат даты	При отображении значения используется полный формат даты, заданный в компоненте **Язык и региональные стандарты** панели управления
Средний формат даты	При отображении значения используется формат **дд-ммм-гг**, например, **15-июл-14**
Краткий формат даты	При отображении значения используется краткий формат даты, заданный в компоненте **Язык и региональные стандарты** панели управления
Длинный формат времени	При отображении значения используется формат времени, заданный в компоненте **Язык и региональные стандарты** панели управления
Средний формат времени	При отображении значения используется формат **ЧЧ:ММ РМ**, где **ЧЧ** – часы; **ММ** – минуты; **РМ** – **AM** или **PM**. Значение часов может находиться в диапазоне от 1 до 12, значение минут – в диапазоне от 0 до 59
Краткий формат времени	При отображении значения используется формат **ЧЧ:ММ**, где **ЧЧ** – часы; **ММ** – минуты. Значение часов может находиться в диапазоне от 1 до 23, значение минут – в диапазоне от 0 до 59

Зная особенности полей, мы сможем подобрать необходимый тип поля во время физической реализации базы данных. Теперь перейдем к рассмотрению не менее важной части проектирования – настройки индексных полей.

1.2.2. Настройка индексов

Индексы ускоряют поиск и сортировку записей в **Microsoft Access**. В индексе хранится местоположение записей на основе одного или нескольких полей, которые являются частью индекса. Сам индекс может быть гораздо меньше описываемой им таблицы (его размер зависит от количества уникальных значений проиндексированных полей), делая чтение данных более эффективным.

Создание индексов. Перед созданием индекса необходимо решить, следует ли реализовать индекс для одного поля или составной индекс. Индекс для одного поля создается в режиме **Конструктор** с помощью установки свойства **Индексированное поле**, которое может принимать следующие значения:

– **Нет** – не создавать индекс для этого поля (или удалить существующий индекс);

– **Да (Допускаются совпадения)** – создать индекс для этого поля;

– **Да (Совпадения не допускаются)** – создать уникальный индекс для этого поля.

При создании уникального индекса невозможно ввести новое значение в определенное поле, если такое значение уже существует в том же поле другой записи. В **Microsoft Access** уникальный индекс автоматически формируется для первичных ключей. Однако может понадобиться, чтобы ввод значений, совпадающих со значениями в других записях, был невозможным. Например, необходимо создать уникальный индекс для поля, в котором содержатся серийные номера товаров, чтобы двум продуктам не мог быть присвоен один и тот же серийный номер.

Чтобы создать составной индекс, в режиме **Конструктор** на вкладке **Конструктор** в группе **Показать или скрыть** выберем пиктограмму **Индексы**. В результате появится окно **Индексы** (рис. 1.7). При необходимости размеры этого окна можно изменять для того, чтобы отобразить пустые строки и свойства индекса.

В первой пустой строке столбца **Индекс** введем имя будущего индекса. Для индекса можно использовать либо имя одного из индексируемых полей, либо другое подходящее имя. В столбце **Имя поля** из выпадающего списка выберем первое поле, которое будет использоваться в индексе. Следующую строку столбца **Индекс** оставляем пустой, затем в столбце **Имя поля** укажем второе индексируемое

поле. Если нам потребуется добавить в индекс больше полей, то повторим эти действия нужное число раз.

При создании индекса все строки обрабатываются как часть одного индекса до тех пор, пока не будет обнаружена строка с другим названием индекса. Чтобы вставить строку (новое поле в существующий индекс), щелкнем правой кнопкой мыши на то место, куда следует вставить строку, а затем в контекстном меню выберем команду **Вставить строки**.

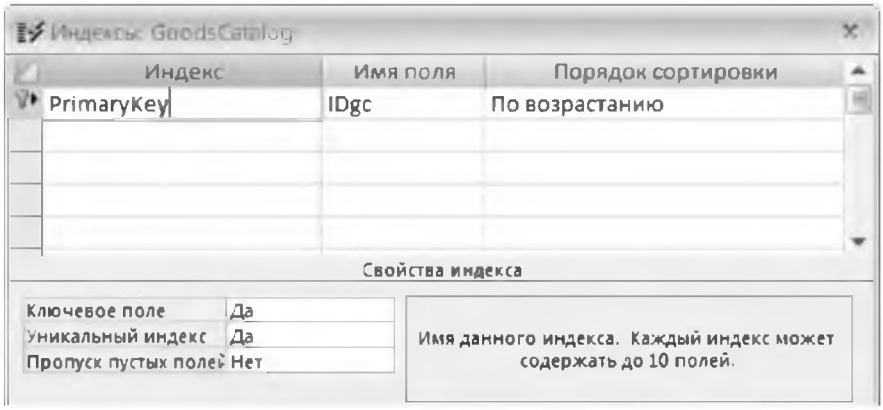


Рис. 1.7. Окно настройки индексов

Для изменения порядка сортировки значений полей в столбце **Порядок сортировки** окна **Индексы** выберем соответствующее значение: **По возрастанию** или **По убыванию**.

В области **Свойства индекса** окна **Индексы** можно установить следующие свойства индекса для строки в столбце **Имя индекса**:

– **Первичный** – значение **Да** показывает, что индекс является первичным ключом;

– **Уникальный** – значение **Да** показывает, что каждое индексируемое значение должно быть уникальным;

– **Пропуск пустых полей** – значение **Да** показывает, что записи с пустыми значениями в индексируемых полях будут исключены из индекса.

Для нашей базы данных индексы будут следующими: в таблице **GoodsCatalog** зададим индекс **GoodsName** с уникальными значениями для поля **gcName** и индекс **GoodsTypeAndName** для полей **IDtg**, **gcName** (рис. 1.8), в таблице **IncomingGoods** – индекс **IncomingDateGoods** для полей **igDate** и **IDgc**.

Удаление индекса. Если индекс становится ненужным или снижает производительность системы, то его можно удалить. Удаление затрагивает только сам индекс, а не поля, которые были в него включены. Для удаления индекса включим редактирование таблицы в режиме **Конструктор**, затем на вкладке **Конструктор** в группе **По-**

казать или скрыть нажмем на пиктограмму **Индексы**. В появившемся окне выделим строки, содержащие индекс, который следует удалить, и нажмем клавишу **Delete**. После этого сохраним изменения и закроем окно **Индексы**.

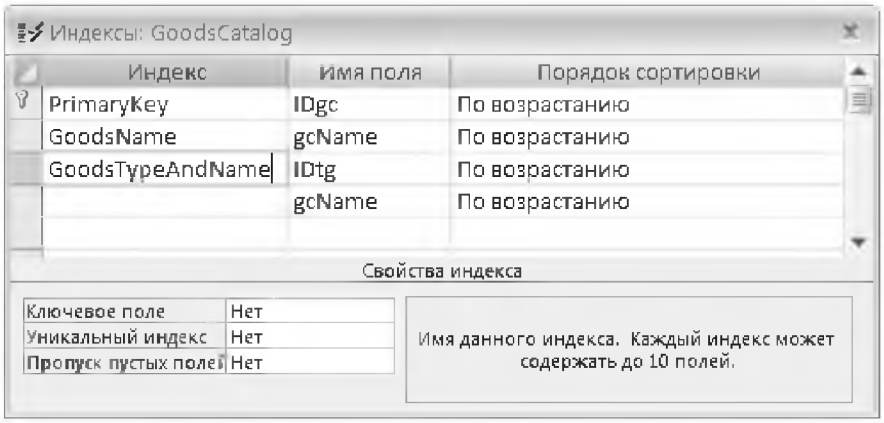


Рис. 1.8. Окно настройки индексов

Просмотр или редактирование индексов. Иногда может понадобиться просмотр индексов в таблице, чтобы оценить их влияние на производительность или убедиться в том, что необходимые поля индексированы. Для этого мы можем просмотреть или изменить индексы и свойства индексов в нужной таблице в соответствии с решаемыми задачами. Для сохранения изменения выполним команду **Сохранить** на панели быстрого доступа или воспользуемся сочетанием клавиш **CTRL + S**.

1.2.3. Установка логических связей для таблиц

Логические связи устанавливаются между одноименными полями таблиц базы данных **Microsoft Access**. Связь данных в одной таблице с данными в других таблицах осуществляется через уникальные идентификаторы (ключи) или ключевые поля. В нашем случае мы должны установить логические связи между таблицами **Товары**, **Тип товаров**, **Поставщики** и **Приход товаров** (см. рис. 1.1).

Прежде чем приступить к созданию логических связей, в окне редактирования закроем все таблицы и перейдем на вкладку **Работа с базами данных**, затем щелкнем на пиктограмме **Схема данных**. В окне редактирования появится активное диалоговое окно

Добавление таблицы на фоне неактивного окна **Схема данных** (рис. 1.9).

В окне **Добавление таблицы** выделим имена таблиц и нажмем кнопку **Добавить**, при этом в окне **Схема данных** появятся все выделенные таблицы. После этого закроем окно диалога и разместим блоки, отображающие структуру таблиц так, чтобы с ними было удобнее работать (рис. 1.10).

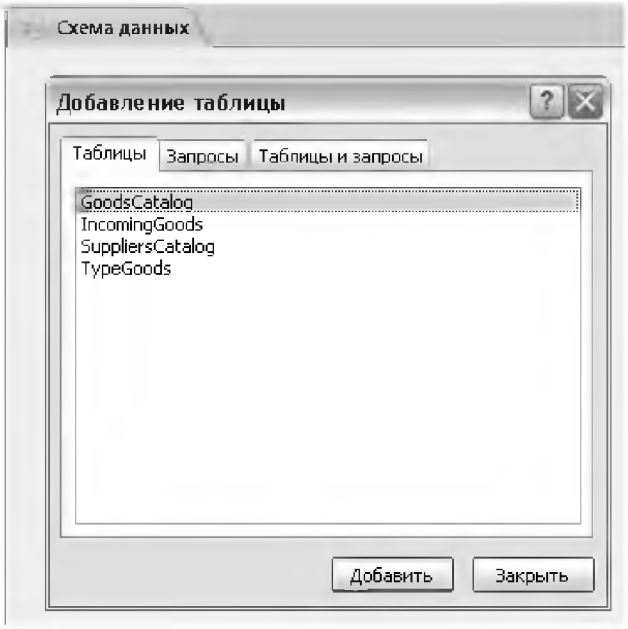


Рис. 1.9. Диалоговое окно **Добавление таблицы**

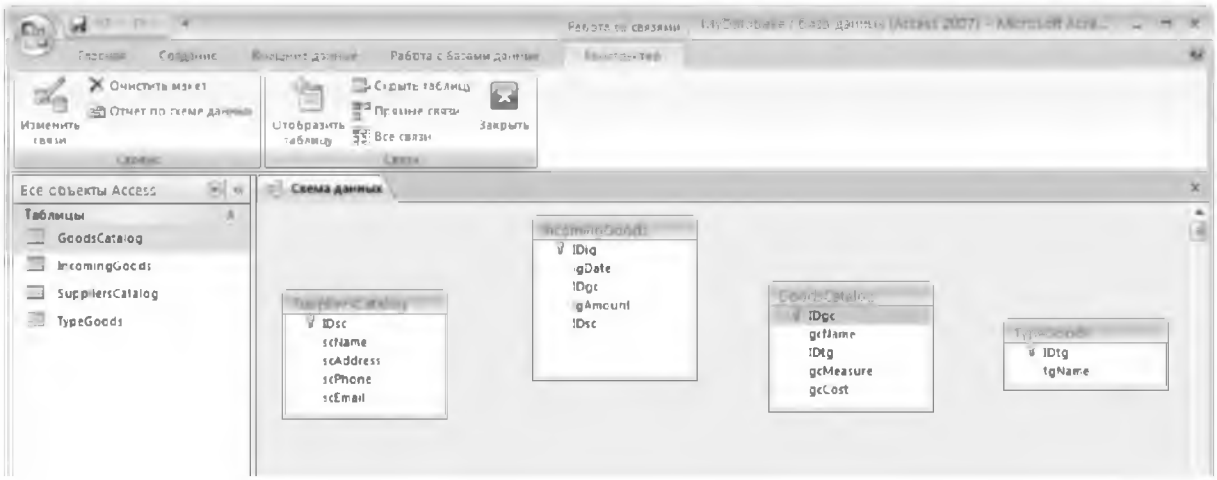


Рис. 1.10. Фрагмент экрана **Microsoft Access** с отображенной схемой данных

Далее установим связи между таблицами в окне **Схема данных**. Для этого нажмем левую кнопку мыши на поле **IDtg** из таблицы **TypeGoods** и переместим его на соответствующее поле таблицы

GoodsCatalog. В результате выполнения этой операции появится окно **Изменение связей** (рис. 1.11).

В этом окне установим флажок **Обеспечить целостность данных**, убедимся в том, что задан тип отношений **Один-ко-многим**, а затем нажмем кнопку **Создать**.

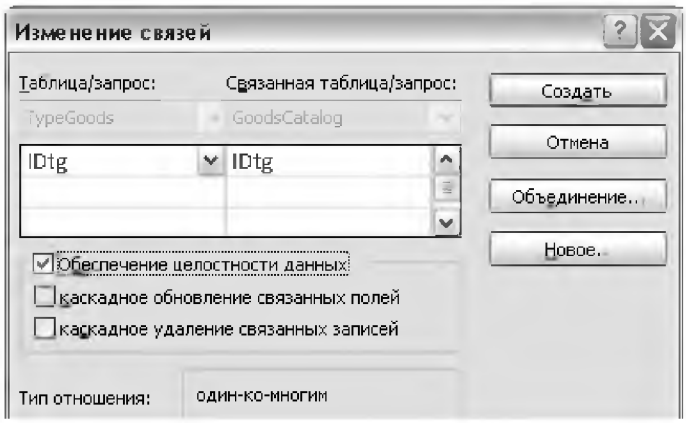


Рис. 1.11. Окно изменения связей

В окне **Изменение связей** можно установить еще два флажка: **Каскадное обновление связанных полей**, который при изменении первичного ключа позволяет автоматически обновлять все поля, ссылающиеся на этот первичный ключ, и **Каскадное удаление связанных записей**, который при удалении записи в таблице первичного ключа автоматически удаляет все записи со ссылкой на данный первичный ключ.

После выполнения этих действий в окне **Схема данных** появится связь **Один-ко-многим** между таблицами **TypeGoods** и **GoodsCatalog**. Аналогичным образом нужно связать ключевые поля **IDsc** в таблицах **SuppliersCatalog** и **IncomingGoods**, а затем поля **IDgc** в таблицах **GoodsCatalog** и **IncomingGoods** (рис. 1.12).

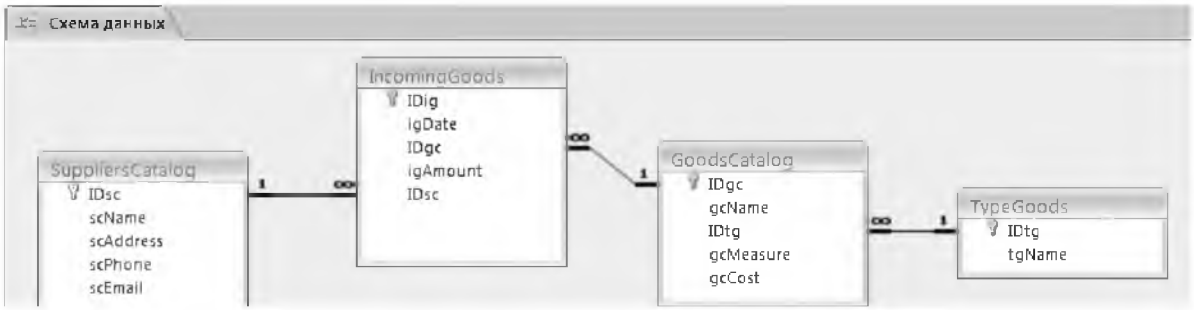


Рис. 1.12. Схема данных с учетом расставленных связей

В итоге будет получена схема данных, представляющая модель, которая соответствует поставленной задаче. Теперь мы можем приступить к наполнению базы данных.

1.2.4. Заполнение таблиц базы данных в Microsoft Access

Таблицы в **Microsoft Access** необходимо наполнять данными в определенной последовательности: сначала заполняются таблицы, существование которых не зависит от других таблиц, а затем те таблицы, которые зависят от этих таблиц. Ввод данных в таблицы для нашей БД целесообразно выполнять в такой последовательности: **TypeGoods**, **SuppliersCatalog**, **GoodsCatalog** и **IncomingGoods**.

Пример заполнения таблицы **TypeGoods** представлен на рис. 1.13. В этой таблице необходимо указать любые 3–5 типов товаров. Для таблицы **SuppliersCatalog** нужно сделать 4–6 записей.

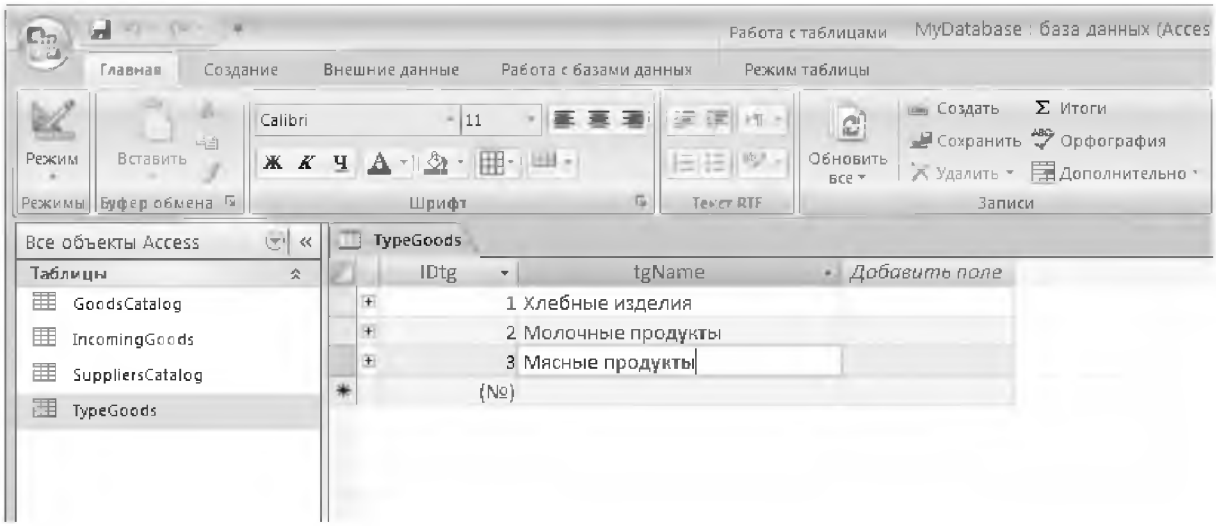


Рис. 1.13. Пример заполнения таблицы **TypeGoods**

При заполнении таблиц, в которых имеются связи с другими таблицами через поля-идентификаторы (в нашем случае это таблицы **GoodsCatalog** и **IncomingGoods**), в этих полях будут отображаться числовые значения (рис. 1.14).

Рис. 1.14. Пример заполнения таблицы **GoodsCatalog**

Если в справочной таблице имеется большое число записей, то будет очень трудно запомнить, какие значения следует вводить

в поле. Для более удобного заполнения полей, организующих связь (в нашем случае это поля **IDtg**, **IDsc**, **IDgc**), можно использовать раскрывающийся список данных, который будет показывать текстовое отображение вводимых значений. Но этот список может появиться только в том случае, если при создании структуры таблиц в режиме **Конструктор** для полей был настроен режим подстановки.

Рассмотрим настройку режима подстановки для поля **IDtg** таблицы **GoodsCatalog**.

Сначала войдем в режим **Конструктор** для настройки структуры таблицы. Выберем поле **IDtg** и в области **Свойства поля** переключимся на вкладку **Подстановка** (рис. 1.15).

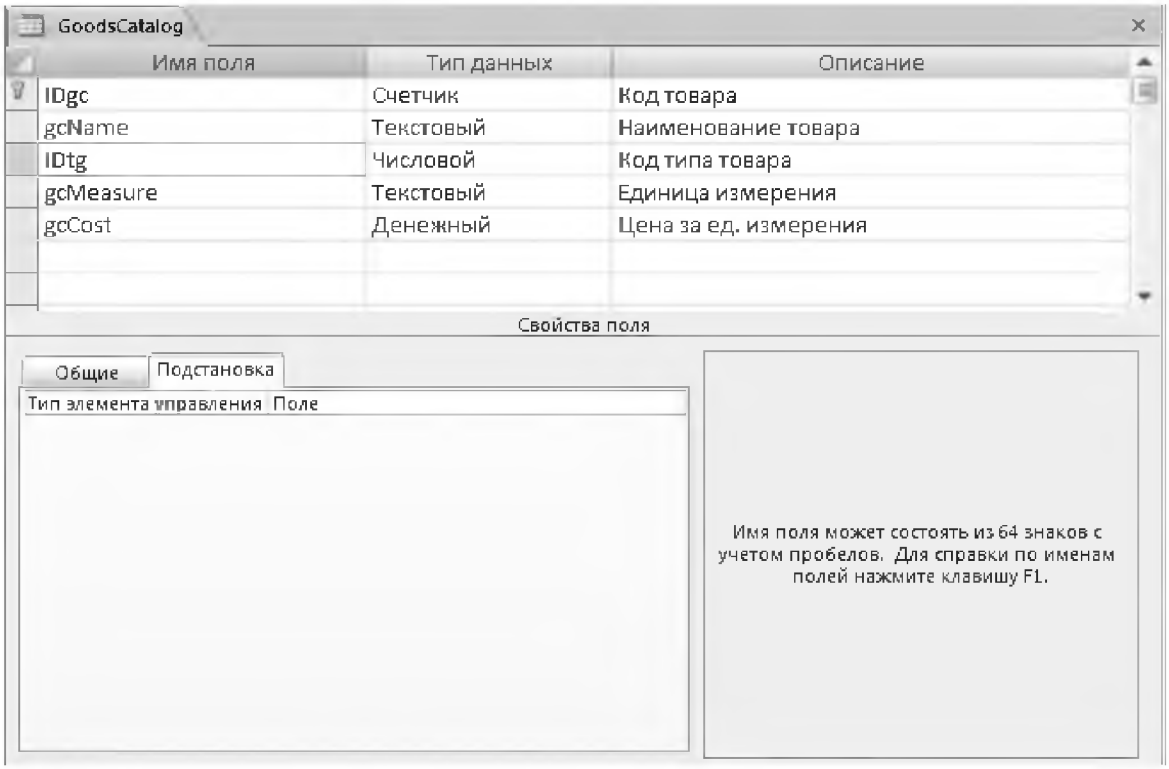


Рис. 1.15. Параметры подстановки по умолчанию (тип подстановки – **Поле**)

Для настройки режима подстановки в списке выбора для свойства **Тип элемента управления** выберем **Список** или **Поле со списком** (этот тип позволяет более детально настраивать режим отображения списка). В нашем случае для ввода первоначальных данных установим значение **Список**, при этом появится набор свойств, которые отображены на рис. 1.16.

В качестве типа источника по умолчанию указывается **Таблица или запрос**. Для того чтобы мы смогли отобразить сведения о типах

товара, нам необходимо настроить свойства следующим образом (рис. 1.17):

– **Источник строк** – **TypeGoods** (выбираем из списка). Здесь указываются таблица, запрос или список значений, являющиеся источником значений для столбца подстановок;

– **Присоединенный столбец** – оставляем без изменения, поскольку первым по счету идет поле, связывающее две таблицы. Здесь указывается столбец в свойстве **Источник строк**, из которого подставляется значение, хранящееся в столбце подстановок. Оно может изменяться от 1 до количества столбцов в свойстве **Источник строк**;

– **Число столбцов** – установим значение 2. Здесь указывается число столбцов из источника строк, которые могут отображаться в столбце подстановок;

– **Ширина столбцов** – установим значение **0см;6см**. Здесь настраивается ширина каждого столбца. Если нет необходимости в отображении какого-либо столбца (например, идентификатора **IDig)**, то задается ширина 0 см. Значения ширины для полей отделяются точкой с запятой.

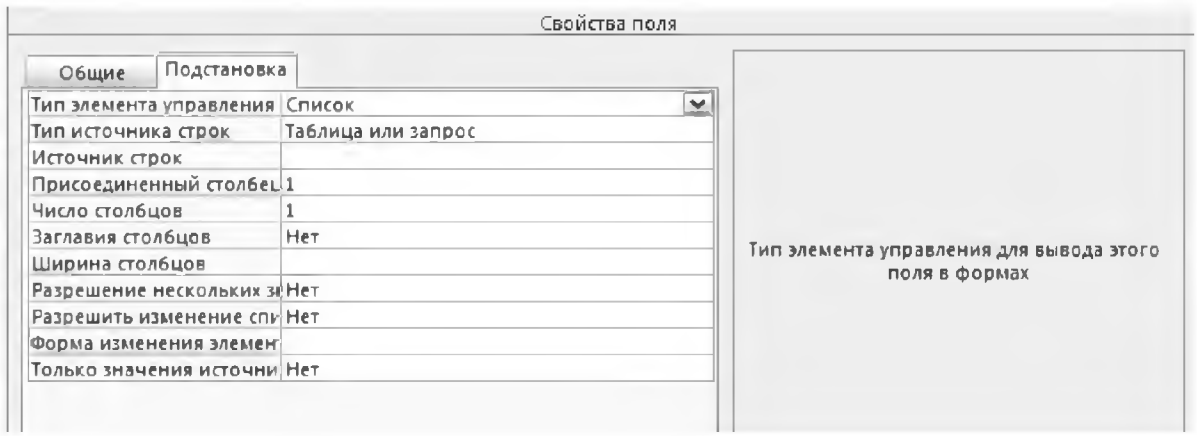


Рис. 1.16. Свойства параметров подстановки по умолчанию (тип подстановки – **Список**)

Теперь мы можем вернуться к добавлению записей в таблицу, переключившись в режим **Таблица**. Вместо цифр в поле **IDtg** у нас будут отображаться подставляемые значения, выбор которых будет осуществляться на основе раскрывающегося списка данных (рис. 1.18).

Заполним таблицу **GoodsCatalog**, добавив 6–10 записей; в таблицу **IncomingGoods** внесем 7–12 записей.

Необходимо отметить, что в базах данных **Microsoft Access** применяются различные методы перемещения по таблице: с помощью клавиш управления курсором, кнопок из области **Запись**, расположенных внизу таблицы в режиме таблицы, команды **Перейти** в группе **Найти** вкладки **Главная**, расположенной на элементе управления **Лента**.

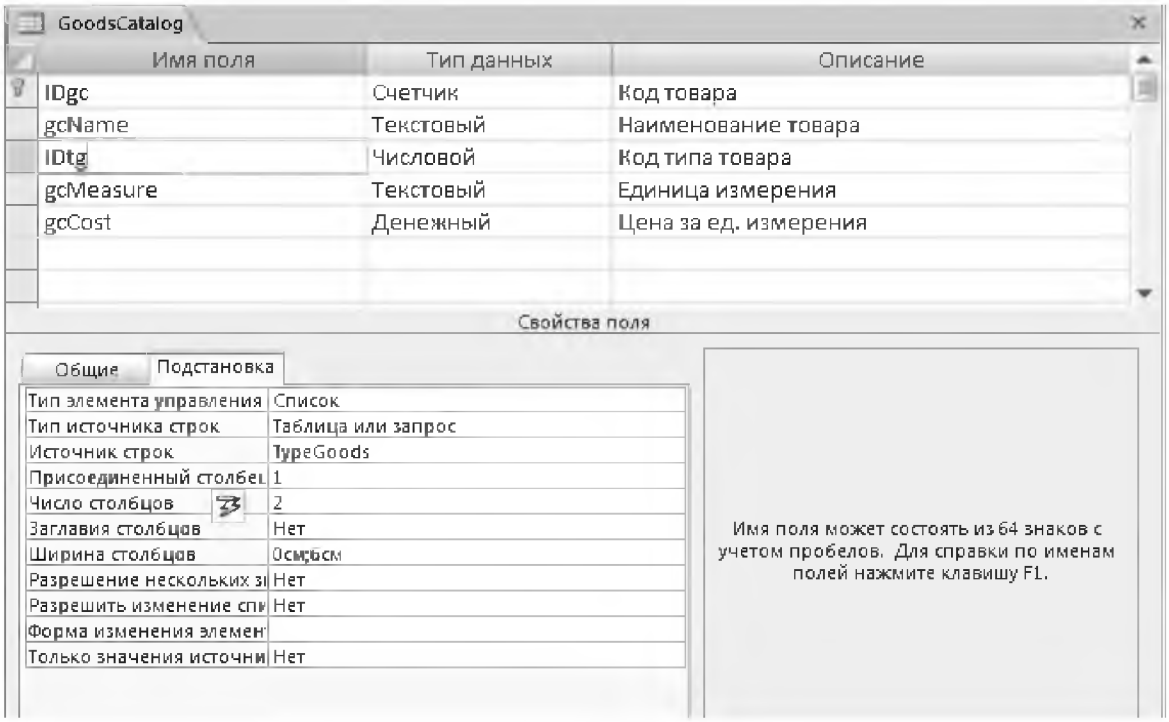


Рис. 1.17. Настроенные параметры подстановки для поля **IDtg**

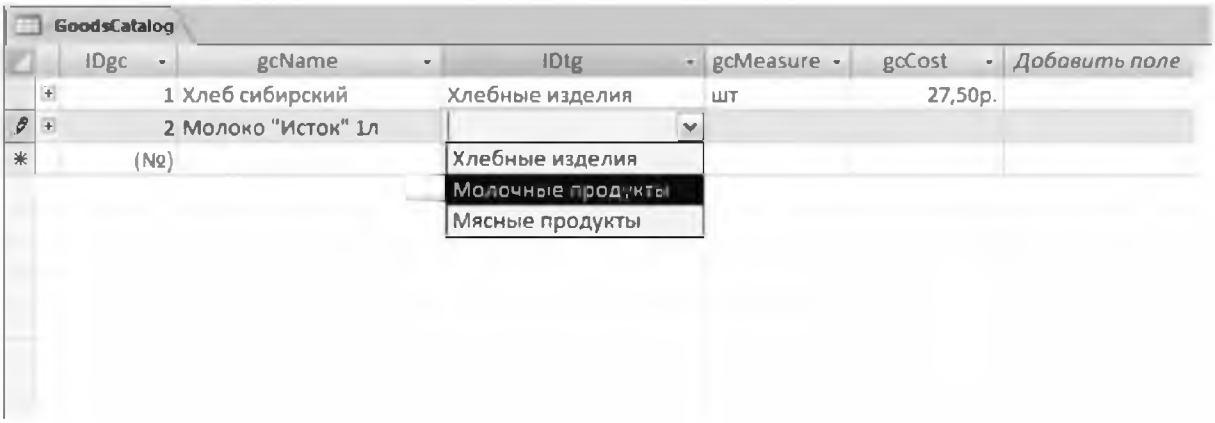


Рис. 1.18. Пример заполнения таблицы **GoodsCatalog**

Для перемещения от поля к полю слева направо применяются клавиши управления курсором →, **Tab** и **Enter**, а в обратном направлении – ← и сочетание клавиш **Shift** + **Tab**.

1.3. Разработка клиентских приложений в среде RAD Studio

Результатом этапа разработки клиентских приложений является готовый продукт, позволяющий пользователям вводить данные в таблицы либо редактировать уже существующие данные, анализировать введенные данные и представлять их в более удобном для восприятия виде графиков, сводных таблиц или отчетов. В данной лабораторной работе мы рассмотрим создание простых клиентских приложений, состоящих из одной формы.

1.3.1. Создание простейшего приложения

Для работы с нашей базой данных создадим новый проект. Для этого в меню **File** выберем пункт **New → VCL Forms Application – Delphi**. Сохраним его в соответствующем каталоге, например в каталоге **C:\User\Group\Surname\Lab1**.

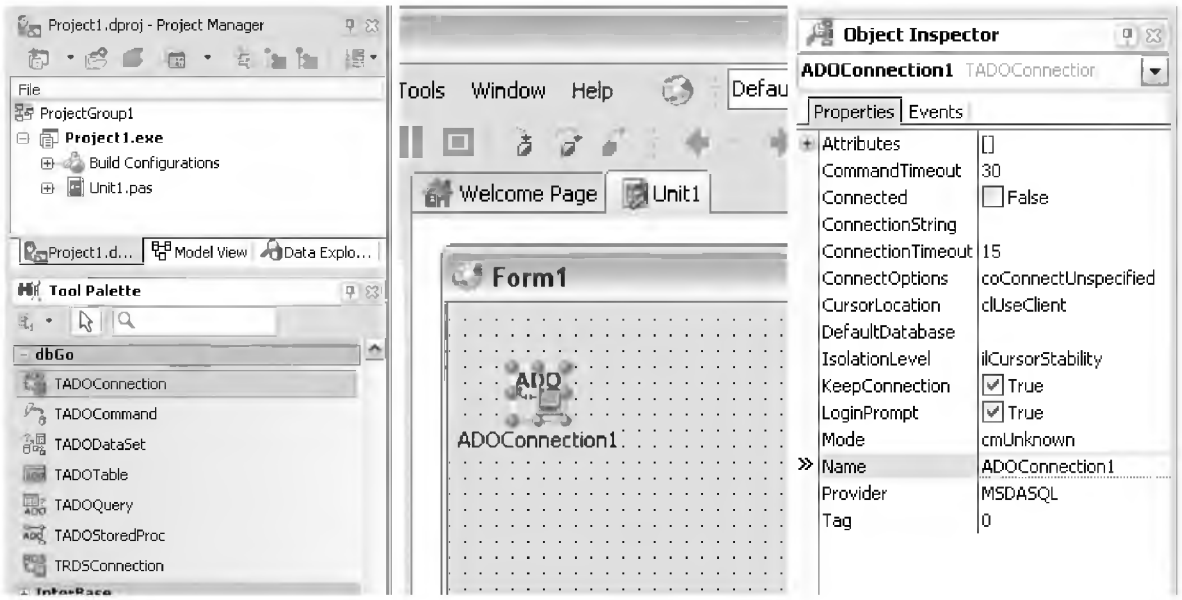
Для организации соединения с базой данных воспользуемся технологией **ADO***, взяв для этого невизуальный компонент **ADOConnection**, расположенный на вкладке **dbGO** палитры компонентов **Tool Pallete** (рис. 1.19, *а*). Выберем его мышью и разместим на форме (рис. 1.19, *б*), при этом в инспекторе объектов **Object Inspector** на вкладке **Properties** будут отображены все основные свойства выбранного компонента (рис. 1.19, *в*). По умолчанию вид инспектора объектов может отличаться от того, который показан на рис. 1.19, *в*. Для отображения свойств в алфавитном порядке необходимо выбрать пункт контекстного меню **Arrange → by Name**.

Для настройки соединения с базой данных выберем свойство **ConnectionString** и нажмем кнопку с многоточием. При этом появится окно формирования строки подключения (рис. 1.20), в котором нажмем на кнопку **Build…**.

В появившемся окне настройки источника соединения в зависимости от типа файла выберем соответствующий **OLE DB**-провайдер. В нашем случае это **Microsoft Office 12.0 Access Database Engine OLED DB Provider** (рис. 1.21), который открывает больше типов файлов **Microsoft Access** (рис. 1.21, *а*). Сделав выбор, нажмем на кнопку **Next** >>. В открывшейся вкладке настроим параметр **Data**

* Описание этой технологии представлено в кн.: Сухарев М. Delphi. Полное руководство. Включая версию 2010. М. : Наука и техника, 2010.

Source, в котором необходимо указать путь к файлу базы данных (рис. 1.21, *б*).



а *б* *в*

Рис. 1.19. Фрагменты рабочих областей среды разработки:
а – менеджер проекта (**Project Manager**) и палитра компонентов;
б – фрагмент формы с установленным компонентом **TADOConnection**;
в – инспектор объектов, отображающий свойства компонента **ADOConnection1**

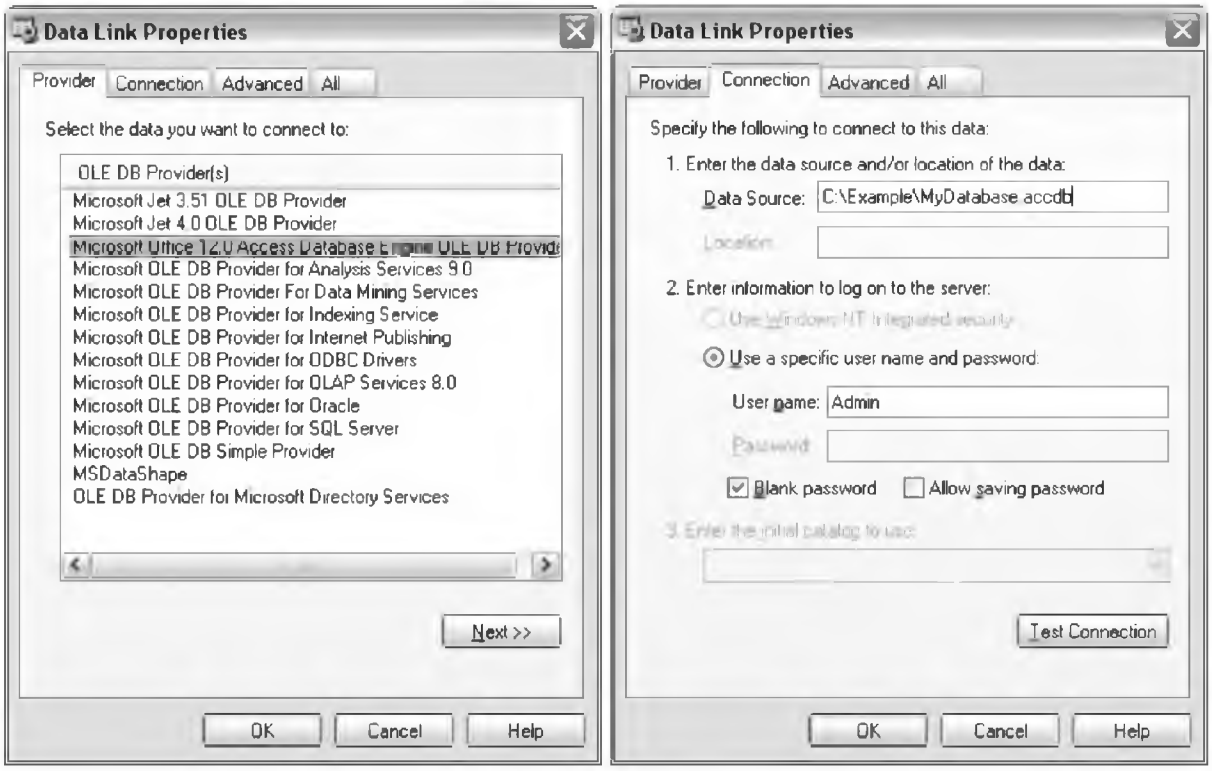
Рис. 1.20. Окно формирования строки подключения

По завершению настройки можно использовать кнопку **Test Connection** для проверки работы соединения. В случае успешного результата несколько раз подряд нажмем на кнопку **OK**.

Для того чтобы при каждом подключении к базе данных у нас не запрашивалось имя пользователя и пароль, свойство **LoginPrompt** нужно установить в состояние **false**.

После настройки соединения разместим на форме невизуальный компонент **TADOTable**, который, как и компонент **ADOConnection**,

расположен на вкладке **dbGO** палитры компонентов. Присвоим свойству **Connection** значение **ADOConnection1** при помощи выпадающего списка. Далее зададим свойство **TableName** (имя таблицы БД) при помощи выпадающего списка в значение **TypeGoods**. После этого установим свойство **Active** в значение **True**. В результате произойдет связывание компонента **ADOTable1** с реально существующей таблицей **TypeGoods**.



а *б*

Рис. 1.21. Окно настройки источника соединения:
а – вкладка выбора **OLE DB**-провайдера; *б* – вкладка настройки подключения провайдера **Microsoft Office 12.0 Access Database Engine OLED DB Provider**

Компоненты **TADOTable** и **TADOQuery** (его мы рассмотрим далее в п. 1.3.5) служат для хранения наборов данных (НД). Понятие набора данных шире, чем понятие таблицы БД, поскольку набор данных может содержать:

– подмножество записей или полей таблицы базы данных (компоненты **TADOTable**, **TADOQuery**, а также **TTable**, **TQuery**);

– записи из нескольких таблиц базы данных (компоненты **TADOQuery**, **TQuery**).

Расположим на форме компонент **TDataSource**, который можно взять на странице палитры компонентов **Data Access**. Он служит связующим звеном между невизуальными компонентами (в данном слу-

чае компонентом **ADOTable1**) и визуальными компонентами, которые мы добавим в форму позднее. Поэтому компоненты **TDataSource** часто называют *источниками данных*. Установим свойство **DataSet** компонента **DataSource1** в значение **ADOTable1**, выбрав его из выпадающего списка.

Далее разместим на форме компонент **TDBGrid**, который служит для отображения записей набора данных в табличной форме, взяв его из палитры компонентов на вкладке **DataControls**. Установим свойство **DataSource** компонента **DBGrid1** в значение **DataSource1**.

Сохраним проект, используя элемент меню **Save All** или одноименную пиктограмму на панели инструментов, и откомпилируем приложение с помощью элемента меню **Run→Run** или клавиши **F9**.

Добавлять записи в набор данных и, соответственно, в таблицу **TypeGoods** можно непосредственно из компонента **TDBGrid**. Для этого нажмем на клавиатуре клавишу **Insert** или, находясь на последней записи набора данных, клавишу управления курсором ↓. Набор данных автоматически перейдет в режим добавления новой записи. После ввода значений в поля записи можно запомнить запись в наборе данных, перейдя на другую запись при помощи клавиш управления курсором. Отказаться от запоминания записи можно, нажав клавишу **Esc**. Следует отметить, что в поля с типом **Счетчик** нельзя вводить данные, тем не менее при сохранении записи им будет автоматически присвоено новое значение.

Для изменения записи переместим указатель текущей записи в нужное место и изменим значения там, где это необходимо. Набор данных автоматически перейдет в режим редактирования. Для удаления записи следует установить на нее указатель текущей записи и нажать комбинацию клавиш **Ctrl + Del**.

1.3.2. Создание приложения для работы с двумя таблицами

Создадим приложение для работы с двумя связанными таблицами. Воспользуемся прототипом проекта из п. 1.3.1. Добавим в приложение компонент **TADOTable** с именем **ADOTable2** для работы с таблицей **GoodsCatalog** (порядок установки и значения свойств аналогичны приведенным в п. 1.3.1 для компонента **ADOTable1**, но свойство **TableName** ссылается на имя таблицы **GoodsCatalog**). Установим свойство **ADOTable2.Active** в значение **True**. Добавим на форму

компонент **TDataSource** с именем по умолчанию **DataSource2**. Установим свойство **DataSet** этого компонента в значение **ADOTable2**. Разместим в форме компонент **TDBGrid** с именем по умолчанию **DBGrid2** и установим его свойство **DataSource** в значение **DataSource2**.

Запустим приложение на выполнение. Ввод и изменение данных в таблице **GoodsCatalog** будем производить при помощи компонента **TDBGrid**. Сделаем копию разработанного проекта, сохранив его в соответствующей папке, например в папке **C:\User\Group\Surname\Lab1_1**, которая понадобится нам в дальнейшем в п. 1.3.6.

1.3.3. Уточнение списка полей и настройка параметров столбцов в компоненте TDBGrid. Смена активного индекса

Значение поля **IDgc** обеспечивает уникальность записей в таблице **GoodsCatalog**, при этом не неся другой смысловой нагрузки. В связи с этим данное поле лучше не приводить в составе столбцов **DBGrid2**. Для этой цели сформируем список полей таблицы **GoodsCatalog**. В среде **Delphi** имеются два способа указать, какие из полей таблицы базы данных следует употребить для набора данных (в нашем случае – для компонента **ADOTable2**).

Первый способ состоит в использовании по умолчанию всех полей из таблицы БД, с которыми ассоциирован набор данных. Этот способ всегда действует по умолчанию и, следовательно, применялся нами неявно при создании наборов данных **ADOTable1** и **ADOTable2**.

Второй способ реализуется с помощью подмножества полей таблицы БД, с которой ассоциирован набор данных. Этим занимается редактор полей набора данных, который позволяет включить в состав отображаемых полей все поля или подмножество полей таблицы базы данных (рис. 1.22).

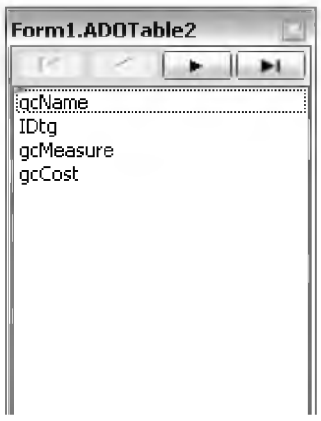


Рис. 1.22. Редактор полей **Field Editor**

Воспользуемся прототипом проекта из п. 1.3.2. Выберем при помощи мыши компонент **ADOTable2**. В контекстном меню выберем элемент **Fields Editor**. Далее в списке редактора полей, пока он пуст, нажмем на правую кнопку

мыши и во всплывающем меню выберем элемент меню **Add fields**. При этом будет показан список всех полей таблицы **GoodsCatalog**. Отметим при помощи мыши и клавиши **Shift** все поля, кроме поля **IDgc**, и нажмем на кнопку **OK**. Теперь список редактора полей будет включать все отмеченные поля, а в составе столбцов в компоненте **DBGrid2** будут присутствовать только те поля, которые добавлены для набора данных **ADOTable2** в редакторе полей (см. рис. 1.22).

Определение списка полей для набора данных в редакторе полей приводит к тому, что для каждого добавленного поля среда **Delphi** автоматически создает компонент **TField** (поле набора данных). Каждый такой компонент по умолчанию именуется уникальным именем: в качестве первой составляющей имени поля берется имя набора данных – **ADOTable2**, в качестве второй составляющей – имя поля в таблице. Так, например, компонент **TField**, соответствующий полю **gcName**, будет именоваться как **ADOTable2gcName**. Теперь в инспекторе объектов можно установить или изменить свойства поля, а также определить для него обработчики события.

Изменим параметры компонента **DBGrid2** так, чтобы названия его столбцов содержали русские наименования и были приемлемой ширины. Для этого щелкнем правой кнопкой мыши на компоненте **DBGrid2** и во всплывающем меню выберем элемент **Columns Editor**. На экране появится окно редактора столбцов компонента. Для изменения характеристик столбцов в компоненте **TDBGrid** перейдем от неявно определяемых столбцов к явно определяемым. Для этого щелкнем на кнопке **Add All Fields**, в результате чего будут добавлены столбцы, каждый из которых соответствует полю, заданному в редакторе полей компонента **ADOTable2** (рис. 1.23).

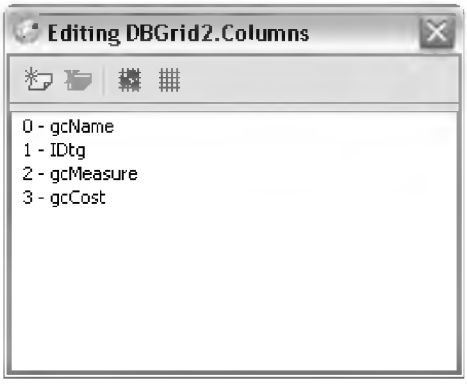


Рис. 1.23. Редактор столбцов **Columns Editor**
для компонента **DBGrid2**

Чтобы изменить заголовок каждого столбца, выберем при помощи мыши имя столбца в редакторе столбцов и раскроем список свойства **Title** в инспекторе объектов. В элементе **Caption** этого списка содержится заголовок столбца. Для изменения ширины столбца настроим параметр **Width**. Изменим соответствующим образом заголовки и при необходимости – параметр ширины столбцов, затем выйдем из редактора столбцов **DBGrid2**. Те же действия проделаем и для набора данных **ADOTable1**. Полученный после выполнения этих действий результат показан на рис. 1.24.

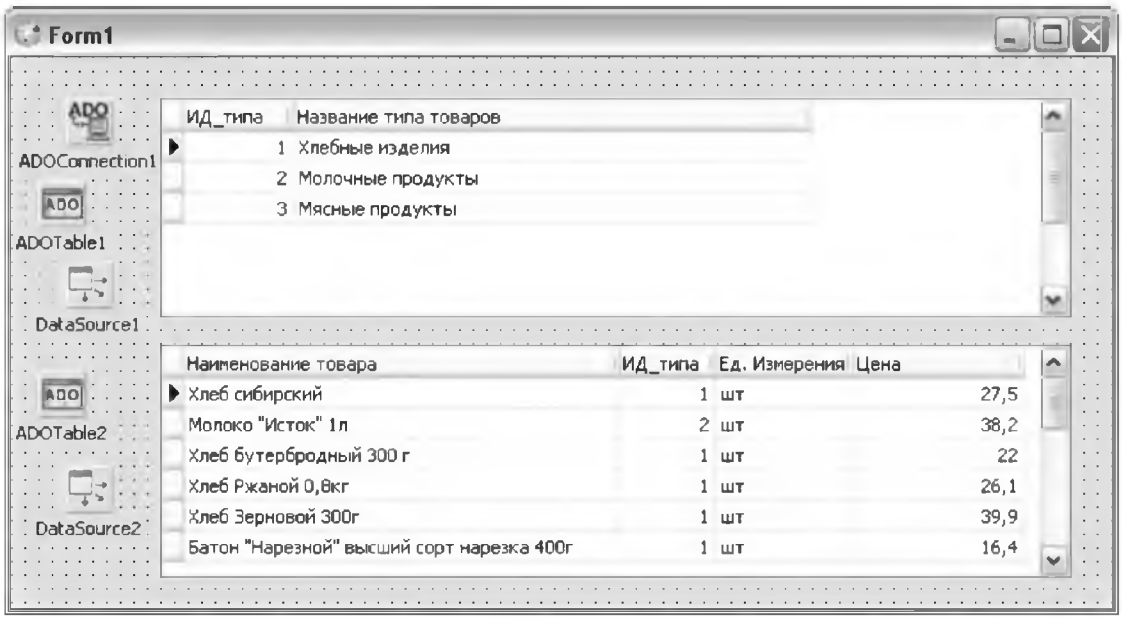


Рис. 1.24. Изменение заголовков столбцов

Изменим порядок сортировки записей в НД **ADOTable2**. В компоненте **TADOTable** существует два способа задания порядка сортировки на основе индекса.

Первый способ состоит в указании полей в свойстве **IndexFieldNames**, для которого нужно задать перечень индексных полей **IDtg;gcName**.

Второй способ использует имена индексов, которые были созданы в **Microsoft Access**. Для этого в инспекторе объектов необходимо выполнить следующие шаги:

1) перевести набор данных в неактивное состояние, установив параметру **Active** значение **false**;

2) выбрать имя индекса из списка в свойстве **IndexName** (в нашем примере – **GoodsTypeAndName**);

3) установить для свойства **TableDirect** значение **true**;

4) перевести набор данных в активное состояние, задав для параметра **Active** значение **true**.

В случае возникновения необходимости в смене индекса при использовании второго способа придется повторить шаги 1, 2 и 4; при применении первого способа достаточно будет указать новые значения индексных полей.

После этого войдем еще раз в редактор столбцов **DBGrid2** и при помощи мыши перенесем столбец **IDtg** так, чтобы он предшествовал столбцу **gcName**. Откомпилируем приложение и запустим его на выполнение (рис. 1.25).

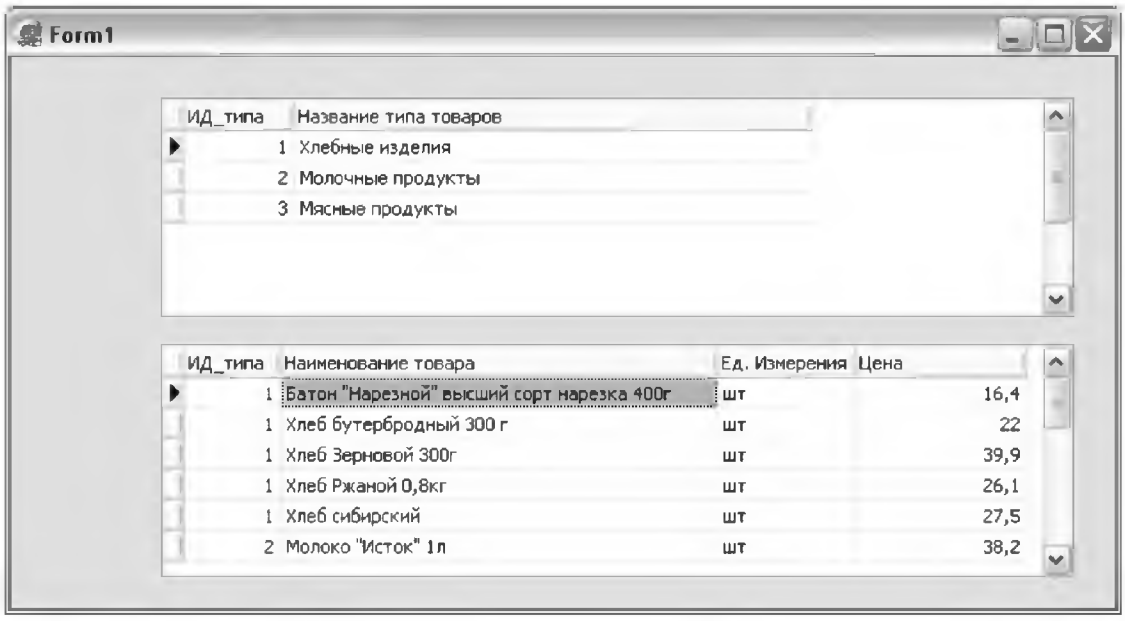


Рис. 1.25. Определение индексных полей при организации сортировки

В результате набор данных **ADOTable2**, ассоциированный с таблицей **GoodsCatalog**, будет отсортирован по типу товара, а внутри каждого типа товара – по названию товара.

1.3.4. Определение визуальных компонентов для работы с полями записи набора данных

Воспользуемся прототипом проекта из п. 1.3.3. Сделаем так, чтобы к полям записи в наборе данных **ADOTable2** можно было обращаться не только из компонента **DBGrid2**, но и из ряда визуальных компонентов, позволяющих осуществлять доступ к отдельным полям набора данных.

Добавим на форму три компонента **TDBEdit** со страницы палитры компонентов **Data Controls**. Разместим компонент **DBEdit1** под столбцом **Наименование товара**, компонент **DBEdit2** – под столбцом **Ед. измерения**, а компонент **DBEdit3** – под столбцом **Цена**. Определим поля, к которым можно обращаться через компонент **TDBEdit**. Установим следующие значения свойств:

– для **DBEdit1**: свойство **DataSource** – в значение **DataSource2**, свойство **DataField** – в значение **gcName**;

– для **DBEdit2**: свойство **DataSource** – в значение **DataSource2**, свойство **DataField** – в значение **gcMeasure**;

– для **DBEdit3**: свойство **DataSource** – в значение **DataSource2**, свойство **DataField** – в значение **gcCost**.

Для доступа к полю **IDtg** нам нужен более сложный компонент, который позволял бы вводить в поле **IDtg** таблицы **GoodsCatalog** только значения полей **IDtg** из таблицы **TypeGoods** и никаких других значений, при этом отображая не числовые значения, а названия типов товара. Для этой цели под столбцом **ИД_Тип** компонента **DBGrid2** разместим компонент **TDBLookupComboBox** с именем по умолчанию **DBLookupComboBox1**. Установим свойства этого компонента:

- **DataSource** – в значение **DataSource2**;
- **DataField** – в значение **IDtg**;
- **ListSource** – в значение **DataSource1**;
- **ListField** – в значение **tgName**;
- **KeyField** – в значение **IDtg**.

Добавим в приложение пять компонентов кнопок **TButton** со страницы **Standard** палитры компонентов. Изменим имена этих компонентов (свойство **Name**), используя инспектор объектов, на **InsertButton**, **EditButton**, **DeleteButton**, **PostButton** и **CancelButton** соответственно. Изменим заголовки этих кнопок (свойство **Caption**) на **Добавить**, **Изменить**, **Удалить**, **Запомнить** и **Отменить** соответственно.

Перейдем к определению обработчиков события нажатия на кнопки. За это действие отвечает обработчик **OnClick**. Для его определения необходимо выполнить двойное нажатие левой кнопки мыши на компоненте **TButton** или в области события **OnClick** в окне инспектора объектов (рис. 1.26). Шаблон процедуры обработчика генерируется автоматически.

Определим обработчик нажатия кнопки **InsertButton**, т. е. обработчик события **OnClick**:

```
procedure TForm1.InsertButtonClick(Sender: TObject);
begin
  if ADOTable2.State = dsBrowse then
    ADOTable2.Insert;
end;
```

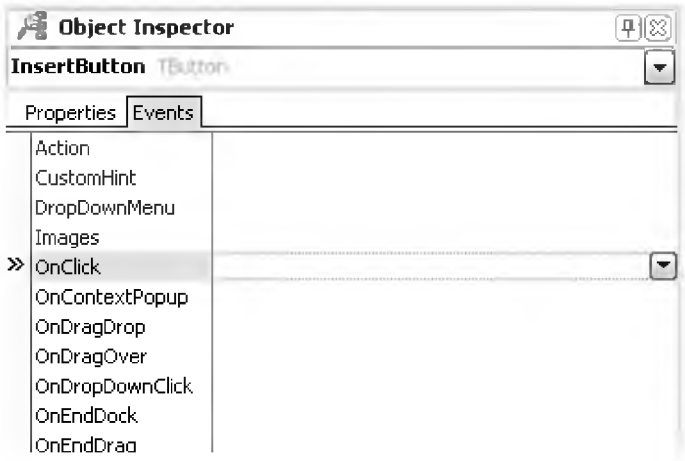


Рис. 1.26. Фрагмент инспектора объектов
с отображаемой областью событий

Метод **Insert** переводит набор данных **ADOTable2** в состояние добавления записи **dsInsert**. Ввод значений полей осуществляется в компонентах **DBLookupComboBox1**, **DBEdit1**, **DBEdit2**, **DBEdit3**. Для перехода в состояние **dsInsert** необходимо, чтобы набор данных находился в режиме просмотра **dsBrowse**.

Определим обработчик нажатия кнопки **EditButton**:

```
procedure TForm1.EditButtonClick(Sender: TObject);
begin
  if ADOTable2.State = dsBrowse then
    ADOTable2.Edit;
end;
```

Метод **Edit** переводит набор данных **ADOTable2** в состояние редактирования записи **dsEdit**. Редактирование значений полей осуществляется в компонентах **DBLookupComboBox1**, **DBEdit1**, **DBEdit2**, **DBEdit3**. При этом необходимо, чтобы набор данных находился в режиме просмотра **dsBrowse**.

Определим обработчик нажатия кнопки **DeleteButton**:

```
procedure TForm1.DeleteButtonClick(Sender: TObject);
begin
  if ADOTable2.State = dsBrowse then
    if MessageDlg('Подтвердите удаление записи',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      ADOTable2.Delete;
end;
```

Если набор данных **ADOTable2** находится в режиме просмотра **dsBrowse**, то при выполнении функции **MesssageDlg** будет вызвано окно диалога. При нажатии кнопки **Yes** произойдет удаление текущей записи в наборе данных **ADOTable2**.

Определим обработчик нажатия кнопки **PostButton**:

```
procedure TForm1.PostButtonClick(Sender: TObject);
begin
  if ADOTable2.State in [dsInsert, dsEdit] then
    ADOTable2.Post;
end;
```

Если набор данных находится в режиме добавления новой записи или редактирования, то происходит выполнение метода **Post**, который запоминает текущее состояние записи в таблице. После запоминания набор данных переводится в режим просмотра **dsBrowse**.

Определим обработчик нажатия кнопки **CancelButton**:

```
procedure TForm1.CancelButtonClick(Sender: TObject);
begin
  if ADOTable2.State in [dsInsert, dsEdit] then
    ADOTable2.Cancel;
end;
```

В этом случае нахождение набора данных в режиме добавления новой записи или в режиме редактирования приводит к выполнению метода **Cancel**. Этот метод отменяет запоминание записи в таблице и переводит набор данных в режим просмотра **dsBrowse**.

Для запрета перевода набора данных в состояние добавления и изменения данных, а также для удаления записей непосредственно из компонента **DBGrid2**, установим для него свойство **ReadOnly** в значение **True**.

При просмотре в компоненте **DBGrid2** по умолчанию выделяется ячейка, чтобы ее можно было редактировать, поэтому при отключении режима редактирования было бы удобно, чтобы выделенная строка подсвечивалась целиком. Для достижения этого визуального эффекта в инспекторе объектов в списке параметров **Options** для **DBGrid2** необходимо установить свойство **dgRowSelect** в значение **true**.

Чтобы при случайном вводе данных в компоненты **TDBEdit** или изменении значения **DBLookupComboBox1** не происходило изменения записей в наборе данных, в компоненте **DataSource2** необходимо изменить свойство **AutoEdit**, назначив ему значение **false**.

После этого запустим приложение на выполнение (рис. 1.27).

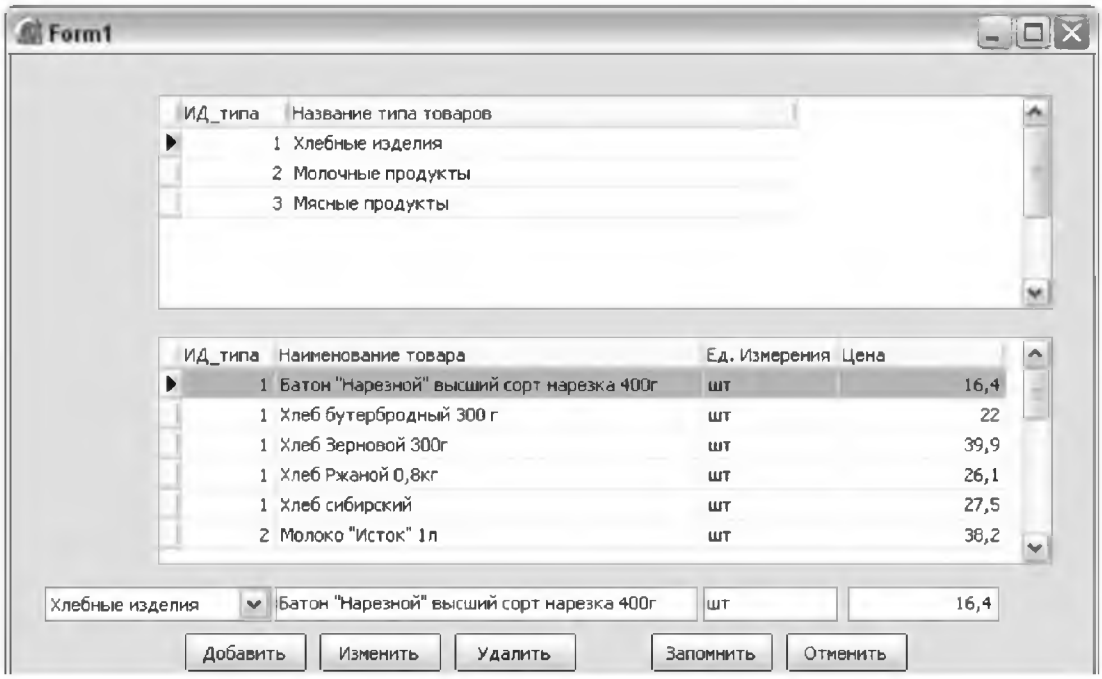


Рис. 1.27. Использование визуальных компонентов для работы с данными

При добавлении новой записи или при корректировке существующей записи в поля можно заносить значения, используя данные из компонентов **DBEdit1**, **DBEdit2**, **DBEdit3**, вводимые пользователем с клавиатуры, и выбирая тип товара из списка значений в компоненте **DBLookupComboBox1**. То же самое будет происходить и при изменении записи. При удалении записи появляется окно диалога для подтверждения удаления.

1.3.5. Использование компонента TADOQuery для формирования набора данных из нескольких таблиц

Для организации работы с данными в среде БД предпочтительным способом является использование языка **SQL**, который поддерживается всеми современными реляционными СУБД. Воспользуемся **SQL**-запросом, для чего разместим на форме компонент **TADOQuery**. В инспекторе объектов при помощи выпадающего списка свойству **Connection** этого компонента зададим значение **ADOConnection1.** Далее разместим на форме компонент **TDataSource** с именем **DataSource3** и установим его свойство **DataSet** в значение **ADOQuery1**. После этого расположим на форме компонент **TDBGrid** и установим его свойство **DataSource** в значение **DataSource3**.

В области инспектора объектов найдем свойство **SQL** у компонента **ADOQuery1** и нажмем на кнопку с многоточием. В появившемся окне редактора наберем текст **SQL**-запроса:

```
SELECT GC.IDgc, GC.gcName, TG.tgName, GC.gcMeasure,
GC.gcCost

FROM GoodsCatalog GC, TypeGoods TG

WHERE GC.IDtg = TG.IDtg

ORDER BY TG.tgName, GC.gcName;
```

и нажмем кнопку **OK**.

После этого установим свойство **ADOQuery1.Active** в значение **True** и увидим, как вместо числового значения кода типа товара будет отображено название типа товара. Настроим вид отображения в **DBGrid3** при помощи редактора столбцов (рис. 1.28).

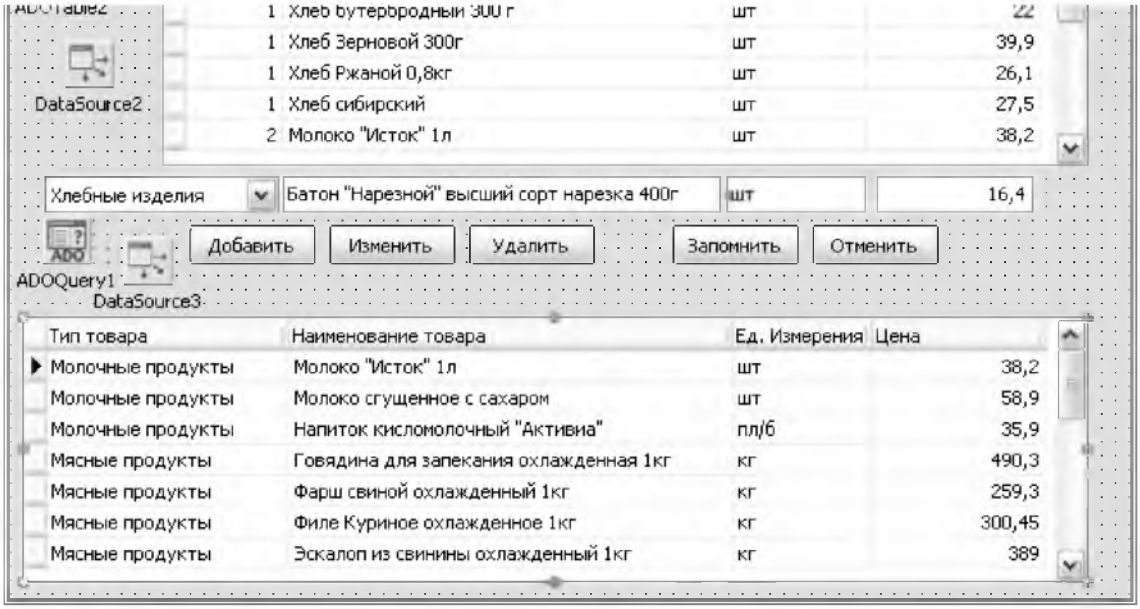


Рис. 1.28. Пример отображение записей активного компонента **ADOQuery1**

Набор данных **ADOQuery1** содержит сведения о приходе товара. В составе записи этого товара присутствуют поля **IDgc** (**Код товара**), **gcName** (**Наименование товара**), **tgName** (**Наименование типа товара**), **gcMeasure** (**Единица измерения**), **gcCost** (**Цена за ед. измерения**). Таким образом, набор данных собирается из двух таблиц базы данных: **GoodsCatalog** и **TypeGoods**. При этом записи из этих таблиц соединяются по одинаковым значениям поля **IDtg**.

Существенным недостатком использования компонента **TADOQuery** является то, что при выполнении приложения нельзя добавить новые записи и изменить или удалить существующие записи в наборе данных. Это происходит потому, что тип набора данных, собираемого более чем из одной таблицы БД, является доступным только для чтения.

В этом случае для того, чтобы пользователь видел удобное для него отображение данных, необходимо взять компонент **TADOQuery**, а для редактирования данных – компонент **TADOTable**. Но каким же образом синхронизировать то, что видит пользователь, с тем, что он редактирует? Для этого после изменения позиции записи в **ADOQuery1** следует выполнить синхронизацию данных в **ADOTable2** с выбранной записью. В нашем примере мы будем использовать поле **IDgc**, которое является идентификатором товара. Ранее мы удалили это поле из списков полей **ADOTable2** и нам нужно его вернуть. Для этого вызовем редактор полей **Field Editor**, в контекстном меню которого выберем пункт **Add Fields…**, а затем поле **IDgc**, после чего нажмем кнопку **OK**. Теперь компонент **ADOTable2** подготовлен для синхронизации.

Теперь перейдем к настройке набора данных **ADOQuery1**. Для этого создадим поля **TField** в редакторе полей, выбрав в его контекстном меню пункт **Add all fields**.

В случае когда набор данных находится в состоянии **dsBrowse**, при переходе к новой записи возникают события:

```
property BeforeScroll: TDataSetNotifyEvent;
```

которое наступает перед переходом на другую запись в наборе данных, и

```
property AfterScroll: TDataSetNotifyEvent;
```

которое происходит после перехода на другую запись в наборе данных.

В нашем случае необходимо, чтобы после перемещения по записям **ADOQuery1** в таблице **ADOTable2** выбиралась соответствующая запись. Для этого настроим событие **AfterScroll**:

```
procedure TForm1.ADOQuery1AfterScroll(DataSet: TDataSet);
begin
  if ADOQuery1.RecordCount > 0 then
    ADOTable2.Locate('IDgc', ADOQuery1IDgc.Value, [ ])
end;
```

При срабатывании события **AfterScroll** вначале проверяется, имеются ли отображаемые записи (количество записей больше нуля), после чего в **ADOTable2** будет происходить поиск записи по значению **ADOQuery1IDgc**.

Компонент **TADOQuery**, как и компонент **TQuery**, имеет следующую особенность: после выполнения запроса на чтение к базе данных происходит отображение записи НД в неизменном виде независимо от того, изменялось ли после запроса содержимое таблицы БД. После обновления информации в базе данных приходится закрывать и снова открывать компонент **TQuery** или **TADOQuery**, т. е. повторно выполнять запрос к удаленной БД. В связи с этим нам необходимо добавить обработчики событий **AfterPost** и **AfterDelete** компонента **ADOTable2**:

```
procedure TForm1.ADOTable2AfterPost(DataSet: TDataSet);
begin
  ADOQuery1.Active := False;
  ADOQuery1.Active := True;
end;
```

```
procedure TForm1.ADOTable2AfterDelete(DataSet: TDataSet);
begin
  ADOQuery1.Active := False;
  ADOQuery1.Active := True;
end;
```

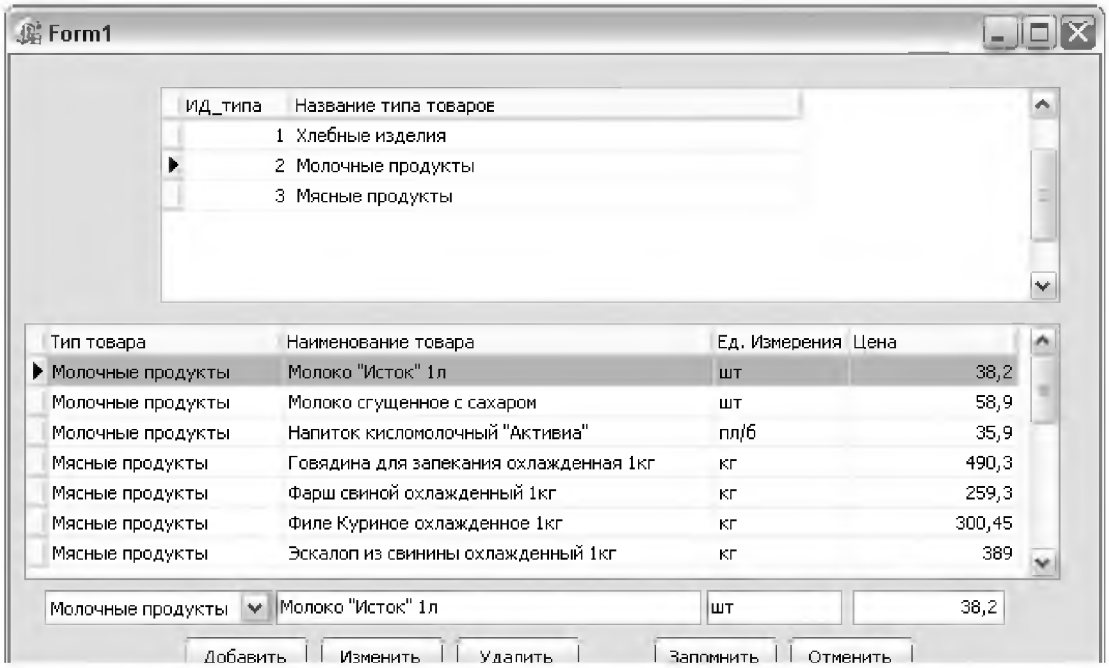


Рис. 1.29. Синхронизация компонентов **TADOQuery** и **TADOTable** по полю **IDgc**

Теперь можно сказать, что мы имеем полноценный блок по настройке списка товаров, в котором возможна перестройка интерфейса приложения путем удаления компонента **DBGrid2** и перемещения компонента **DBGrid3** выше элементов управления редактирования записей (рис. 1.29).

1.3.6. Реализация связи Master–Detail между наборами данных

Нам уже известно, что таблицы **TypeGoods** и **GoodsCatalog** находятся в отношении «один-ко-многим». А поскольку мы определили ссылочную целостность между ними, то можно сделать так, чтобы при установке указателя на запись в наборе данных **ADOTable1**, ассоциированном с таблицей **TypeGoods**, и в наборе данных **ADOTable2**, ассоциированном с таблицей **GoodsCatalog**, отображались бы только те записи товара, которые соответствуют выбранному типу

товара в НД **ADOTable1**. Это может быть реализовано через механизм связи наборов данных **Master**–**Detail**.

Воспользуемся копией проекта из п. 1.3.2. В окне инспектора объектов для компонента **ADOTable2** установим свойство **IndexFieldNames** в значение **IDtg**, а свойство **MasterSource** – в значение **DataSource1**. Далее перейдем на свойство **MasterFields** и нажмем кнопку с многоточием. В появившемся окне **Field Link Designer** установим параметры связи. В списке **Detail Fields** выберем поле **IDtg**, в списке **Master Fields** – поле **IDtg** и нажмем кнопку **Add**. В поле **Joined Fields** будет сформировано выражение **IDtg –> IDtg** (рис. 1.30). В завершение нажмем кнопку **OK**.

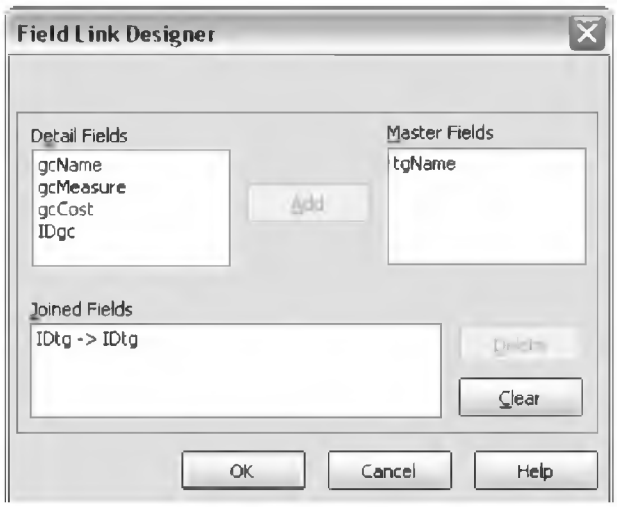


Рис. 1.30. Окно настройки соединения полей **Field Link Designer**

Запустим разработанное нами приложение. Теперь в наборе данных **ADOTable2** показываются только записи соответствующего типа товара, который выбран из набора данных **ADOTable1** (рис. 1.31). Более того, при добавлении записи в набор данных **ADOTable2** значение поля **IDtg** по умолчанию берется равным значению поля **IDtg** из текущей записи в наборе данных **ADOTable1**. Это позволяет обойтись без запоминания идентификатора соответствующего типа товара, что значительно упрощает работу пользователя.

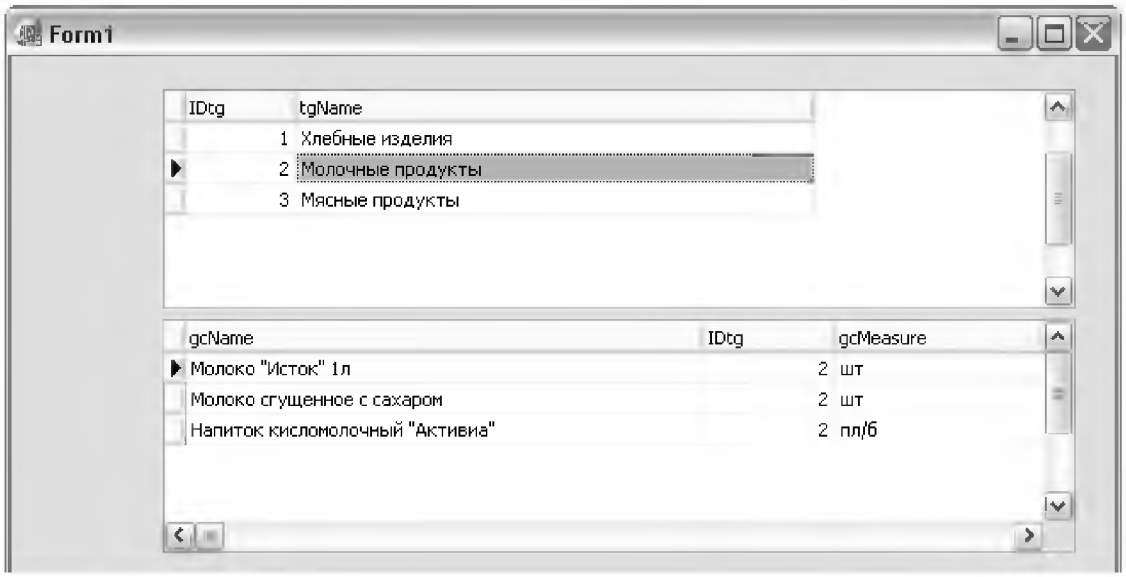


Рис. 1.31. Организация связи **Master–Detail**
при помощи компонента **TADOTable**

Реализуем механизм организация связи **Master–Detail** для компонентов **ADOTable1** и **TADOQuery**. Для этого разместим на форму и настроим компонент **TADOQuery** с именем **ADOQuery1**. Зададим его свойству **Connection** значение **ADOConnection1** и запишем код **SQL**-запроса:

```
SELECT GC.IDgc, GC.gcName, TG.tgName, GC.gcMeasure, GC.gcCost
FROM GoodsCatalog GC, TypeGoods  TG
WHERE GC.IDtg = TG.IDtg AND GC.IDtg =:IDtg
ORDER BY TG.tgName, GC.gcName;
```

После этого свойству **ADOQuery1 DataSource** установим значение **DataSource1**, а свойству **Active** – значение **True**.

Как можно заметить, в нашем в **SQL**-запросе в блоке **WHERE** появилось новое условие, которое дополнительно выполняет еще одну проверку **GC.IDtg =:IDtg**, где **:IDtg** означает параметр, значение которого берется из набора данных **ADOTable1** через компонент **DataSource1**.

Настроив компоненты **DataSource3** и **DBGrid3** для организации визуального отображения НД компонента **ADOQuery1** и запустив программу, мы увидим работающий аналог связки **Master–Detail** для компонентов **TADOTable** (рис. 1.32).

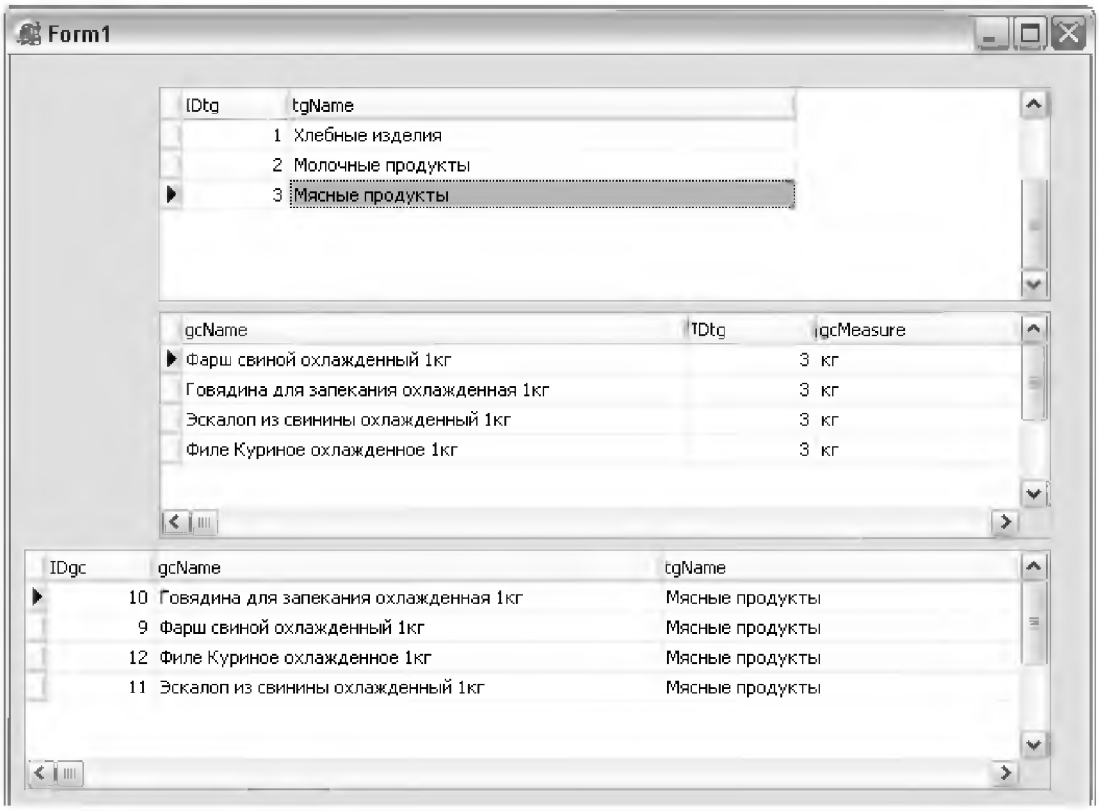


Рис. 1.32. Организация связи **Master–Detail**
при помощи компонента **TADOQuery**

Используя возможности реализации связи **Master–Detail**, мы можем создавать приложения, учитывающие отображения взаимосвязанных данных как на основе запросов, так и при непосредственной работе с таблицами.

1.4. Базовые принципы использования конструктора запросов Microsoft Access

Высокоуровневым средством формирования запросов в СУБД **Microsoft Access** является конструктор запросов, который можно рассматривать как своеобразную оболочку к языку запросов **SQL**. Для формирования запроса в конструкторе используется табличная форма. Рассмотрим, как с помощью конструктора реализовать аналог **SQL**-запроса, использованного в п. 1.3.5.

Для вызова конструктора запросов необходимо перейти на вкладку **Создание** и в группе **Другие** нажать пиктограмму **Конструктор запросов** (рис. 1.33).

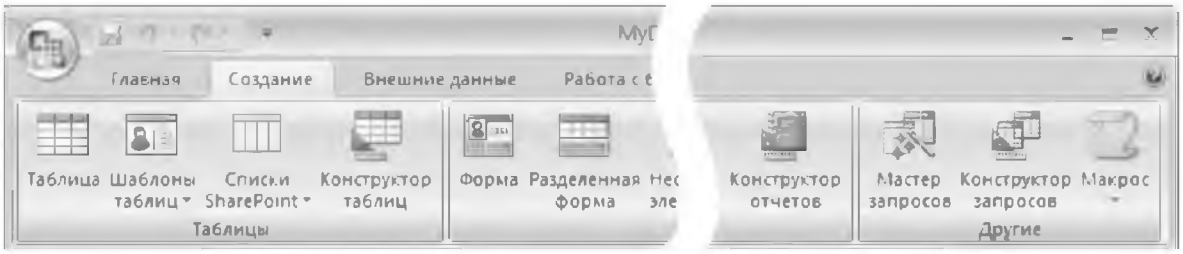


Рис. 1.33. Вкладка **Создание** с группой **Другие**

Сразу после нажатия на кнопку появится область конструктора запросов и диалоговое окно **Добавление таблицы**. На данном этапе работы нам понадобятся две таблицы: **GoodsCatalog** и **TypeGoods**, которые мы туда и добавим. В результате будут отображены таблицы с установленной связью (рис. 1.34).

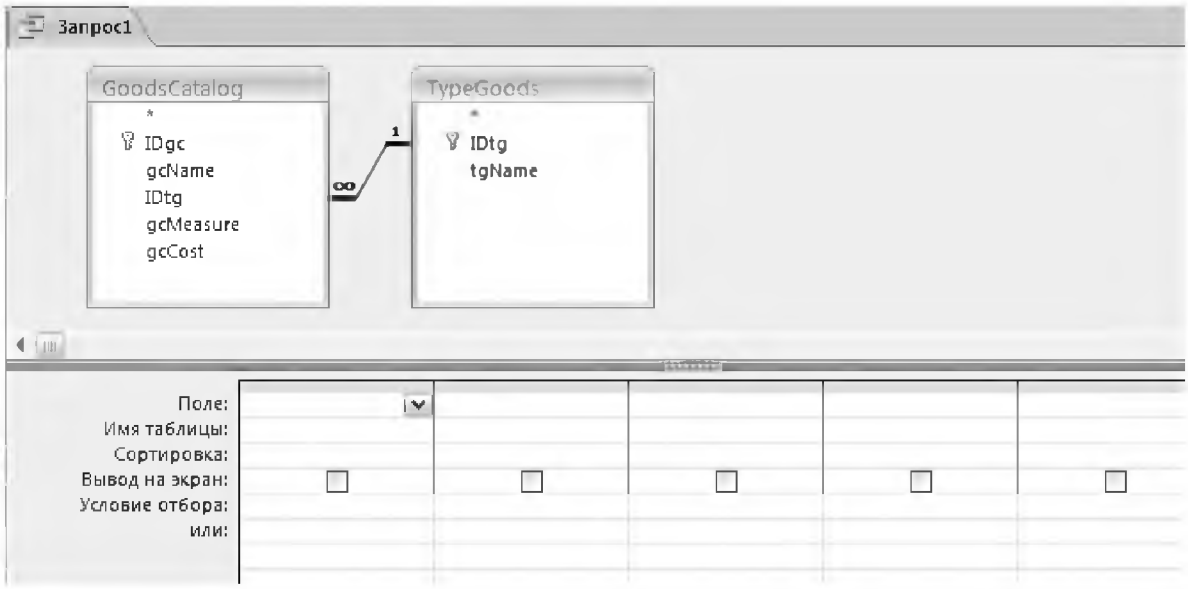


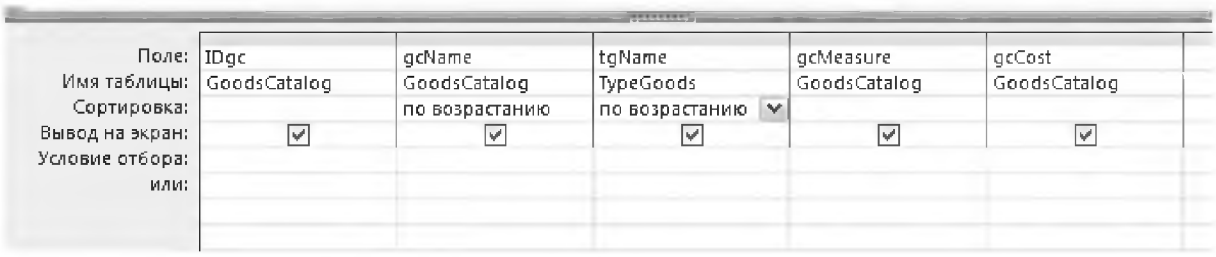
Рис. 1.34. Окно конструктора запроса после добавления таблиц

В первой строке таблицы, расположенной в нижней части рис. 1.34, указываются имена полей, участвующих в формировании запроса. Вторая строка содержит имена таблиц, из которых извлекаются нужные поля. В третьей строке находятся признаки сортировки. Флажки в четвертой строке являются признаками вывода значений полей на экран при выполнении запроса. В следующих строках формируется условие отбора.

Для того чтобы заполнить эту таблицу, можно воспользоваться одним из двух способов: первый способ заключается в выборе нужного поля из выпадающего списка в первой строке таблицы, второй

способ – в переносе поля мышью, для чего необходимо, выбрав нужное поле и удерживая левую кнопку мыши, перетащить его в таблицу.

Настроим поля таблицы конструктора, как показано на рис. 1.35.



Поле:	IDgc	gcName	tgName	gcMeasure	gcCost
Имя таблицы:	GoodsCatalog	GoodsCatalog	TypeGoods	GoodsCatalog	GoodsCatalog
Сортировка:		по возрастанию	по возрастанию		
Вывод на экран:	☑	☑	☑	☑	☑
Условие отбора:					
или:					

Рис. 1.35. Настройка параметров в конструкторе запросов

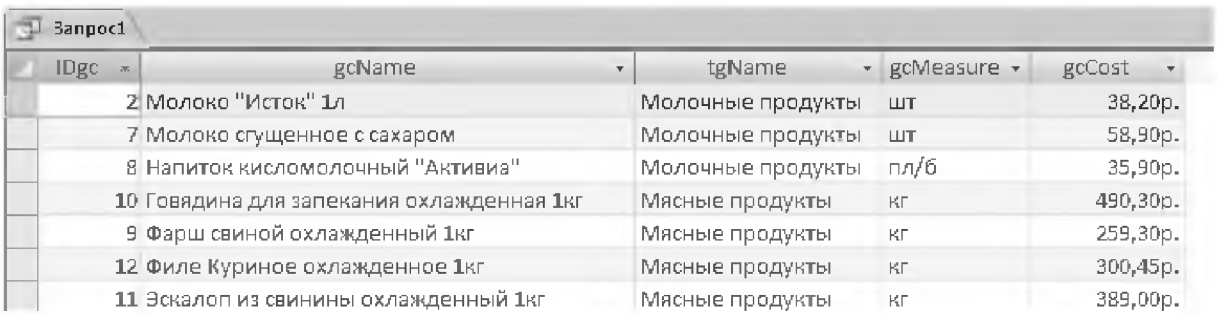
Переключившись в режим редактирования **SQL**-кода нажатием в правом нижнем углу кнопки **Режим SQL**, мы увидим следующий **SQL**-запрос:

```
SELECT GoodsCatalog.IDgc, GoodsCatalog.gcName,
  TypeGoods.tgName, GoodsCatalog.gcMeasure,
  GoodsCatalog.gcCost
FROM TypeGoods INNER JOIN GoodsCatalog ON
  TypeGoods.IDtg = GoodsCatalog.IDtg
ORDER BY GoodsCatalog.gcName, TypeGoods.tgName;
```

В этом запросе реализована та же функциональная возможность, что и в запросе из п. 1.3.5, однако здесь ведется сортировка вначале по полю **gcName**, а затем по полю **tgName**. Исправим строчку **ORDER BY** следующим образом:

```
ORDER BY TypeGoods.tgName, GoodsCatalog.gcName;
```

Выполнив данный запрос, мы увидим следующий результат (рис. 1.36).



Запрос1

IDgc	gcName	tgName	gcMeasure	gcCost
2	Молоко "Исток" 1л	Молочные продукты	шт	38,20р.
7	Молоко сгущенное с сахаром	Молочные продукты	шт	58,90р.
8	Напиток кисломолочный "Активиа"	Молочные продукты	пл/б	35,90р.
10	Говядина для запекания охлажденная 1кг	Мясные продукты	кг	490,30р.
9	Фарш свиной охлажденный 1кг	Мясные продукты	кг	259,30р.
12	Филе Куриное охлажденное 1кг	Мясные продукты	кг	300,45р.
11	Эскалоп из свинины охлажденный 1кг	Мясные продукты	кг	389,00р.

Рис. 1.36. Фрагмент выполненного запроса

Таким образом, СУБД **Microsoft Access** позволяет сформировать запросы с минимальными знаниями языка **SQL**. Созданный в конструкторе запрос можно вставить в компонент **ADOQuery1** и он будет работать точно так же, как и написанный собственноручно.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Создать базу данных в соответствии с п. 1.2.
2. Создать набор таблиц в соответствии с п. 1.2.1.
3. Выполнить настройку индексов для таблиц **GoodsCatalog** (**GoodsName**, **GoodsTypeAndName**) и **IncomingGoods** (**IncomingDateGoods**) (см. п. 1.2.2).
4. Организовать связь между таблицами базы данных по ключевым полям (см. п. 1.2.3).
5. Заполнить таблицы БД данными с использованием режима подстановки (см. п. 1.2.4).
6. Создать простейшее приложение с двумя таблицами в среде **Delphi** (см. пп. 1.3.1–1.3.2).
7. Настроить параметры отображения столбцов в компоненте **TDBGrid** и выполнить смену активного индекса (см. п. 1.3.3).
8. Организовать работу с полями таблиц БД с использованием визуальных компонентов **TDBGrid**, **TDBEdit**, **TDBLookupComboBox** и **TButton** (см. п. 1.3.4).
9. Использовать компонент **TADOQuery** для формирования набора данных из нескольких таблиц с последующей синхронизацией компонентов **TADOQuery** и **TADOTable** (см. п. 1.3.5).
10. Установить связь **Master–Detail** между наборами данных с использованием компонентов **TADOQuery** и **TADOTable** (см. п. 1.3.6).
11. Сгенерировать **SQL**-запрос с использованием конструктора запросов **Microsoft Access** (см. п. 1.4).
12. Ответить на контрольные вопросы.
13. Составить отчет, который должен содержать титульный лист, цель и ход выполнения лабораторной работы, сформулированные выводы.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. В каком виде хранятся базы данных, выполненные в известных вам СУБД?
2. Приведите структурные схемы однопользовательской архитектуры при работе с локальными базами данных и архитектуры «файл–сервер».
3. Представьте структурную схему архитектуры «клиент–сервер».
4. Дайте расшифровку аббревиатуры **ADO**, **ODBC**.
5. Какие расширения имеют файлы базы данных в **Microsoft Access**?
6. Какие типы данных используются в СУБД **Microsoft Access**?
7. Объясните, для чего применяются ключевые поля.
8. Охарактеризуйте тип поля **Счетчик**.
9. Для каких целей используются индексы?
10. Какие индексы могут быть созданы в **Microsoft Access**?
11. Какими свойствами обладают индексы в **Microsoft Access**?
12. Опишите создание составных индексов в **Microsoft Access**.
13. Приведите способ задания связей между таблицами БД.
14. Охарактеризуйте работу каскадного обновления связанных полей БД.
15. Каким образом выполняется каскадное удаление связных записей?
16. Для чего в **Microsoft Access** используется подстановка?
17. Опишите процесс настройки подстановки данных для полей в **Microsoft Access**.
18. Какие функции выполняет палитра компонентов в среде **RAD Sudio**?
19. Для каких целей необходим инспектор объектов?
20. Какие компоненты применяются для создания приложений БД?
21. Опишите процесс подключения к БД с помощью компонента **TADOConnection**.
22. Дайте определение понятия «набор данных».
23. В чем состоит отличие компонента **TADOTable** от компонента **TADOQuery**?
24. Какими способами можно сформировать список полей таблицы, отображаемых в приложении?
25. Опишите работу с компонентом **TADOQuery**.

26. Какие функции выполняет редактор полей?

27. Охарактеризуйте свойство **IndexFieldNames**.

28. С какой целью применяется компонент **TDBLookupComboBox**?

29. В каких состояниях может находиться набор данных?

30. Каким образом можно запретить непосредственное редактирование записей в компоненте **TDBGrid**?

31. Каким образом в компоненте **TDBGrid** выполняется настройка отображаемых полей?

32. Приведите текст обработчика для удаления записи.

33. Приведите текст обработчика для отмены запоминания записи.

34. Составьте текст обработчика для сохранения записи.

35. Какие события возникают при смене записи в **TDBGrid**?

36. Объясните необходимость реализации связи **Master–Detail**.

37. Опишите порядок реализации связи **Master–Detail** между наборами данных для компонентов **TADOTable**.

38. В чем заключается особенность реализации связи **Master–Detail** для компонентов НД с использованием компонента **TADOQuery**?

39. Какие функции выполняет конструктор запросов **Microsoft Access**?

40. Каким образом задается порядок следования и сортировки полей в компоненте **TADOQuery**?

Лабораторная работа 2

РАЗРАБОТКА ИНТЕРФЕЙСА НА ОСНОВЕ НЕСКОЛЬКИХ ФОРМ И РАБОТА С ПОЛЯМИ

Цель работы: ознакомиться с принципами построения интерфейса на основе использования нескольких форм; ознакомиться со способами обращения к полям таблиц и их значениям; ознакомиться с форматированием полей и проверкой введенного в поле значения; разработать приложение – пример многооконного интерфейса с обработкой событий **OnGetText**, **OnSetText**, **OnValidate**, **OnChange**; разработать приложение, иллюстрирующее создание полей выбора данных и вычисляемых полей.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В большинстве случаев при разработке программ, работающих с базой данных, в которых имеется более трех таблиц, для удобства пользователя используют интерфейс на основе нескольких форм. При этом возможна различная организация обращения к полям таблиц и их значений, необходимых для реализации бизнес-правил*.

Среда разработки **RAD Studio** предлагает размещать компоненты доступа к данным на специальной форме (модуле) – контейнере **TDataModule**. Следует отметить, что контейнер **TDataModule** не имеет ничего общего с классической формой приложения, поскольку на нем можно размещать только невизуальные компоненты (**TADOConnection**, **TADOTable**, **TADOQuery**, **TDataSource** и т. д.). Тем не менее он доступен разработчику, как и любой другой модуль проекта, на этапе разработки и в то же время не является видимым для пользователя. Основным преимуществом размещения компонентов доступа к данным в контейнере **TDataModule** является то, что изменение значения любого свойства проявится сразу же во всех обычных модулях, к которым подключен этот контейнер.

Контейнер **TDataModule** хранит только невизуальные компоненты, а весь интерфейс организован в виде отдельных форм, поэтому перейдем к рассмотрению организации взаимодействия форм с НД, расположенных на этом контейнере.

---

* Более подробно об этом см. в кн.: Культнн Н. Программирование в Delphi 2010. Самоучитель. СПб. : БХВ-Петербург, 2010.

2.1. Разработка интерфейса с учетом нескольких форм (окон)

При разработке интерфейса с учетом нескольких форм (окон) необходимо принимать во внимание различные аспекты, связанные с обращениями к базе данных. Для работы с базой данных, спроектированной в лабораторной работе 1, создадим новый проект и сохраним его в соответствующем каталоге, например в каталоге **C:\User\Group\Surname\Lab2**.

2.1.1. Создание контейнера TDataModule

Для добавления в проект модуля данных зайдем в главное меню **File → New → Other**. В появившемся окне **New Items** (рис. 2.1) в группе **Delphi Projects → Delphi Files** выберем значок **Data Module**.

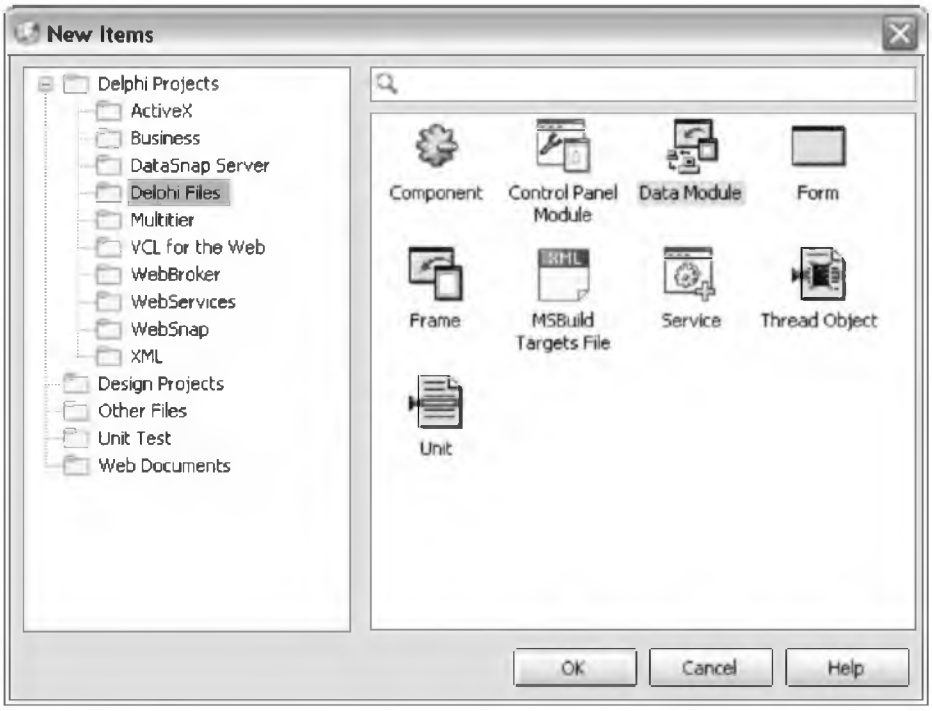


Рис. 2.1. Окно создания нового элемента проекта

Контейнер **TDataModule**, добавленный в приложение, автоматически приобретает название с цифрой следующего модуля (в нашем случае это **DataModule2**), который при необходимости можно переименовать. Невизуальная форма **TDataModule** похожа на обычную форму **TForm**, но в нее можно помещать только невизуальные компоненты для работы с базами данных. При выборе в качестве активного

окна **TDataModule** палитра инструментов будет отображать только те компоненты, которые можно на нем разместить. Связь компонентов **TADOConnection**, **TADOTable**, **TADOQuery** и **TDataSource**, расположенных в контейнере **TDataModule**, производится стандартным образом.

Разместим в области формы **DataModule2** следующие компоненты: **TADOConnection**; четыре экземпляра **TADOTable** (для вывода и редактирования таблиц базы данных); два экземпляра **TADOQuery** (для отображения результатов многотабличных запросов); шесть экземпляров **TDataSource**.

Если в приложении имеется большое количество наборов данных и используются стандартные имена, то в них очень легко запутаться. Поэтому введем следующие обозначения для задания имени компонентов: имя компонента **TADOTable** зададим как **T_ИмяТаблицы**, имя компонента **TADOQuery** – как **Q_ИмяЗапроса**; для компонента **TDataSource** имя будет формироваться на основе имени компонента, предоставляющего набор данных **DS_ИмяКомпонентаНД**. Так, например, имена у компонентов, обеспечивающих отображения таблицы **GoodsCatalog**, будут следующими: у компонента **TADOTable** – **T_GoodsCatalog**, у компонента **TDataSource** – **DS_ T_GoodsCatalog**.

Для компонентов **TADOQuery** необходимо задать **SQL**-запросы. В первом компоненте **TADOQuery**, который следует переименовать в **Q_GoodsCatalogD**, поскольку он отображает сведения о товарах, поместим **SQL**-запрос, представленный в п. 1.3.5 лабораторной работы 1. Для отображения сведений о приходе товаров во второй компонент **TADOQuery** с именем **Q_IncomingGoodsD** поместим **SQL**-запрос, который создадим при помощи конструктора запросов **Microsoft Access**. В качестве отображаемых полей выберем поля **IDig**, **IDgc**, **gcName**, **igDate**, **igAmount**, **scName** из соответствующих таблиц, при этом у нас должен получиться следующий **SQL**-запрос:

```
SELECT IncomingGoods.IDig, IncomingGoods.igDate,
  GoodsCatalog.IDgc, GoodsCatalog.gcName,
  IncomingGoods.igAmount, SuppliersCatalog.scName
FROM SuppliersCatalog INNER JOIN (GoodsCatalog INNER JOIN
  IncomingGoods ON GoodsCatalog.IDgc = IncomingGoods.IDgc) ON
  SuppliersCatalog.IDsc = IncomingGoods.IDsc;
```

Результат этого **SQL**-запроса будет отображаться на основной форме.

2.1.2. Организация взаимодействия форм с набором данных в контейнере TDataModule

При создании приложения будем использовать три формы: на первой форме будет отображаться информация по приходу товара, на второй – сведения о поставщиках, на третьей – информация о товаре и типах товара. Поскольку первая форма была сформирована в момент создания проекта, то необходимо добавить еще две формы. Для этого в меню **RAD Studio** выберем **Files → New → Form – Delphi**.

Разместим на первой форме компонент **TDBGrid**, который свяжем с компонентом **DS_Q_IncomingGoodsD**. Пытаясь выбрать его из списка свойства **DataSource** компонента **TDBGrid**, мы увидим, что в этом списке ничего нет. Это связано тем, что все невизуальные компоненты находятся в контейнере **DataModule2**. Для организации взаимосвязи между формами и/или модулями выберем пункт меню **File → Use Unit…**, после чего появится окно связи модулей проекта (рис. 2.2). В связи с тем что первая форма будет вызывать все остальные, выберем все модули и нажмем кнопку **OK**.

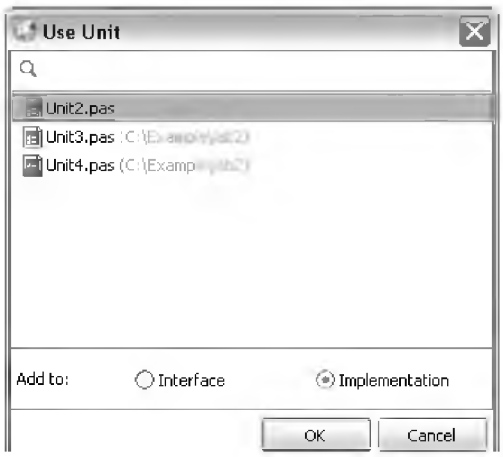


Рис. 2.2. Окно создания связи
с модулями проекта

Теперь мы можем выбрать значение **DataModule2.DS_Q_ IncomingGoodsD** в свойстве **DataSource** компонента **DBGrid1**.

2.1.3. Создание главного меню и вызов форм

Чтобы добавить к программе главное меню, нужно разместить на форме в произвольном месте невизуальный компонент **TMainMenu**. Опции главного меню создаются с помощью специального редактора. Редактор меню вызывается с помощью двойного

щелчка по компоненту **TMainMenu** или при помощи пункта контекстного меню **Menu Designer**. Первоначально меню пустое, но имеет один выделенный элемент (рис. 2.3).

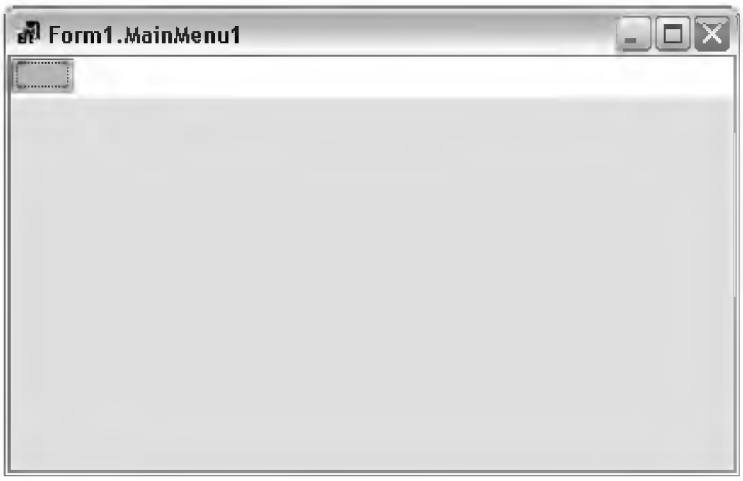


Рис. 2.3. Окно дизайнера меню

Для создания первой опции (как правило, это опция главного меню **Файл**) нужно перейти в инспектор объектов и свойству **Caption** присвоить нужное название.

Обратите внимание на то, что среда **Delphi** автоматически создает следующий пустой пункт меню верхнего уровня. При выборе одного из пунктов этого меню (например, **Файл**) мы получим пустой пункт меню второго уровня. Выделим нижний элемент меню, в инспекторе объектов изменим свойство **Caption** на **Выход**. Затем перейдем на вкладку инспектора объектов – **Events**, щелкнем дважды мышью по обработчику **OnClick** и запишем следующий текст в обработчике:

```
Close;
```

или

```
Application.Terminate;
```

В контекстном меню выделенного элемента имеется пункт **Create SubMenu**, нажав на который, мы создадим подменю выбранного элемента, а к его названию прибавится изображение треугольника – стрелки, указывающей на его наличие. Работа с подменю осуществляется так же, как и с элементами основного меню.

Создадим в главной строке меню два пункта **Поставщики** и **Товары**. Для вызова формы, содержащей информацию о поставщиках (пусть это будет **Form3**), в обработчик **OnClick** пункта

Поставщики необходимо вписать код одного из двух способов показа нужной формы:

– в случае обычной формы:

```
Form3.Show;
```

– в случае модальной формы:

```
Form3.ShowModal;
```

Разница между этими формами заключается в том, что обычная форма разрешает свободный переход между всеми формами, находящимися в данный момент на экране, а модальная форма в момент вызова блокирует переход между формами проекта до тех пор, пока не будет закрыта, и работа возможна только в ней.

Зададим связь между формой **Form3**, отвечающей за поставщиков, и модулем с контейнером **TDataModule** (**Unit2**). После этого разместим на форме компонент **TDBGrid** и свяжем его с набором данных, содержащим таблицу **Поставщики** (**SuppliersCatalog**).

Далее настроим форму **Form4**, которая будет отвечать за отображение таблиц товаров и типов товаров. Поскольку нам необходимо выводить и редактировать две таблицы, то для создания вкладок мы можем воспользоваться компонентом **TPageControl**, в контекстном меню которого нам необходимо выбрать пункт **New Page**. Каждой вкладке **PageControl1** зададим понятное будущему пользователю имя в свойстве **Caption** (рис. 2.4).

Рис. 2.4. Использование компонента **PageControl** для создания вкладки

В заключение расставим визуальные компоненты и настроим работу НД, как мы делали это в лабораторной работе 1. Полученый результат показан на рис. 2.5.

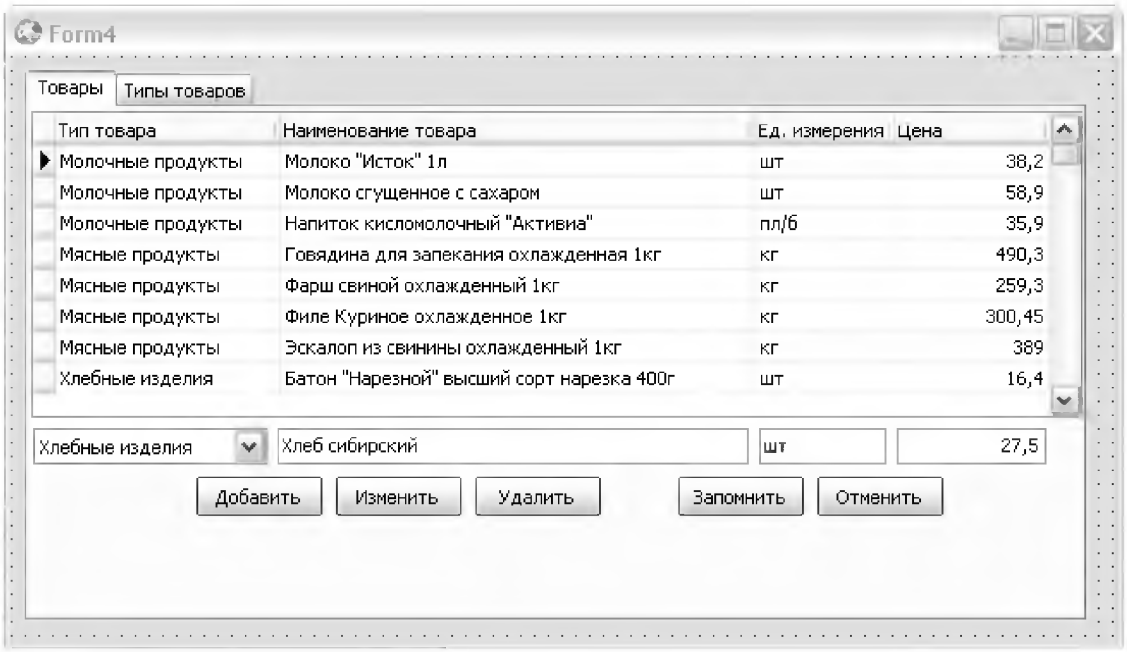


Рис. 2.5. Размещение компонентов на вкладке **PageControl**

Теперь запишем обработчики событий. Поскольку невизуальные компоненты размещены в контейнере **DataModule2**, то при написании кода обращения к ним с данной формы нужно указывать место его размещения.

Например, обработчик события **OnClick** кнопки сохранения для нашего случая будет выглядеть следующим образом:

```
procedure TForm4.PostButtonClick(Sender: TObject);
begin
  if DataModule2.T_GoodsCatalog.State in [dsInsert, dsEdit] then
    DataModule2.T_ GoodsCatalog.Post;
end;
```

Необходимо обратить внимание на то, что при написании этого кода, возможно, будет сообщено об ошибках, потому что компилятор не знает констант **dsInsert** и **dsEdit**. Для того чтобы компилятор обрабатывал эти константы правильно, необходимо подключить соответствующие модули в блоке **USES** вверху исходного кода текущего модуля (в нашем случае нужно добавить **DB**).

Примечание. Если вы не знаете, какой модуль необходим, то достаточно поставить на форму компонент, к которому вы обращаетесь в данный момент на другой форме (например, **TADOTable**), и сохранить проект. После этого можно удалить компонент и еще раз сохранить проект. Таким образом, для данной формы будут подключены все необходимые модули.

2.2. Обращение к полям и их значениям

Довольно часто разработчикам приходится создавать специфические обработчики, реализующие бизнес-правила (например, некоторые вычисления или ограничения), результат которых заносится в поля таблиц базы данных. Необходимо различать обращение к полю и обращение к его значению.

2.2.1. Обращение к полям

К полю можно обратиться, указав его имя несколькими способами:

– через имя данного компонента, определяемого свойством **Name**, если полю соответствует компонент **TField**, например:

```
T_SuppliersCatalogscName.Value := 'Рога и Копыта';
```

– используя метод **FieldByName** набора данных:

```
function FieldByName(const FieldName: String): TField;
```

например:

```
T_SuppliersCatalog.FieldByName('scName').Value := 'Рога и Копыта';
```

– используя свойство **Fields** набора данных:

```
property Fields[Index: Integer]: TField;
```

Индекс является порядковым номером поля в определении таблицы базы данных. Отсчет идет от нуля. Например, если поле **Supplier_Name** определено в таблице базы данных вторым по счету, то его индекс будет равен 1:

```
T_SuppliersCatalog.Fields[1].Value := 'Рога и Копыта';
```

– используя свойство набора данных:

```
property FieldValues[const FieldName: String]: Variant;
```

которое позволяет обращаться к полю через его имя, указываемое как содержимое параметра **FieldName**, например:

```
T_SuppliersCatalog.FieldValues['scName'] := 'Рога и Копыта';
```

Это свойство принимается для набора данных по умолчанию, поэтому его имя при обращении к полю можно опускать, например:

```
T_SuppliersCatalog['scName'] := 'Рога и Копыта';
```

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, исключая возможность обращения к несуществующему полю. К примеру, если поле **scName** удалено из структуры таблицы базы данных, то обращение к нему по имени из набора данных, ассоциированного с этой таблицей, приведет к ошибке.

Менее предпочтительным является обращение к полю через свойство набора данных **Fields**. Это связано с тем, что если поле **scName** было объявлено в таблице базы данных вторым по счету (например, обращение **Fields[1]**), а затем удалено из структуры этой таблицы, то это обращение в программном коде будет воспринято как обращение ко второму полю в ТБД. Однако фактически этим полем будет третье поле, следовавшее за полем **scName** перед тем, как оно было удалено. Таким образом, возникает алгоритмическая ошибка, которая может быть не распознана, если поле **scName** и поле, ставшее после удаления поля **scName** вторым, имеют одинаковые или совместимые типы. Такие ошибки очень трудно локализовать. Поэтому следует по возможности воздерживаться от обращения к полю через свойство **Fields**.

2.2.2. Обращение к значению поля

К значению поля можно обратиться при помощи свойств **Value** и **As*****.

Свойство

```
property Value: Variant;
```

возвращает значения следующих типов:

```
property Value: Variant;        // Все компоненты
property Value: String;         // TStringField, TBlobField
property Value: LongInt;        // TAutoIncField, TField, TWordField
```

```
                                    // TSmallIntField
property Value: Double;             // TBCDField, TCurrencyField,
                                    // TFloatField
property Value: Boolean;            // TBooleanField
property Value: TDateTime;          // TDateField, TDateTimeField,
                                    // TTimeField
```

Например, для поля **IDtg** типа **TIntegerField** таблицы **T_GoodsCatalog** справедлива запись

```
var N: Integer;
. . . . .
N := T_GoodsCatalogIDtg.Value;
```

Аналогичные значения возвращает рассмотренное в п. 2.2.1 свойство набора данных **FieldValues**.

При обращении к значению поля через свойство **As***** используются следующие свойства приведения типов полей:

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

Каждое из этих свойств приводит значение поля к соответствующему типу данных, обозначенному в названии свойства. Например, если поле **T_GoodsCatalogIDtg** является компонентом класса **TIntegerField**, то для приведения его к типу **String** можно воспользоваться свойством

```
Edit1.Text := T_GoodsCatalogIDtg.AsString;
```

Однако при этом нужно учитывать совместимость типа поля с типом данных, к которому приводится значение поля (табл. 2.1). Например, попытка привести поле **gcCost** (**TBCDField**) компонента НД **T_GoodsCatalog** к несовместимому типу **Boolean**

```
if T_GoodsCataloggcCost.AsBoolean then
```

во время выполнения программы приведет к ошибке.

Также возможно использование следующих специализированных функций преобразования типов данных:

а) приведение строки к определенному типу:

– приведение строки к вещественному типу:

```
function StrToFloat(const S: string): Extended;
```

– приведение строки к целочисленному типу:

```
function StrToInt(const S: string): Integer;
```

– приведение строки к типу **Дата**:

```
function StrToDate(const S: string): TDateTime;
```

– приведение строки к типу **Время**:

```
function StrToTime(const S: string): TDateTime;
```

б) приведение значения определенного типа в строку:

– приведение вещественного типа в строку:

```
function FloatToStr(Value: Extended): string;
```

– приведение целочисленного значения в строку:

```
function IntToStr(Value: Integer): string;
```

– приведение значения типа **Дата** в строку:

```
function DateToStr(const DateTime: TDateTime): string;
```

– приведение значения типа **Время** в строку:

```
function TimeToStr(const DateTime: TDateTime): string;
```

Таблица 2.1

Совместимость значений полей разных типов

Тип поля	**AsString**	**AsInteger**	**AsFloat**	**AsDateTime**	**AsBoolean**
TStringField	=	?	?	?	?
TWideString-Field	=	?	?	?	?
TIntegerField	+	=	+	×	×
TSmallIntField	+	=	+	×	×
TWordField	+	=	+	×	×

Тип поля	**AsString**	**AsInteger**	**AsFloat**	**AsDateTime**	**AsBoolean**
TFloatField	+	**+RI**	=	×	×
TCurrencyField	+	**+RI**	=	×	×
TBCDField	+	**+RI**	=	×	×
TDateTimeField	+	×	*1	=	×
TDateField	?	×	*1	=	×
TTimeField	?	×	*2	=	×
TBooleanField	*3	×	×	×	=
TBytesField	+	×	×	×	×

Окончание табл. 2.1

Тип поля	**AsString**	**AsInteger**	**AsFloat**	**AsDateTime**	**AsBoolean**
TVarBytesField	+	×	×	×	×
TBlobField	**memo**	×	×	×	×
TMemoField	**memo**	×	×	×	×
TGraphicField	**memo**	×	×	×	×

Примечание. В таблице применяются следующие обозначения:

= – типы равнозначны;

+ – преобразование возможно;

+**RI** – преобразование возможно с округлением до ближайшего целого;

? – преобразование происходит, если возможно; часто преобразование зависит от формата показа (свойство **DisplayFormat**);

× – преобразование не разрешено;

memo – имеет значение для **memo**-поля;

*1 – преобразование даты к числу дней, начиная с даты 01.01.0001;

*2 – преобразование времени делением на 24 часа;

*3 – преобразование в строку со значениями **True** или **False**.

При необходимости мы можем применять функции преобразования типов для того, чтобы работать с полями различных типов. Так, можно заносить в протокол текстовые записи, содержащие дату, числовые и иные значения. Помимо этого мы можем использовать универсальное поле ввода в текстовом формате (компонент **TEdit**) и при работе с разными полями выполнять их преобразование в нужный тип.

2.3. Форматирование отображения значений полей во время выполнения приложения

При отображении данных часто возникает задача модификации отображаемого значения для улучшения восприятия этих данных. Так, например, может потребоваться изменение структуры вывода значения (для даты – указать определенное расположение элементов, для числа – поставить разделители или ограничения на количество знаков после запятой) или определенное форматирование для значения (добавить дополнительную подпись к значениям, взять в кавычки, изменить регистр символов и т. п.). Для выполнения такого форматирования можно воспользоваться событием **OnGetText** со свойствами полей.

2.3.1. Событие OnGetText

Пусть поле таблицы базы данных хранится не в том виде, в котором оно должно быть представлено. Тогда его можно отформатировать перед тем, как оно будет показано в визуальных компонентах, работающих с данными, определив алгоритм форматирования в обработчике события **OnGetText** компонента **TField**. В процедуре обработчика присутствуют следующие параметры:

– **Text** – выдает отформатированное значение, показываемое в визуальных компонентах, связанных с таблицами базы данных;

– **DisplayText** – определяет, когда произошло событие **OnGetText**, при показе поля (значение **True**) или при модификации поля (значение **False**).

Например, пусть поле **scName** из набора данных **T_SuppliersCatalog** отображается в компоненте **DBGrid1**. При показе содержимое данного поля необходимо заключить в кавычки (хотя хранится оно без кавычек). Если мы захотим изменить его значение в какой-либо записи, то нам нужно отобразить содержимое поля заглавными буквами. Для этого в событиях компонента **TField** с именем **T_SuppliersCatalogscName** выберем событие **OnGetText** и запишем обработчик (рис. 2.6):

```
procedure TDataModule2. T_SuppliersCatalogscName GetText(Sender:
  TField; var Text: string; DisplayText: Boolean);
begin
  if DisplayText then
    Text := ' " '+ T_SuppliersCatalogscName.AsString+ ' " '
  else
    Text := AnsiUpperCase(T_SuppliersCatalogscName.AsString);
```

end;

Рис. 2.6. Назначение обработчика **OnGetText**

При показе записи из набора данных **T_SuppliersCatalog** в компоненте **DBGrid1** возникает событие **OnGetText** и вызывается обработчик **DisplayText** со значением **True**. В результате в компоненте **DBGrid1** весь столбец, соответствующий полю **scName**, будет содержать значения в кавычках. Если пользователь захочет изменить значение этого поля, то во время редактирования оно будет выдано заглавными буквами.

Если для поля был определен обработчик события **OnGetText**, то режимы форматирования, определенные в свойствах **DisplayFormat** и **EditMask** данного поля, игнорируются.

2.3.2. Свойство DisplayFormat

Свойство

property DisplayFormat : **String**;

применяется для форматирования данных при отображении полей. Это свойство поддерживает спецификаторы форматов в зависимости от типа. Для полей типа **TIntegerField**, **TSmallIntField**, **TwordField**, **TBCDField**, **TCurrencyField**, **TFloatField** спецификаторы приведены в табл. 2.2, а для полей типа **TDateField**, **TDateTimeField** и **TTimeField** – в табл. 2.3.

Таблица 2.2

Спецификаторы форматов для числовых полей

Спецификатор	Описание

0	Число. Если незначащий разряд равен 0, то показывает его
#	Число. Если незначащий разряд равен 0, то не показывает его
.	Десятичная точка. Разделяет целую и дробную часть числа. Принимается во внимание только первая точка, остальные игнорируются
,	Разделитель тысяч. Каждая группа чисел из трех разрядов в целой части отделяется от иных разрядов запятой
;	Разделитель положительного, отрицательного и нулевого значения
E+	Научный формат действительных чисел
"xx" или **'xx'**	Символы внутри двойных или одинарных парных кавычек не форматируются и выводятся как есть. Например, число 123.45 с форматом **#.# "рублей"** выведется как **'123.5 рублей'**

Таблица 2.3

Спецификаторы форматов для полей даты/времени

Спецификатор	Описание
c	Отображает дату в формате **ShortDateFormat**, затем время в формате **LongTimeFormat**. Время не отображается, если значение поля имеет нулевую дробную часть
d	Отображает число месяца без нуля в левом разряде (**1–31**)
dd	Отображает число месяца с нулем в левом разряде (**01–31**)
ddd	Отображает день недели в соответствии с сокращенными именами из переменной **ShortDayNames** (**пн – вс**)
dddd	Отображает день недели в соответствии с полными именами из переменной **LongDayNames** (**понедельник – воскресенье**)
ddddd	Отображает дату в соответствии с форматом **ShortDateFormat**
dddddd	Отображает дату в соответствии с форматом **LongDateFormat**
m	Отображает месяц как число без нуля в левом разряде (**1–12**)
mm	Отображает месяц как число с нулем в левом разряде (**01–12**)
mmmm	Отображает месяц в соответствии с полными именами из переменной **LongMonthNames** (**январь – декабрь**)
yy	Отображает год двумя цифрами
yyyy	Отображает год четырьмя цифрами
h	Отображает час как число без нуля в левом разряде (**0–23**)
hh	Отображает час как число с нулем в левом разряде (**00–23**)
n	Отображает минуты как число без нуля в левом разряде (**0–59**)
nn	Отображает минуты как число с нулем в левом разряде (**00–59**)
s	Отображает секунды как число без нуля в левом разряде (**0–59**)
ss	Отображает секунды как число с нулем в левом разряде (**00–59**)
t	Отображает время в соответствии с форматом **ShortTimeFormat**
tt	Отображает время в соответствии с форматом **LongTimeFormat**
am/pm	Отображает время в 12-часовой шкале. **'AM'** означает часы до полудня, **'PM'** – часы после полудня

a/p	Отображает время в 12-часовой шкале. При этом для времени до полудня отображается содержимое переменной **TimeAMString**, а после полудня – **TimePMString**
am/pm	Отображает время в 12-часовой шкале. '**a**' означает часы до полудня, '**p**' – часы после полудня
/	Отображает символ разделителя даты, содержащийся в переменной **DataSeparator**
:	Отображает символ разделителя времени, содержащийся в переменной **TimeSeparator**
'xx'/"xx"	Символы, заключаемые в простые или двойные кавычки, отображаются как есть и не подвергаются форматированию

Пусть значение поля равно **3456.777**. Тогда если свойство **DisplayFormat** имеет значение **'#.##'**, то будет показано **3456.78**.

Формат показа поля может быть динамически переназначен во время выполнения. Например, для набора данных, отображающего информацию о товаре (**Q_GoodsCatalogD**), нужно добавить возможность изменения формата поля **gcCost**. Для этого на соответствующей вкладке разместим компоненты **TEdit** с именем **Edit1** и **TButton** с именем **Button1**, для которого запишем обработчик события **OnClick**:

```
procedure TForm4.Button1Click(Sender: TObject);
begin
  DataModule2.Q_GoodsCatalogDgcCost.DisplayFormat := Edit1.Text;
end;
```

Свойство **DisplayFormat** игнорируется, если определен обработчик для события **OnGetText**.

2.4. Форматирование полей во время их редактирования

Для контроля корректности вводимых в поле значений применяется свойство

```
property EditMask: String;
```

Ограничения на вводимую информацию накладываются при помощи формата. При этом если введенный символ не удовлетворяет маске, то он не воспринимается. Для строковых полей значение дан-

ного свойства может использоваться для форматирования не только входных, но и выходных значений вместе со свойством **DisplayText**.

Маска представляет собой символьную строку, состоящую из трех частей:

– собственно маски;

– символа, определяющего, будут ли литералы (символ после указателя \) включаться в форматируемое значение как его часть (значение **1**) или не будут (значение **0**);

– символа, используемого для представления пробела.

Символы, которые могут входить в маску, приведены в табл. 2.4.

Таблица 2.4

Символы маски

Символ	Описание
!	Подавляет ведущие пробелы. Если этот символ в данных отсутствует, то хвостовые пробелы подавляются
>	Все следующие символы будут на верхнем регистре, пока не встретится символ <
<	Все следующие символы будут на нижнем регистре, пока не встретится символ >
<>	Регистр не проверяется
****	Следующий за этим знаком символ является литералом, т. е. включается в форматируемое значение
L	В этой позиции должен появиться только символ алфавита
l	Аналогично **L**, но символ в данной позиции может и отсутствовать
A	В этой позиции должен появиться только символ алфавита или цифра
a	Аналогично **A**, но символ в данной позиции может и отсутствовать
C	В позиции обязателен любой символ
c	Аналогично **C**, но символ в данной позиции может и отсутствовать
0	В данной позиции обязателен цифровой символ
#	В этой позиции должен появляться только цифровой символ, плюс или минус или не появляться никакой
:	Разделитель часов, минут и секунд для значения типа времени. Если в данной национальной кодировке для указанных целей используется иной символ, то он используется вместо символа **:**
/	Разделитель месяца, дня и года в датах. Если в данной национальной кодировке для указанных целей используется иной символ, то он используется вместо символа **/**
;	Разделитель частей маски
_	Заменитель пробела в маске

В модуле **Mask** имеются следующие константы, которые определяют некоторые компоненты маски по умолчанию (табл. 2.5).

Таблица 2.5

Значения маски по умолчанию

Имя константы	Начальное значение	Описание
DefaultBlank	_	Обозначение пробела в маске
MaskFieldSeparator	;	Разделитель частей в маске
MaskNoSave	**0**	Если 0, то символы маски не будут включаться в значение; если 1, то будут

Рассмотрим пример. Пусть имеется маска **'!\(999\) 000-0000;0;_'** и введено значение **'0952223344'**. Во время ввода оно представлялось на экране как **'(095) 222-3344'**, а запомнилось как **'(095) 222-3344'**. Если была бы использована маска **'!\(999\) 000\-00\-00;0;_'**, то во время ввода это значение представлялось бы на экране как **'(095) 222-33-44'**, а запомнилось бы как **'0952223344'**.

Настроим маску для поля **scPhone** таблицы **SuppliersCatalog** (компонент **T_SuppliersCatalog**). Выберем нужное поле в редакторе полей, после чего нажмем на кнопку с многоточием свойства **EditMask** в инспекторе объектов и зададим маску для редактирования телефона поставщика в федеральном формате **+7-XXX-XXX-XXXX** (рис. 2.7).

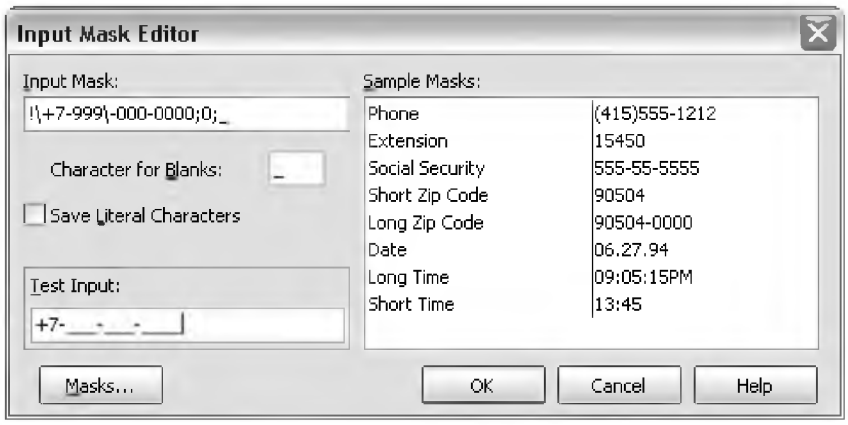


Рис. 2.7. Окно редактора маски

Свойство

property EditMaskPrt: **String**;

возвращает значение маски редактирования. Поскольку его свойство доступно только для чтения, то его следует использовать вместо свойства **EditMask** в тех случаях, когда маска должна быть только прочитана.

Свойство

```
property EditFormat: String;
```

применяется для форматирования значений полей перед их редактированием.

Для полей типа **TIntegerField**, **TSmallIntField**, **TWordField** форматирование выполняется функцией **FloatToTextFmt**. Если свойству **DisplayFormat** не была назначена строка, то значение форматируется посредством функции **Str**.

В полях типа **TDateField**, **TDateTimeField**, **TTimeField** форматирование происходит с помощью функции **DateTimeToStr**. Если свойству **DisplayFormat** не присвоена строка, то значение форматируется согласно спецификациям **Windows**.

Для полей типа **TBCDField**, **TCurrencyField**, **TFloatField** форматирование выполняется посредством функции **FloatToTextFmt**. В случае когда свойству **DisplayFormat** не присвоена строка, значение форматируется согласно значению свойства **Currency**.

Свойство

```
property Text: String;
```

содержит строковое изображение значения поля в том виде, в котором оно показывается в визуальном компоненте, когда НД находится в режиме редактирования (**dsEdit**).

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

2.5. Проверка введенного в поле значения

Рассмотрим средства проверки данных значений, вводимых в базу данных, основанных на обработчиках **OnSetText**, **OnValidate**, **OnChange** и **BeforePost**.

2.5.1. Свойство IsNull и обработчики OnSetText, OnValidate, OnChange

Для проверки, содержит ли поле какие-либо данные, используется свойство **IsNull**, которое во время выполнения возвращает значение **True**, если поле содержит пустое значение:

```
property IsNull: Boolean;
```

Проверить введенное в поле значение на его соответствие некоторым ограничениям или условиям можно в обработчике события **OnValidate**. Оно наступает при изменении значения поля вручную или программно до выполнения метода **Post**, запоминающего изменения БД. Поэтому если полю присвоено неверное значение, то выполнение метода **Post** можно предотвратить с помощью метода **Abort** или вызова исключительной ситуации (**raise Exception.Create**). Для новых записей событие **OnValidate** выполняется аналогично, поскольку при занесении значений в поля пустые значения заменяются на непустые, т. е. модифицируются. Такой подход к контролю корректности значений является ориентированным на поля.

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

Для примера выполним в поле **T_SuppliersCatalogscEmail** проверку на наличие символа **@**:

```
procedure TDataModule2.T_SuppliersCatalogscEmailValidate(Sender:
  TField);
begin
  if not (pos('@', T_SuppliersCatalogscEmail.AsString) > 0) then
    raise Exception.Create('указан неверный e-mail');
end;
```

Мы также можем использовать альтернативный код, но для его функционирования в случае размещения компонентов в контейнере **TDataModule** в блок **uses** раздела **interface** исходного кода нужно добавить **Dialogs**:

```
procedure TDataModule2.T_SuppliersCatalogscEmailValidate(Sender:
  TField);
begin
  if not (pos('@', T_SuppliersCatalogscEmail.AsString) > 0) then
```

```
  begin
    ShowMessage('В поле отсутствует символ @!');
    Abort;
  end;
end;
```

Если символ **@** не содержится в значении, присвоенном полю, то принудительно вызванная исключительная ситуация или метод **Abort** не позволят выполнить метод **Post** и запись с неверным данными не будет физически сохранена в БД, а НД останется в том состоянии, в котором он находился – в режиме редактирования **dsEdit** или добавления новой записи **dsInsert**.

Для проверки введенного значения может быть использовано и другое событие – **OnSetText**. Подобно событию **OnValidate**, оно возникает при изменении значения поля. Однако если на момент события новое значение полю не было присвоено, т. е. этого не было сделано программно в обработчике события, то значение в БД не сохранится.

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

Для примера выполним проверку введенного значения в поле **gcCost** таблицы **GoodsCatalog** (НД **T_GoodsCatalog**), которое должно быть выше некоторой минимальной цены, допустим 5 руб.:

```
procedure TDataModule2.T_GoodsCataloggcCostSetText(Sender:
  TField; const Text: string);
var
  Tmp: Real;
begin
  Tmp := StrToFloat(Text);
  if Tmp < 5 then
     ShowMessage('Значение, должно быть >= 5')
  else
    T_GoodsCataloggcCost.Value := Tmp;
end;
```

Событие **OnChange** может быть использовано для тех же целей, что и событие **OnValidate**. В качестве примера выполним дополнительную проверку, но уже введя неравенство, показывающее, что значение в поле **gcCost** должно быть не больше 1 000:

```
procedure TDataModule2.T_GoodsCataloggcCostChange(Sender:
  TField);
begin
  if T_GoodsCataloggcCost.Value > 1000 then
    raise Exception.Create('Значение, должно быть <= 1000');
end;
```

При изменении значения поля события, обработчики которых позволяют контролировать правильность занесенного в поле значения, вызываются в следующем порядке:

– **OnSetText**;
– **OnValidate**;
– **OnChange**.

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

Про событие **OnSetText** можно сказать, что если полю присвоено ошибочное значение, то нет необходимости выполнять метод **Abort** или вызывать исключительную ситуацию, чтобы предотвратить занесение этой записи в таблицу базы данных, поскольку новое значение в поле в этом случае еще не занесено. Если же поле удовлетворяет критериям корректности, то в него нужно программно записать введенное пользователем новое значение, передаваемое в обработчик как параметр **const Text: String**.

В обработчиках событий **OnValidate** и **OnChange** в этом случае, наоборот, необходимо выполнить метод **Abort** или вызывать исключительную ситуацию для предотвращения занесения этой записи в таблицу базы данных в связи с тем, что 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 вызывает исключительную ситуацию для

того, чтобы измененная запись с некорректным значением поля не была записана в таблицу базы данных.

2.5.2. Обработчик BeforePost

В качестве дополнительного средства проверки данных можно использовать обработчик события **BeforePost**. Это событие происходит в компоненте **TADOTable** перед пересылкой любых изменений, выполненных в текущей записи. Причиной наступления события может быть вмешательство пользователя либо вызов в программном коде метода **Post**. Этот метод можно сгенерировать неявно, переместившись с текущей записи. Если в процедуре обработчика события **BeforePost** вызвать исключение, то можно предотвратить пересылку данных.

Типичным применением процедуры обработчика событий **BeforePost** является реализация правил проверки для всей записи. Если запись не удовлетворяет проверке, то генерируется исключение и некорректная запись не принимается.

Для рассматриваемой нами таблицы **GoodsCatalog** зададим обработчик **BeforePost**, в котором при сохранении записи происходит проверка заполнения всех необходимых текстовых полей. При этом длина названия товара должна быть не меньше трех символов, а единицы измерения – двух. Для этого запишем следующий код:

```
procedure TDataModule2.T_GoodsCatalogBeforePost(DataSet:
  TDataSet);
begin
  if (Length(T_GoodsCataloggcName.AsString) < 3) or
    (Length(T_GoodsCataloggcMeasure.AsString) < 2) then
      raise Exception.Create('Наименования товара и единицы
измерения не должны иметь пустые значения или короткие записи');
end;
```

Проверки подобного рода желательно выполнять при работе с текстовыми полями. Однако событие **BeforePost** можно использовать для организации проверки не только текстовых полей, но и полей других типов. Например, у полей, указывающих диапазон, можно

проверить, чтобы левая граница диапазона всегда была меньше правой, и т. п.

2.6. Создание полей выбора данных

Кроме обычных полей, связанных с полями таблицы базы данных, в среде **Delphi** имеется возможность создавать поля выбора данных (**lookup**).

Поля выбора данных одного набора данных содержат значения из другого набора данных, связанного по первичному ключу с НД, которому принадлежит поле выбора данных. Поле выбора данных всегда доступно только для чтения и не может быть одновременно и полем выбора данных, и вычисляемым полем. Реляционное отношение НД, служащего источником значений для поля выбора данных, и НД, которому оно принадлежит, имеет связи типа «один-ко-многим» и реже связи типа «один-к-одному». Это означает, что на один вариант значения в НД-источнике должно приходиться одно или несколько связанных значений в НД, к которому принадлежит поле выбора.

Для определения поля набора данных создадим новое поле в редакторе полей, установив радиогруппу **Field Type** в значение **Lookup**. Затем зададим значения свойств:

– **DataSet** – имя НД – источника значений для поля выбора данных;

– **Key Fields** – индексные поля набора данных – владельца поля выбора данных. По этим полям НД-владелец соединяется с НД – источником значений поля выбора данных. Если в индексе имеется несколько полей, то они перечисляются через точку с запятой;

– **Lookup Fields** – индексные поля НД – источника значений для поля выбора. По значениям этих индексных полей устанавливается связь НД-источника со значениями индексных полей НД – владельца поля выбора, указанными в параметре **Key Fields**. Если в индексе имеется несколько полей, то они перечисляются через точку с запятой;

– **Result Field** – поле НД-источника, возвращаемого в качестве результата. Необходимо следить, чтобы тип вновь создаваемого поля и поля результата совпадали.

Указанным выше параметрам редактора полей соответствуют свойства компонента **TField**:

```
property LookupDataSet: TDataSet;
property KeyFields: String;
```

property LookupKeyFields: **String**;

property LookupResultField: **String**;

Аналогичной по последствиям будет установка соответствующих свойств в инспекторе объектов для вновь добавляемого поля. При этом свойство поля

property Lookup: Boolean;

должно быть установлено в значение **True**.

Рассмотрим пример использования отношения «один-ко-многим» для организации поля выбора данных. Для этого воспользуемся НД **Q_IncomingGoodsD**, отображающим результат запроса по таблицам **IncomingGoods**, и НД **GoodsCatalog**. При этом отображение цены товара, т. е. создаваемое поле выбора данных, реализуем с использованием компонента **TDBText**. Для формирования поля выбора данных в компоненте **Q_IncomingGoodsD**, связанном с НД **Приход товаров**, войдем в редактор полей и создадим новое поле **Cost** типа **TCurrencyField** с именем **Q_IncomingGoodsDCost**, сразу же установим радиогруппу **Field Type** в значение **Lookup**. После этого настроим значения свойств в блоке **Lookup definition** (рис. 2.8):

– **DataSet** – НД **T_ GoodsCatalog** (ТБД **Товары**);

– **Key Fields** – поле **IDgc** НД **Q_IncomingGoodsD**;

– **Lookup Fields** – поле **IDgc** НД **T_GoodsCatalog** (ТБД **Товары**);

– **Result Field** – поле **gcCost** НД **T_GoodsCatalog** (ТБД **Товары**).

Таким образом, нами определен новый компонент **Q_IncomingGoodsDCost** типа **TCurrencyField**, источником данных для которого служит поле **gcCost**, из той записи таблицы **GoodsCatalog**, у которой значение поля **IDgc** совпадает со значением поля **IDgc** в НД таблицы **Приход товаров (Q_IncomingGoodsD)**.

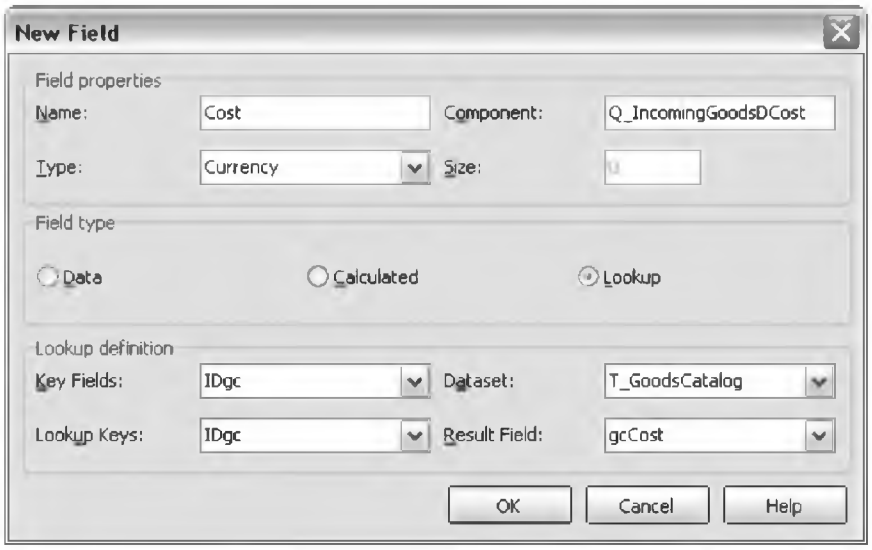


Рис. 2.8. Окно создания полей с настройками
поля выбора данных

Далее разместим на форме **Form1** компонент **DBText1** и свяжем его с полем **Q_IncomingGoodsDCost**, а поле **Cost** таблицы уберем из компонента **DBGrid2**, используя редактор столбцов. Тогда для текущей записи товара компонент **DBText1** отобразит значение цены товара из таблицы **Товары**. Перед компонентом **DBText1**, т. е. слева от него, можно разместить компонент **TLabel** с именем **Label1**, в свойстве которого подписать **Цена товара** (рис. 2.9).

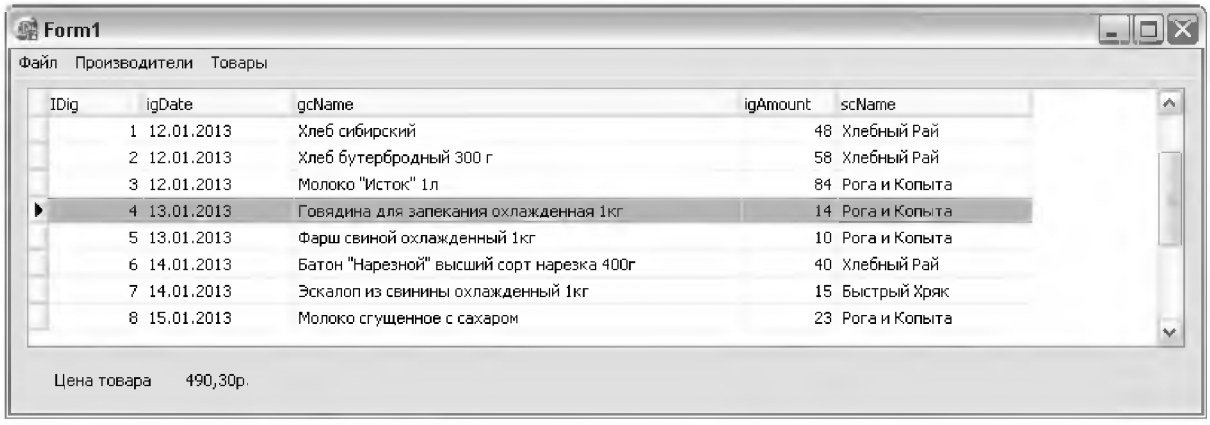


Рис. 2.9. Создание поля выбора данных

Поля выбора данных позволяют отобразить информацию из связанных таблиц, что значительно расширяет возможности по проектированию приложения базы данных. Так, например, в таблице **T_GoodsCatalog** можно отобразить поле с названием типа товаров и при этом можно не использовать **SQL**-запрос.

2.7. Создание вычисляемых полей

Если необходимо создать вычисляемое поле, значение которого определяется по значениям других полей, то поступим следующим образом.

В редакторе полей создадим новое поле, пометив его как **Calculated**. Для этого сделаем текущим необходимый НД, нажав правую кнопку мыши и выбрав элемент меню **Field Editor**. В редакторе полей снова нажмем правую кнопку мыши и выберем элемент меню **New Field**. Затем в окне диалога введем имя поля, его тип и для строковых полей – длину. Для нового поля будет создан компонент **TField**, доступ к которому осуществляется в редакторе полей.

Для компонента НД, к которому принадлежит вычисляемое поле, определим обработчик событий **OnCalcFields**. Это событие возникает всякий раз, когда курсор (указатель записи) перемещается в НД от записи к записи (например, после выполнения методов **Next**, **Last** и т. д., или при движении по записям в компоненте **TDBGrid** вручную). Данное событие происходит и при инициации НД после открытия, а также после фильтрации записей в НД, что, впрочем, также связано с изменением положения указателя записи.

Кроме того, если свойство набора данных **AutoCalcFields** установлено в значение **True**, то событие **OnCalcFields** наступает и при модификации значений невычисляемых полей в режимах **dsInsert** и **dsEdit** данного НД или НД, реляционно с ним связанного, т. е. когда установлены ограничения целостности в самой таблице базы данных, а не тогда, когда они подразумеваются.

Процедура-обработчик события **OnCalcFields** содержит реализацию алгоритма подсчета значения вычисляемого поля или группы полей. Необходимо помнить, что в этом обработчике значение может быть присвоено только вычисляемому полю, а не полю, определенному в структуре таблицы БД.

Для примера добавим вычисляемое поле **TotalCost**, которое будет отображать полную стоимость пришедшего товара в НД **Q_IncomingGoodsD**, ассоциированном с таблицей базы данных **Приход товаров** (рис. 2.10).

После этого в обработчике события **OnCalcFields** компонента **Q_IncomingGoodsD** запишем следующий код:

```
procedure TDataModule2.Q_IncomingGoodsDCalcFields(DataSet:
```

```
    TDataSet);
begin
Q_IncomingGoodsDTotalCost.Value := Q_IncomingGoodsDigAmount.Value *
     Q_IncomingGoodsDCost.Value;
end;
```

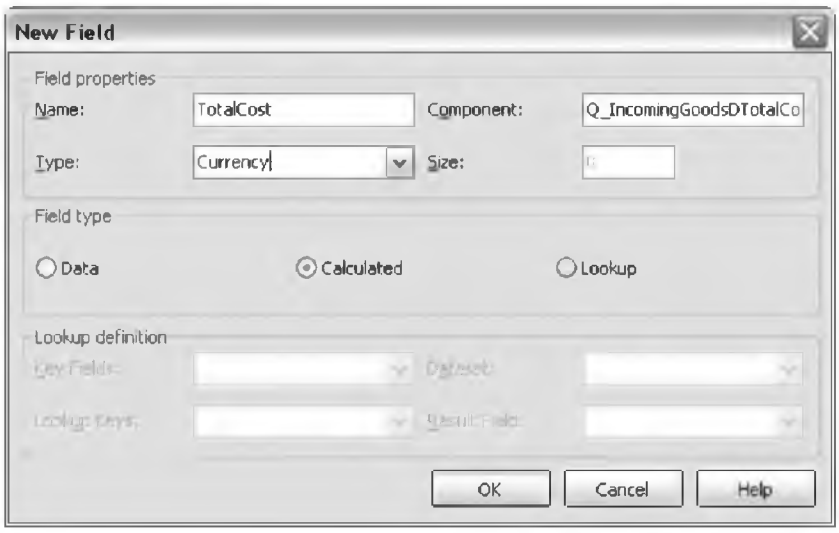


Рис. 2.10. Окно создания полей с настройками вычисляемого поля

Для отображения полной стоимости (**TotalCost**) в компоненте **DBGrid1** на форме **Form1** выполним добавление нового поля и в значении свойства **FieldName** укажем **TotalCost**.

Пример выполнения программы с добавленным вычисляемым полем представлен на рис. 2.11.

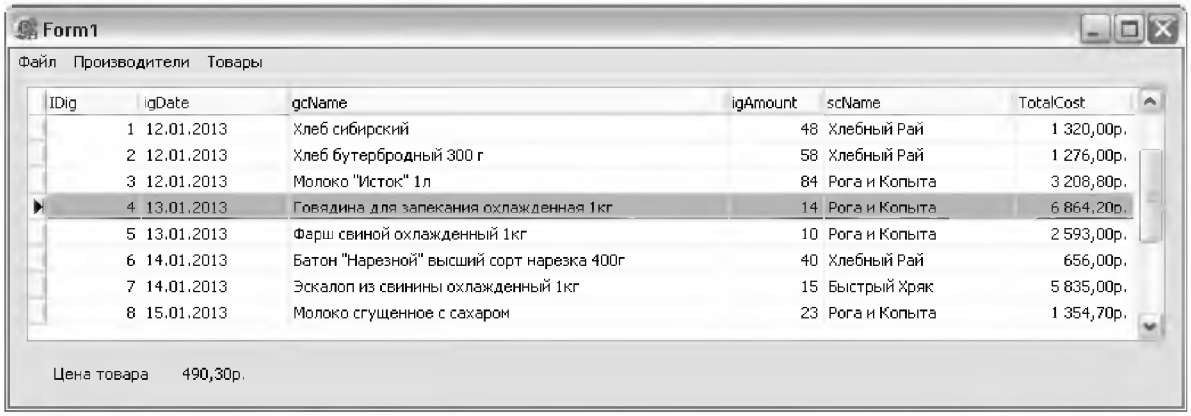


Рис. 2.11. Создание вычисляемого поля

Как можно заметить, в данном примере полная стоимость автоматически рассчитывается на основании информации из двух наборов

данных. Это дает преимущество в разработке базы данных, поскольку у нас нет необходимости хранить полную стоимость товара в таблице **IncomingGoods**.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Создать проект и подключить к нему модуль **TDataModule**. Разместить в контейнере необходимые компоненты для работы с БД и задать **SQL**-запросы для отображения списка товаров и прихода товаров от поставщиков (см. п. 2.1.1).
2. Добавить в проект две формы и настроить их взаимодействия (см. п. 2.1.2).
3. Организовать главное меню для основной формы. Для компактного отображения и работы с данными использовать разные формы и компонент **PageControl** (см. п. 2.1.3).
4. Ознакомиться со способами получения доступа к значениям полей таблиц и преобразованием типов данных в среде **Delphi** (см. пп. 2.2.1, 2.2.2).
5. Реализовать обработчик **OnGetText** на поле с именем поставщика (см. п. 2.3.1).
6. Применить свойство **DisplayFormat** при форматировании данных в поле цены товаров (см. п. 2.3.2).
7. Задать маску редактирования данных на поле **Телефон поставщика** (см. п. 2.4).
8. Организовать проверку корректности вводимых данных с использованием событий **OnValidate**, **OnSetText**, **OnChange** (см. п. 2.5.1).
9. Реализовать проверку вводимых данных в таблицу **Товары** с использованием обработчика **BeforePost** (см. п. 2.5.2).
10. Создать поле выбора данных (см. п. 2.6).
11. Создать вычисляемое поле (см. п. 2.7).
12. Ответить на контрольные вопросы.
13. Составить отчет, который должен содержать титульный лист, цель и ход лабораторной работы, сформулированные выводы.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Для каких целей необходим контейнер **TDataModule**?
2. Какими признаками отличается модальная форма от обычной?
3. Охарактеризуйте способ обращения к полю по свойству **Name**.
4. Опишите способ обращения к полю по методу **FieldByName**.
5. Каким образом реализуется способ обращения по свойству **Fields[индекс]**?
6. Дайте характеристику способа обращения к полю по свойству **FieldValues**.
7. Сравните способы обращения к полям.
8. В каком случае важен порядок следования полей?
9. Каким образом происходит обращение к значению поля через свойство **Value**?
10. Опишите обращение к значению поля через свойство **As*****.
11. Какие преобразования типов полей разрешены?
12. Перечислите запрещенные преобразования типов полей.
13. Каким образом осуществляется динамическое переназначение формата показа поля во время выполнения приложения?
14. Дайте характеристику события **OnGetText**.
15. Опишите свойство **DisplayFormat**.
16. Перечислите известные вам спецификаторы форматов.
17. Приведите примеры форматирования числа **123456.789**.
18. Представьте примеры форматирования числа **–123456.789**.
19. Каким образом можно отформатировать число **0.35**?
20. Покажите примеры форматирования числа **–0.35**.
21. Приведите примеры форматирования числа **0**.
22. Каким образом осуществляется динамическое переназначение формата показа поля во время выполнения приложения?
23. Дайте характеристику свойства **EditMask**.
24. Опишите структуру маски свойства **EditMask**.
25. Какие символы входят в маску?
26. Приведите примеры использования маски для отображения и хранения значений даты/времени.
27. Дайте характеристику свойства **EditMaskPrt**.
28. Для каких целей предназначено свойство **EditFormat**?
29. Опишите событие **OnSetText**.
30. Укажите область применения события **OnValidate**.
31. Охарактеризуйте событие **OnChange**.

32. Сравните способы проверки введенного в поле значения.
33. Какое исключение генерирует функция **Abort**?
34. Каким образом указываются ограничения на значения полей?
35. Опишите порядок создания вычисляемых полей.
36. В каком порядке создаются поля выбора данных?
37. Каким образом можно использовать буфер значений поля выбора?
38. С помощью каких средств можно определить вид поля?

Лабораторная работа 3

НАВИГАЦИЯ ПО НАБОРУ ДАННЫХ. СЦЕНАРИЙ ОБНОВЛЕНИЯ ЗАПИСЕЙ С КОМПОНЕНТОМ TDBGrid

Цель работы: ознакомиться со способами навигации по набору данных; разработать приложение, иллюстрирующее навигацию по набору данных; разработать приложение для работы с закладками и обновления записей на одной форме с компонентом **TDBGrid**.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Существуют два способа работы с записями набора данных.

Первый способ основан на использовании операторов **SQL** и предполагает оперирование группами записей. Именно так работают **SQL**-операторы группового обновления НД **UPDATE**, **INSERT**, **DELETE** и выборки групп записей **SELECT**. Записи, удовлетворяющие некоторому условию, выдаются группами и даже если условию удовлетворяет только одна запись, то считается, что группа состоит из одной записи.

Второй способ состоит в оперировании единичными записями. При измении, добавлении, удалении группы записей необходимая операция выполняется для каждой из таких записей. Для этого записи нужно найти в НД, для чего применяются навигационные методы. Они всегда работают с единичной записью и связаны с понятием курсора НД.

3.1. Навигация по набору данных

Рассмотрим работу навигационных методов, для чего необходимо ввести понятие курсора. Под *курсором* набора данных понимается указатель текущей записи в конкретном наборе данных. Текущей записью является такая запись, над которой в данный момент времени можно выполнять какие-либо операции (удаление, изменение, чтение значений, содержащихся в записи).

Существует пять методов изменения курсора набора данных (табл. 3.1).

Таблица 3.1

Методы изменения курсора НД

Метод	Выполняемые действия
procedure First;	Устанавливает курсор на первую запись в наборе данных
procedure Last;	Устанавливает курсор на последнюю запись в наборе данных
procedure Next;	Перемещает курсор на следующую запись в наборе данных
procedure Prior;	Перемещает курсор на предыдущую запись в наборе данных
function MoveBy (n: Integer): Integer;	Перемещает курсор на **n** записей к концу набора данных (**n** > 0) или к началу набора (**n** < 0)

При перемещении по набору данных может возникнуть задача проверки местоположения курсора, т. е. того, находится ли курсор в начале набора данных или в конце. Для решения этой задачи используются свойства **BOF** и **EOF** набора данных.

Свойство

```
property BOF: Boolean;
```

возвращает значение **True**, если курсор установлен на первую запись в НД.

Свойство

```
property EOF: Boolean;
```

возвращает значение **True**, если курсор установлен на последнюю запись в НД.

Свойство набора данных

```
property RecordCount: Integer;
```

возвращает текущее число записей в НД.

При работе, связанной с навигацией по набору данных, может происходить ряд событий, для которых необходима организация дополнительной обработки данных НД или взаимосвязи с другими НД.

Событие **OnDataChange** компонента **TDataSource** возникает всякий раз при изменении курсора НД, т. е. при переходе к новой текущей записи. Это событие наступает, когда курсор БД уже находится на новой записи, и происходит в режимах **dsInsert** и **dsEdit**:

– при изменении какого-либо поля;

– первом перемещении с измененного поля на другое поле.

При переходе к новой записи также возникают два события с компонентами типа НД (**TADOTable**, **TADOQuery** и т. п.):

property BeforeScroll: TDataSetNotifyEvent;

которое наступает перед переходом на другую запись в наборе данных, и

property AfterScroll: TDataSetNotifyEvent;

которое наступает после перехода на другую запись в наборе данных.

Теперь перейдем к рассмотрению особенностей взаимосвязи отображаемых данных НД и хранимых данных ТБД, связанных с НД.

3.1.1. Порядок следования и порядок сортировки записей

При работе программы у пользователя может сложиться впечатление, что первая и последняя записи набора данных всегда фиксированны и что это физически первая и последняя записи в наборе данных. Однако это неверно. Как уже отмечалось в лабораторной работе 1 (см. п. 1.3.6), набор данных может содержать часть записей из таблицы базы данных. Поэтому набор данных – это понятие логическое, а не физическое.

Для компонента **TADOTable** последовательность расположения записей в наборе данных определяется используемым индексом, обусловливающим сортировку. Для компонента **TADOQuery** порядок следования записей либо случаен, особенно при использовании в качестве источника более одной таблицы базы данных, либо упорядочен в соответствии с перечислением полей в операторе **ORDER BY**.

Сортировку записей в компонентах НД **TADOTable** и **TADOQuery** можно изменить, не используя для этого индекс, установленный при открытии НД или **SQL**-запроса, с помощью свойства

property Sort: WideString

Особенностью использования данного свойства для изменения порядка следования записей является то, что для сортируемого поля создается временный индекс. В общем виде формат записи для этого свойства будет выглядеть следующим образом:

```
ADOTable.Sort := 'ИмяПоля1 ТипСортировки1, ИмяПоля2
   ТипСортировки2';
```

Сортировку можно выполнять как по возрастанию (значение **ASC**), так и по убыванию (значение **DESC**). В случае если тип сортировки не указан, сортировка будет выполняться по возрастанию.

Для того чтобы отменить сортировку, т. е. установить сортировку по умолчанию, которая была на момент открытия набора данных, свойству **Sort** необходимо присвоить пустое значение:

```
ADOTable.Sort := '';
```

В начале работы НД может иметь одну сортировку записей, в дальнейшем эту сортировку можно переопределить и записи перестроятся в соответствии с новой сортировкой, т. е. логический порядок их следования изменится.

Таким образом, при изучении вопросов навигации по НД речь прежде всего идет о логическом характере следования записей, поскольку физический характер их расположения в каждом конкретном случае неизвестен.

3.1.2. Навигация по набору данных вниз и вверх

Для выполнения действий с некоторой стартовой записи и до конца набора данных используется цикл **while not EOF**.

Приведем фрагмент обработчика события **OnClick** кнопки для случая, когда стартовая запись является первой в наборе данных:

```
with ADOTable1 do
  begin
    First;
    while not EOF do
    begin
     {Какие-либо действия}
    Next;
    end;     // While
  end;       // With
```

Можно использовать и альтернативный код:

```
ADOTable1.First;
while not ADOTable1.EOF do
  begin
    {Какие-либо действия}
```

```
    ADOTable1.Next;
  end;        // While
```

Как можно заметить, в первом случае используется конструкция **with *** do**, которая позволяет вызывать методы или элементы сложной переменной, компонента, типа записи или объекта. Эта конструкция упрощает код, удаляя потребность в префиксе для каждого упомянутого элемента переменной со сложным именем.

Для выполнения действий, начиная от некоторой стартовой записи и до начала набора данных, используется цикл **while not BOF**.

Приведем пример фрагмента кода для случая, когда стартовая запись является последней в наборе данных:

```
with ADOTable1 do
  begin
    Last;
    while not BOF do
      begin
       {Какие-либо действия}
       Prior;
    end;     // While
  end;       // With
```

3.1.3. Случайные перемещения по набору данных

Часто бывает необходимо перемещаться по НД в разных направлениях. Поэтому распространен вариант одновременного использования методов **Next**, **Prior** и **MoveBy** в одном программном блоке. При этом важно помнить о том, что применение метода **Edit**, когда изменяется значение индексного поля, по которому в настоящий момент ведется сортировка в НД, может переместить запись вниз или вверх. Следует всегда придерживаться правила: *не изменять значения индексного поля при прохождении набора данных в цикле*.

Для решения этой проблемы можно предложить способ, который состоит в простом переборе записей при отключенной сортировке по исходному индексу или при сортировке по другому индексу. Возможен и другой способ, имеющий более узкое применение, так как в этом случае индексное поле у всех записей должно иметь одно и

то же значение. Этот способ состоит в использовании временной фильтрации и часто применяется при работе со связанными НД – родительским и дочерним. Отметим, что фильтрация в данном случае обеспечивается свойством **Filtered**, общим для компонентов **TADOTable** и **TADOQuery**.

При выполнении действий с НД, имеющим большое количество записей, что влечет за собой частое изменение местоположения курсора БД, в визуальном компоненте, показывающем записи (например, **TDBGrid**) или текущую запись (например, **TDBEdit**), будет возникать эффект прокрутки записей, который не всегда может устраивать пользователя. Кроме того, при смене местоположения курсора БД, т. е. при смене текущей записи НД, необходимо дополнительное время для отражения в визуальном компоненте произошедших изменений.

Для устранения прокрутки используются два метода:

– **procedure** DisableControls;

– **procedure** EnableControls;

Первый метод отключает связь с визуальным компонентом, а второй восстанавливает ее. При этом в таблице компонента **TDBGrid** не будет видно эффекта прокрутки записей. Наоборот, у пользователя возникнет иллюзия, что курсор БД сразу переустановился с прежней записи набора данных на нужную запись.

3.2. Внесение изменений в набор данных

Свойство набора данных

property CanModify: Boolean;

определяет, может ли НД переводиться в состояния **dsInsert** и **dsEdit** (свойство **CanModify** имеет значение **True**) или не может (свойство **CanModify** установлено в значение **False**). Это свойство зависит от значения свойства **ReadOnly** НД. Если **ReadOnly** имеет значение **True**, то свойство **CanModify** автоматически переводится в значение **False**. В противном случае свойство **CanModify** может принимать как значение **True**, так и значение **False**, устанавливая таким образом возможность изменения НД в зависимости от каких-либо условий.

Свойство **AutoEdit** компонента **TDataSource**, связанного с НД, определяет, возможен ли (значение **True**) или невозможен (значение **False**) автоматический перевод НД в состояние **dsEdit**. В последнем случае для изменения НД программа должна вызвать метод **Edit**.

Свойство **AutoEdit** компонента **TDataSource** не влияет на возможность перевода НД в состояние **dsEdit** или **dsInsert**. Для запрета перехода в режим **dsInsert** достаточно либо сделать НД открытым только для чтения (свойство НД **ReadOnly** имеет значение **True**), либо установить режим **ReadOnly** в значение **True** для полей, входящих в состав первичного ключа. В этом случае корректировка записи будет возможна для всех полей за исключением указанных.

Набор данных может автоматически переводиться в состояния **dsInsert** или **dsEdit**, если пользователь в визуальном компоненте, связанном с НД, выполнит определенные действия. Вид этих действий зависит от визуального компонента, связанного с НД. Например, для перехода в режим **dsEdit** в компоненте **TDBGrid** достаточно изменить значение любого поля; в компонентах **TDBEdit** или **TDBMemo**, связанных с отдельными полями НД, следует изменить значение поля, с которым связаны **TDBEdit** или **TDBMemo**; для компонента **TDBNavigator** (рис. 3.1) нужно нажать соответствующую кнопку, и т. д.

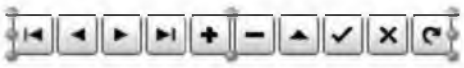


Рис. 3.1. Вид компонента **TDBNavigator**

Компонент **TDBNavigator** позволяет управлять связанным с ним НД. В этом компоненте реализованы возможности навигации и манипулирования данными. При необходимости в нем можно создать список отображаемых кнопок, для чего нужно настроить свойства группы **VisibleButtons** в инспекторе объектов.

3.2.1. Изменение текущей записи

Чтобы изменить запись, нужно перевести набор данных методом **Edit** из состояния **dsBrowse** в состояние **dsEdit**, затем поменять значение одного или нескольких полей записи и использовать метод **Post** для запоминания записи в НД. Метод **Post** в данном случае при успешном исполнении переведет НД из состояния **dsEdit** в состояние **dsBrowse**. Для отказа от запоминания измененной записи в НД применяется метод **Cancel**. Он также переводит НД из состояния **dsEdit** в состояние **dsBrowse**.

Метод **Edit** будет выполняться только в том случае, когда редактирование записи разрешено (свойство **ReadOnly** имеет значение

False). Помимо этого для корректировки могут быть запрещены отдельные поля записи (свойство **ReadOnly** соответствующих компонентов **TField** имеет значение **True**).

Метод **Edit** может вызываться:

– программно;

– автоматически, когда пользователь в визуальном компоненте, связанном с НД, выполнит определенные действия, вид которых будет зависеть от визуального компонента. Автоматический перевод набора данных в режим редактирования должен быть разрешен свойством **AutoEdit** соответствующего компонента **TDataSource**, установленного в значение **True**.

Заполнение записи данными в режиме **dsEdit** может выполняться двумя способами. Первый способ заключается в задании значений для каждого поля, например при помощи компонентов **TDBEdit** или непосредственного определения значений полей в программном коде. Второй способ состоит в одновременном заполнении необходимых полей нужными значениями. Для этих целей применяется метод **SetFields**:

```
procedure SetFields(const Values: array of const);
```

Реализацию этого метода рассмотрим на следующем примере. Пусть имеется таблица с тремя полями. Первое поле является ключевым и имеет тип автоинкремент (счетчик), необходимо заполнить (изменить) второе и третье поля значениями из компонентов **TEdit**. Для этого напишем следующий обработчик:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ADOTable1.Edit;
  ADOTable1.SetFields([nil, Edit1.Text, Edit2.Text]);
  ADOTable1.Post;
end;
```

При выполнении этого кода НД сначала переведется в режим **dsEdit**. Затем полям записи будут присвоены значения, перечисленные в открытом массиве **Values**. При этом первое значение в списке присваивается первому полю, второе – второму и т. д. Естественно, что значения в списке должны быть совместимы с теми полями, которым они присваиваются. Если в списке число значений меньше числа

полей записи, то те поля, которым не хватило значений, сохранят свое первоначальное значение. И наоборот, если значений больше, чем нужно для заполнения полей, то лишние значения потеряются. В случае если полю будет передано значение **nil**, оно сохранит свое текущее значение. После успешного присваивания значения полям записи выполняется метод **Post**.

Код с методом **SetFields** обычно выполняется после того, как пользователь ввел значения в переменные и был произведен контроль корректности этих значений.

3.2.2. Добавление новой записи

Чтобы добавить новую запись в НД, нужно вызвать метод **Insert** для перевода НД из состояния **dsBrowse** в состояние **dsInsert**, а затем произвести присваивание значения одному или нескольким полям записи, после чего выполнить метод **Post** для запоминания новой записи в НД. Метод **Post** при успешном выполнении переводит НД из состояния **dsEdit** в состояние **dsBrowse**. Для отказа от запоминания новой записи в НД используется метод **Cancel**. Он также переводит НД из состояния **dsEdit** в состояние **dsBrowse**.

При добавлении записи изменение набора данных должно быть разрешено (свойство **ReadOnly** должно быть установлено в значение **False**). Помимо этого для изменения могут быть запрещены отдельные поля записи (свойство **ReadOnly** соответствующих компонентов **TField** установлено в значение **True**). В этом случае в них нельзя ввести новые значения.

Метод **Insert** может вызываться:

– программно;

– автоматически, когда пользователь в визуальном компоненте, связанном с набором данных, предпринимает соответствующие действия. Для перехода в режим **dsInsert** в компоненте **TDBGrid** достаточно нажать на клавиатуре клавишу **Insert** или, находясь на последней записи набора данных, попытаться перейти на нижнюю, несуществующую запись. То же происходит при нажатии соответствующей кнопки связанного с набором данных компонента **TDBNavigator**.

Метод **Append** аналогичен методу **Insert**, но он добавляет запись в конец набора данных, в то время как метод **Insert** добавляет ее после текущей записи. Для индексированных НД применение метода **Append** приводит к тем же последствиям, что и применение метода **Insert**.

Методы

```
procedure InsertRecord(const Values: array of const);
```

procedure AppendRecord(**const** Values: **array of const**);

позволяют выполнять добавление записи в базу данных целиком.

Метод **InsertRecord** объединяет функциональность методов **Insert**, **Post** и действий по присваиванию значений полям новой записи. В функциональном отношении он аналогичен методу **SetFields**.

Метод **AppendRecord** аналогичен методу **InsertRecord** и отличается от него только тем, что помещает новую запись не после текущей записи, а вслед за последней записью набора данных.

3.2.3. Запоминание изменений

Выполнение метода **Post** приводит к запоминанию изменений, сделанных в режиме добавления или изменения записи. Если НД не находится в режиме **dsInsert** или **dsEdit**, то применение метода **Post** приводит к генерации исключительной ситуации.

Вызов метода **Post** зависит от способа, которым ранее были вызваны методы **Insert** или **Edit**: программно или автоматически.

Метод **Post** обычно вызывается автоматически, если пользователь предпринимает действия по запоминанию измененной записи в НД. Вид этих действий зависит от визуального компонента, связанного с НД. Например, для компонента **TDBGrid**, связанного с набором данных, это переход к другой записи, а для НД, управляемого компонентом **TDBNavigator**, – нажатие соответствующей кнопки компонента **TDBNavigator**. Иногда изменения в НД, автоматически переведенном в режим редактирования, запоминаются путем программного вызова метода **Post**.

Метод **Post** независимо от способа вызова может завершиться неудачно. Причинами этого могут стать неверные значения в соответствующих полях записи, например:

– поле обязательного заполнения (свойство **Required** имеет значение **True** у соответствующего компонента **TField**) может содержать пустое значение;

– для ТБД, для которой определен уникальный ключ, возникла ситуация дублирования ключа (**Key Violation**);

– обработчики событий типа **OnValidate** (компонент **TField**) или **BeforePost** (компонент НД) обнаружили, что какое-либо поле содержит неверное значение, не удовлетворяющее заданным условиям. В этом случае программно вызывается исключительная ситуация, которая прерывает работу метода **Post**.

При отказе выполнения метода **Post** запись может быть переведена в состояние **dsInsert** или **dsEdit**, в котором НД находился до выполнения метода. Программисту также требуется предусмотреть перевод базы данных в состояние **dsBrowse** в аварийных ситуациях.

3.2.4. Отмена сделанных изменений

Метод **Cancel** отменяет все изменения, сделанные в записи. Если НД находился в режиме добавления новой записи, то запись в НД не добавляется. Если же НД был в режиме изменения записи, то эта запись не записывается в НД и данные в ней остаются в том состоянии, в котором они пребывали до перехода в режим **dsEdit**. При этом НД переводится в режим **dsBrowse**.

Вызов метода **Cancel** зависит от способа, которым ранее были вызваны методы **Insert** или **Edit**: программно или автоматически.

Этот метод вызывается автоматически, если пользователь предпримет соответствующие действия, направленные на запоминание измененной записи в НД. Вид этих действий зависит от визуального компонента, связанного в НД. Например, для компонента **TDBGrid** это нажатие клавиши **Esc**, для НД, управляемого компонентом **TDBNavigator**, – нажатие соответствующей кнопки компонента **TDBNavigator**.

3.2.5. Оценка изменений записи и реакция на изменение данных

Часто бывает необходимо знать, вносились ли в запись изменения в режимах **dsInsert** или **dsEdit**. Это актуально в тех случаях, когда внесение изменений в поля записи зависит от каких-либо условий, которые могут наступить или не наступить в разные моменты работы приложения. Свойство НД

```
property Modified: Boolean;
```

автоматически устанавливается в значение **True**, если какое-либо поле записи НД было изменено в режимах **dsInsert** или **dsEdit**. Методы **Post** и **Cancel** переводят свойство **Modified** в состояние **False**.

Реакция на изменение данных отслеживается в событиях **OnUpdateData** и **OnValidate**. Событие **OnUpdateData** компонента **TDataSource** возникает для измененной или вновь добавляемой записи, когда выполнен метод **Post**, но физического перезаписывания измененной записи в таблице базы данных еще не произошло. Событие **OnValidate** компонента **TField** происходит после любого изменения значения поля, произведенного вручную или программно (это относится и к вводу значения в поле при создании новой записи).

3.2.6. Удаление записи

Удаление текущей записи в наборе данных реализуется методом **Delete** и может производиться:

– программно;

– автоматически, если это предусмотрено в том или ином компоненте.

Так, в компоненте **TDBGrid** нажатие комбинации клавиш **Ctrl + Del** влечет за собой удаление записи, которое, в соответствии с опциями настройки компонента **TDBGrid**, может выполняться как с запросом подтверждения, так и без него.

При удалении записи необходимо помнить о том, что записи в различных СУБД могут удаляться двумя способами:

– пометкой записи в таблице базы данных как удаленной, в то время как сама запись физически не удаляется из НД. В зависимости от вида СУБД новые записи могут записываться на место помеченных как удаленные или в конец таблицы базы данных. В последнем случае таблица может увеличиваться до больших размеров, поэтому время от времени для нее нужно проводить операцию сжатия, при которой помеченные в качестве удаленных записи физически уничтожаются, а остальные записи сдвигаются вверх, заполняя образовавшиеся пустоты в таблице базы данных;

– немедленным удалением записей из ТБД, вследствие чего последующие записи сдвигаются вверх, заполняя образовавшиеся в таблице базы данных пустоты.

В среде **Delphi** при работе с НД реализован второй способ, при котором после удаления записи все оставшиеся записи сдвигаются вверх. При удалении одной записи это может быть несущественным, однако если нужно удалить несколько записей, то это может привести к определенным осложнениям.

Например, пусть требуется удалить все записи в НД **ADOTable1**. Для выполнения этого действия мы могли бы использовать следующий программный код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ADOTable1 do
    while not EOF do
      begin
```

```
        Delete;
        Next;       // Ошибка!
      end;  // While
end;
```

Однако в действительности этот код приведет к удалению примерно половины записей в НД **ADOTable1**. Причина этого состоит в том, что когда мы удаляем запись (например, № 3), то последующие записи автоматически перемещаются вверх и поэтому запись, бывшая до удаления следующей (№ 4), становится текущей (№ 3). После выполнения метода **Next** осуществляется переход к записи № 4. Таким образом, записи удаляются через одну.

Эту проблему можно решить, удалив ненужный вызов метода **Next** и используя встроенную переменную **RecordCount**:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ADOTable1 do
    begin
      First;
      while not(RecordCount = 0) do
        Delete;
    end; // With
end;
```

Чаще всего нужно удалять не все записи НД, а часть записей, удовлетворяющих некоторому условию. Для этого обычно используются два способа: временная фильтрация удаляемых записей в момент группового удаления или метод **Locate**, реализующий поиск необходимой записи по точному соответствию значений некоторых полей.

Для больших НД, а также если в процессе удаления в НД происходит изменение индексов, определяющих текущую сортировку, рекомендуется перед удалением отключать связь НД с управляющими компонентами, а по окончании удаления – включать.

3.3. Закладки на записях набора данных

Подобно тому, как в книге нужную страницу можно заложить закладкой и впоследствии быстро найти, в НД аналогичные действия можно осуществить и для записи. Для этой цели набор данных использует следующие методы:

– **function** GetBookmark: TBookmark;

который создает для текущей записи объект-закладку и возвращает ссылку на него;

– **procedure** GotoBookmark(Bookmark: TBookmark);

который перемещает курсор БД на запись, определяемую закладкой-параметром. Этот метод вызывает исключительную ситуацию, если передаваемый параметр не указывает на закладку, т. е. если закладка не создана, а мы пытаемся на нее перейти;

– **function** BookmarkValid(Bookmark: TBookmark): Boolean;

который возвращает значение **True**, если закладка **Bookmark** имеет значение, и значение **False** в противном случае;

– **function** CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;

который сравнивает закладки **Bookmark1** и **Bookmark2** и возвращает значение 0, если закладки идентичны, и значение 1, если они различаются. Тип **TBookmark** является указателем на экземпляр типа «закладка».

Приведем программный код, использующий упомянутые выше методы. Для работы с закладками необходимо объявить глобальную переменную, для чего в блоке **var** раздела **interface** добавим код

```
MyBookmark: TBookmark;
```

В этом случае блок глобальных переменных будет выглядеть следующим образом:

```
var
  Form1: TForm1;
  MyBookmark: TBookmark;
```

Расположим на форме **Form1** три кнопки типа **TButton**: **Создать закладку**, **Перейти на закладку**, **Освободить закладку** – и напишем для них обработчики события **OnClick**:

```
// Создать закладку
procedure TForm1.GetBookmarkButtonClick(Sender: TObject);
```

```
begin
  MyBookmark := DataModule2.Q_IncomingGoodsD.GetBookmark;
end;
// Перейти на запись, на которой заложена закладка
procedure TForm1.GotoBookmarkButtonClick(Sender: TObject);
begin
  if DataModule2.Q_IncomingGoodsD.BookmarkValid(MyBookmark)
then
    DataModule2.Q_IncomingGoodsD.GotoBookmark(MyBookmark);
end;
// Освободить ресурсы, выделенные для закладки
procedure TForm1.FreeBookmarkButtonClick(Sender: TObject);
begin
  if DataModule2.Q_IncomingGoodsD.BookmarkValid(MyBookmark)
then
    MyBookmark := nil;
end;
```

Таким образом, закладки используются когда, во-первых, необходимо проделать какие-либо действия над набором данных, изменяющие местоположение курсора БД, и, во-вторых, вернуться к той записи, которая была текущей до выполнения действий над набором данных.

3.4. Сценарий обновления записей на одной форме с компонентом TDBGrid

Данные в БД периодически добавляются, изменяются и удаляются. Если выполнение указанных действий ложится на пользователя, то в приложении должен быть реализован удобный интерфейс добавления, изменения и удаления записей.

Существует несколько вариантов внесения изменений в набор данных в приложении.

При первом варианте изменения реализуются при выполнении программного кода. Этот вариант, как правило, используется для

транзакционных таблиц, т. е. таблиц, данные в которых формируются приложением автоматически.

При другом варианте изменения в НД вносит пользователь. Реализовать пользовательский интерфейс для компонентов **TADOTable** в этом случае можно одним из следующих способов:

– пользователь вносит изменения в НД через компонент **TDBGrid**. Для добавления новой записи в НД он нажимает на клавиатуре кнопку **Insert**, после чего в компоненте **TDBGrid** появляется новая, пустая строка; вводит данные в поля новой строки; запоминает запись, перемещая курсор на другую запись, или отказывается от запоминания, нажимая на клавиатуре клавишу **Esc**;

– пользователь нажимает одну из трех кнопок: **Добавить**, **Изменить**, **Удалить**, размещенных в той же форме, что и компонент **TDBGrid**, в котором показываются записи НД. По нажатию кнопки набор данных переводится в соответствующее состояние и вызывается другая форма, куда пользователь вводит новые значения полей записи и нажимает кнопку **Запомнить**, **Отменить** или **Удалить**. После этого выполняются методы **Post**, **Cancel** или **Delete**;

– пользователь нажимает одну из трех кнопок: **Добавить**, **Изменить**, **Удалить**, размещенных в той же форме, что и компонент **TDBGrid**, в котором показываются записи НД. После нажатия кнопки набор данных переводится в соответствующее состояние и на этой же форме активизируется панель, на которой расположены поля текущей записи набора данных. Пользователь вводит новые значения полей записи, после чего подтверждает или отменяет добавление или изменение записи. При удалении записи ему нужно лишь подтвердить или отменить удаление.

Поскольку **TADOQuery** не предусматривает изменения набора данных при работе с компонентом **TDBGrid**, то для компонентов **TADOQuery** могут быть использованы только последние два способа организации интерфейса. При этом возможны две стратегии обновления данных: первая заключается в применении **SQL**-запросов для обновления и/или добавления записи, вторая – в обращении к компоненту **TADOTable** для работы с записью. Вторая стратегия является более предпочтительной благодаря тому, что в ней могут быть реализованы различные механизмы наложения ограничений и проверки введенных значений на удовлетворение заданным условиям.

Рассмотрим реализацию сценария, в котором для отображения используется компонент **TADOQuery**, а для работы с записями НД – компонент **TADOTable**. При этом визуальная часть будет основана на

использования панели редактирования данных с использованием событий компонентов.

За основу возьмем главную форму (**Form1**), на которой будет происходить редактирование таблицы базы данных **IncomingGoods** (**T_IncomingGoods**). На форме **Form1** для показа записей из этой таблицы используется компонент **TDBGrid**, причем для отображения берется результат запроса **Q_IncomingGoodsD**, в котором присутствуют поля выбора данных и вычисляемое поле, а также элементы для работы с закладками (кнопки) и область вывода информации из поля выбора (компоненты **TLabel** и **TDBText**).

Изменим имеющийся интерфейс следующим образом. Создадим область отображения набора данных, которая будет подстраиваться под размеры окна. Для этого разместим на форме компонент **TPanel** с именем **Panel1**, затем еще один экземпляр компонента **TPanel** с именем **Panel2** и установим его свойство **Align** в значение **alBottom** (выравнивание компонента по всей нижней части контейнера, в качестве которого выступает **Panel1**). Установим на **Panel2** кнопки, отвечающие за работу с закладками и компоненты вывода данных из поля выбора. Для этого в дереве проекта в области **Structure** выделим нужные элементы (рис. 3.2), перенесем их на надпись **Panel2** и настроим их визуальное отображение (размещение). Затем переместим компонент **DBGrid1** на **Panel1** и зададим его свойству **Align** значение **alClient** (выравнивание компонента по всей клиентской области контейнера, т. е. по оставшемуся пространству).

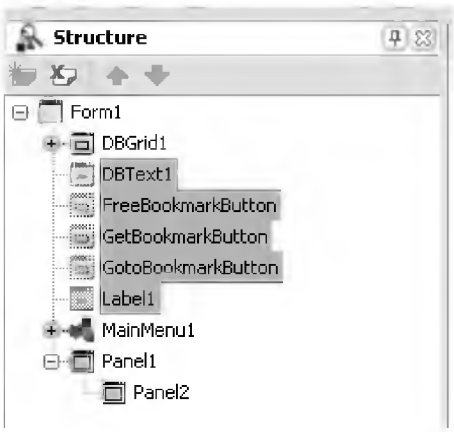


Рис. 3.2. Размещение компонентов
с использованием компонента **TPanel**

Поместим на форме **Form1** панель **InsertEditDeletePanel** (компонент **TPanel**) и зададим ее свойству **Visible** значение **True**, а свойство **Align** – в значение **alBottom**. На этой панели установим

кнопки для работы с отдельной записью: **Вставить** (**InsertButton**), **Изменить** (**EditButton**), **Удалить** (**DeleteButton**).

Разместим на форме еще одну панель **PanelToInputValues** (компонент **TPanel**) и определим ее как невидимую (свойство **Visible** имеет значение **False**), свойству **Align** присвоим значение **alBottom**. На этой панели расположим компоненты **DBLookupComboBox1**, **DBEdit1**, **DBLookupComboBox2**, которые будут ссылаться на соответствующие поля. Для удобства выбора даты можно также использовать компонент **DateTimePicker1**. Для сохранения или отказа от сделанных изменений в таблице поместим на панели кнопки **Запомнить** (**PostButton**) и **Отменить** (**CancelButton**).

Панель **PanelToInputValues** невидима, как и компоненты, расположенные в ней. Для настройки внешнего отображения у компонентов этой панели можно использовать параметры:

– **AlignWithMargins** – выполнение выравнивания компонента с учетом отступов в случае значения **True**;

– **Margins** – значение отступов для каждой стороны;

– **BevelInner** – тип внутренней рамки;

– **BevelOuter** – тип внешней рамки;

– **BorderStyle** – стиль обрамления;

– **BevelWidth** – толщина рамки в пикселях;

– **Caption** – надпись, отображаемая на панели, которую лучше делать пустой.

Чтобы пользователь программы мог лучше ориентироваться в том, что он редактирует, целесообразно установить компоненты **TLabel**, в свойстве **Caption** которых нужно задать поясняющий текст.

Настроим компоненты для связи с редактируемыми полями НД. Установим свойства для компонента **DBLookupComboBox1**:

– **DataSource** – в значение **DataModule2.DS_T_IncomingGoods**;

– **DataField** – в значение **IDgc**;

– **ListSource** – в значение **DataModule2.DS_T_GoodsCatalog**;

– **ListField** – в значение **gcName**;

– **KeyField** – в значение **Idgc**;

для компонента **DBEdit1**:

– **DataSource** – в значение **DataModule2.DS_T_IncomingGoods**;

– **DataField** – в значение **igAmount**;

для компонента **DBLookupComboBox2**:

– **DataSource** – в значение **DataModule2.DS_T_IncomingGoods**;

– **DataField** – в значение **IDsc**;

– **ListSource** – в значение **DataModule2.DS_T_SuppliersCatalog**;

– **ListField** – в значение **scName**;

– **KeyField** – в значение **IDsc**.

Если мы хотим, чтобы в базе данных сохранялась только дата, то нам необходимо присвоить свойству **Time** компонента **DateTimePicker1** пустое значение, которое автоматически преобразуется в **0:00:00**.

Зададим свойству **Align** у компонента **Panel1** значение **alClient**. После размещения и настройки компонентов главная форма приложения должна иметь вид, представленный на рис. 3.3.

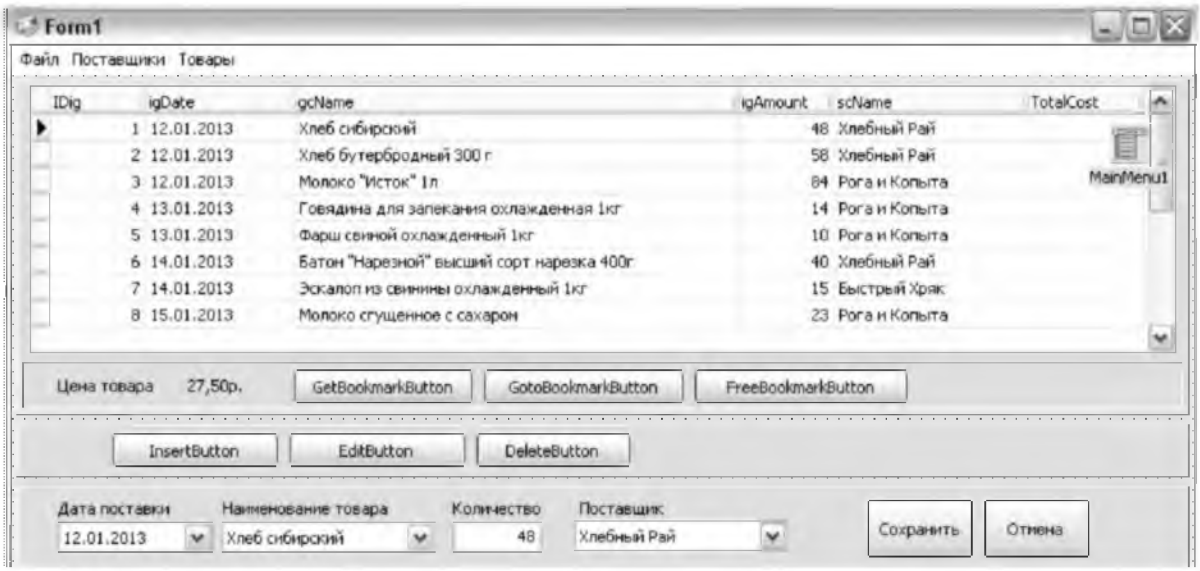


Рис. 3.3. Использование визуальных компонентов при работе с данными БД

Теперь напишем необходимые обработчики для НД и кнопок. Поскольку у нас отображается результат запроса, то нам необходимо сделать так, чтобы записи, выбранной в компоненте **DBGrid1**, соответствовала запись в компоненте **T_IncomingGoods**, а также чтобы после каждого обновления данных в таблице базы данных обновлялись данные запроса в компоненте **Q_IncomingGoodsD**:

```
// Объявить переменную закладки в блоке var у DataModule2
  PrihodBookmark:TBookmark;


// Обработчик синхронизации выбранных записей
procedure TDataModule2.Q_IncomingGoodsDAfterScroll(DataSet:
  TDataSet);
begin
// Проверить количество выводимых записей
  if Q_IncomingGoodsD.RecordCount > 0 then
```

```
// Если это количество больше нуля, то выполнить поиск
// соответствующей записи в компоненте T_IncomingGoods
  T_IncomingGoods.Locate('IDig', Q_IncomingGoodsDIDig.Value, [ ])
end;

// Обработчик действий после сохранения записи в таблице
procedure TDataModule2.T_IncomingGoodsAfterPost (DataSet:
  TDataSet);
begin
// Обновить НД в компоненте Q_IncomingGoodsD
  Q_IncomingGoodsD.Active := false;
  Q_IncomingGoodsD.Active := true;
// Проверить, существует ли закладка
  if Q_IncomingGoodsD.BookmarkValid(PrihodBookmark) then
    begin
// Перейти на закладку, если она существует
    Q_IncomingGoodsD.GotoBookmark(PrihodBookmark);
// Удалить закладку
    PrihodBookmark := nil;
  end;
end;

// Обработчик действий после удаления записи из таблицы
procedure TDataModule2.T_IncomingGoodsAfterDelete(DataSet:
  TDataSet);
begin
// Обновить НД в Q_IncomingGoodsD
  Q_IncomingGoodsD.Active := false;
```

```
  Q_IncomingGoodsD.Active := true;
end;

// Обработчик действий события OnChange поля T_igAmount
procedure TDataModule2.T_IncomingGoodsigAmountChange(Sender:
   TField);
begin
  if T_IncomingGoodsigAmount.Value <= 0 then
    raise Exception.Create('Количество должно быть больше нуля');
end;
```

После этого составим код обработчиков кнопок, расположенных на форме **Form1**:

```
// Обработчик нажатия кнопки Вставить
procedure TForm1.InsertButtonClick(Sender: TObject);
begin
// Отключить возможность изменения курсора НД
// в компоненте DBGrid1
  DBGrid1.Enabled:=False;
// Сделать невидимой панель с кнопками Вставить, Изменить,
// Удалить
  InsertEditDeletePanel.Visible := False;
// Сделать видимой панель для ввода значения в запись
  PanelToInputValues.Visible := True;
// Перевести НД в режим добавления записи, используя
// метод Append
  DataModule2.T_IncomingGoods.Append;
// Установить компоненту выбора даты текущую дату
  DateTimePicker1.Date := Now;
// Передать фокус управления для корректировки даты
```

```
    DateTimePicker1.SetFocus;
  end;

  // Обработчик нажатия кнопки Изменить
  procedure TForm1.EditButtonClick(Sender: TObject);
  begin
  // Создать закладку в компоненте Q_IncomingGoodsD
  // для редактируемой записи
    PrihodBookmark := DataModule2.Q_IncomingGoodsD.GetBookmark;
  // Отключить возможность изменения курсора НД
  // в компоненте DBGrid1
    DBGrid1.Enabled := False;
  // Сделать невидимой панель с кнопками Вставить, Изменить,
  // Удалить
    InsertEditDeletePanel.Visible := False;
  // Сделать видимой панель для ввода значения в запись
    PanelToInputValues.Visible := True;
  // Перевести НД в режим редактирования записи
    DataModule2.T_IncomingGoods.Edit;
  // Установить компоненту выбора даты дату, хранимую
  // в поле записи
    DateTimePicker1.Date := DataModule2.T_IncomingGoodsigDate.-
Value;
  // Передать фокус управления для корректировки даты
    DateTimePicker1.SetFocus;
  end;

  // Обработчик нажатия кнопки Удалить
  procedure TForm1.DeleteButtonClick(Sender: TObject);
```

```
begin
// Выдать запрос на подтверждение удаления записи
  if MessageDlg('Подтвердите удаление записи', mtConfirmation,
     [mbYes, mbNo], 0) = mrYes then
// Удалить запись в случае подтверждения пользователем
     DataModule2.T_IncomingGoods.Delete;
// Передать управление на компонент DBGrid1
  DBGrid1.SetFocus;
end;

// Обработчик нажатия кнопки Запомнить
procedure TForm1.PostButtonClick(Sender: TObject);
begin
// Назначить полю значения даты, выбранной в DateTimePicker1
  DataModule2.T_IncomingGoodsigDate.Value :=
DateTimePicker1.Date;
// Попытаться сохранить запись
  try
    DataModule2.T_IncomingGoods.Post;
  finally
// Сделать невидимой панель для ввода значений в запись
    PanelToInputValues.Visible := False;
// Визуализировать панель с кнопками Вставить, Изменить,
// Удалить
    InsertEditDeletePanel.Visible := True;
// Активировать возможность изменения курсора
// в компоненте  DBGrid1
    DBGrid1.Enabled := True;
```

```
// Передать управление на компонент DBGrid1
    DBGrid1.SetFocus;
  end; // finally
end;

// Обработчик нажатия кнопки Отменить
procedure TForm1.CancelButtonClick(Sender: TObject);
begin
// Перевести НД в состояние dsBrowse путем отмены
  DataModule2.T_IncomingGoods.Cancel;
// Сделать невидимой панель для ввода значений в запись
  PanelToInputValues.Visible := False;
// Сделать видимой панель с кнопками Вставить, Изменить,
// Удалить
InsertEditDeletePanel.Visible := True;
// Активировать возможность изменения курсора
// в компоненте DBGrid1
  DBGrid1.Enabled := True;
// Передать управление на компонент DBGrid1
  DBGrid1.SetFocus;
end;

// Обработчик события выхода из формы
// Если в этот момент НД находится в состоянии, отличном
// от состояния dsBrowse, то обработчик принудительно
// возвращается в это состояние
procedure TForm1.FormDeactivate(Sender: TObject);
begin
```

```
// Проверить, находится ли НД в состоянии dsBrowse
  if DataModule2.T_IncomingGoods.State <> dsBrowse then
// В случае если НД находится в состоянии, отличном от dsBrowse,
// вызвать обработчик событий кнопки отмены
    CancelButtonClick(nil);
end;
```

При нажатии кнопки **Удалить** выдается модальное окно подтверждения удаления, реализуемое функцией **MessageDlg**.

3.5. Синхронизация содержимого наборов данных

Подходы к синхронизации содержимого в НД различаются в зависимости от того, обновляются ли они в одном приложении или в нескольких. Рассмотрим, как осуществляется синхронизация НД в одном приложении.

В приложении может существовать несколько НД, ассоциированных с одной и той же таблицей БД. Например, это может быть компонент **ADOTable1**, расположенный в модуле данных **TDataModule** приложения, и компонент **ADOTable2**, расположенный на форме **Form1** и выполняющий там специфические функции, отличные от функций компонента **ADOTable1**. Тогда если эти компоненты активны во время выполнения приложения, то необходимо обновлять содержимое одного набора данных в случае обновления другого.

Например, если изменяется НД компонента **DataModule1.ADOTable1**, то для синхронизации изменения с содержимым НД компонента **Form1.ADOTable2** следует задать следующие обработчики событий:

```
procedure TDataModule1.ADOTable1AfterDelete(DataSet: TDataSet);
begin
  TForm1.ADOTable2.Refresh;
end;

procedure TDataModule1.ADOTable1AfterPost(DataSet: TDataSet);
```

```
begin
  TForm1.ADOTable2.Refresh;
end;
```

Если НД находится в состоянии **dsBrowse**, то его обновление можно реализовать периодически, например с помощью таймера:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if ADOTable2.State = dsBrowse then
    ADOTable2.Refresh;
end;
```

Если нужно обновить все НД приложения, то для события таймера необходимо написать следующий код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  i: Integer;
begin
with ADOConnection1 do
  begin
// Для каждого НД выяснить количество НД с помощью
// свойства DataSetCount
  for i := 0 to DataSetCount – 1 do
// Проверить, находится ли НД в состоянии dsBrowse
    if DataSets[i].State = dsBrowse then
 // Обновить НД
      DataSets[i].Refresh;
  end;        // With
end;
```

Если требуется обновлять лишь некоторые из наборов данных, то указатели на них можно поместить в список **StringList1**, который

создается и наполняется в момент создания формы, а при разрушении формы удаляется.

3.6. Конфигурация приложения для работы с базой данных

Довольно часто возникает ситуация, когда разрабатываемое приложение с использованием базы данных на основе СУБД **Microsoft Access**, не работает после переноса на другой компьютер. Это может быть вызвано тем, что при организации подключения (формирование **ConnectionString**) мы указываем конкретный путь к базе данных, а на другом компьютере может не быть ни такого пути, ни самого файла БД.

Для того чтобы избежать проблем с переносом приложения на другой компьютер, необходимо сделать так, чтобы при запуске программы происходило динамическое подключение к базе данных. Это можно сделать, записав следующий код на событие **OnCreate** у контейнера **TDataModule** (в данной лабораторной работе имя контейнера **DataModule2**):

```
procedure TDataModule2.DataModuleCreate(Sender: TObject);
var
  i: Integer;
begin
// Настроить строку соединения с базой данных MyDatabase.accdb
ADOConnection1.ConnectionString := 'Provider =
Microsoft.ACE.OLEDB.12.0; Data Source =' +
ExtractFilePath(Application.ExeName) +  '\MyDatabase.accdb;
Persist Security Info = False';
// Выполнить соединение
  ADOConnection1.Connected := true;
  with ADOConnection1 do
    for i := 0 to DataSetCount - 1 do
// Свойство DataSetCount означает  количество НД
```

```
// Для каждого компонента НД выполнить открытие НД
DataSets[i].Active := true
end;
```

Команда **ExtractFilePath(Application.ExeName)** позволяет узнать место расположения исполняемого файла программы, на основе которого можно сформировать нужный путь для подключения файла базы данных. Для выполнения этой команды необходимо в блоке **uses** раздела **interface** указать модули **SysUtils** и **Forms**.

Перед компиляцией проекта свойство **Connected** компонента **TADOConnection** (в нашем случае **ADOConnection1**) нужно установить в значение **False**.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Для выполнения работы воспользоваться копией приложения, разработанного в лабораторной работе 2.
2. Ознакомиться с процедурами навигации по набору данных. Добавить возможность навигации по набору данных с использованием компонентов **TButton** (см. п. 3.1).
3. Изменить сортировку для набора данных поставщиков (см. п. 3.1.1).
4. Ознакомиться со способом изменения состояний набора данных (см. п. 3.1.2).
5. Реализовать метод **SetFields** для НД **Тип товаров** (см. п. 3.2.1).
6. Ознакомиться со способами вставки строк в набор данных и применить метод **AppendRecord** для НД **Поставщики** (см. п. 3.2.2).
7. Рассмотреть методы **Post**, **Cancel** и **Delete** (см. п. 3.2.3–3.2.6).
8. Реализовать использование закладок на записях набора данных прихода товаров (см. п. 3.1.3).
9. Реализовать функционал модификации набора данных прихода товаров на основе сценария, предложенного в п. 3.4.
10. Ознакомиться с методами синхронизации наборов данных (см. п. 3.5).
11. Настроить конфигурацию приложения для работы с базой данных, хранящейся в каталоге приложения (см. п. 3.6).
12. Ответить на контрольные вопросы.

13. Составить отчет, который должен содержать титульный лист, цель и ход выполнения лабораторной работы, сформулированные выводы.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Что понимается под набором данных в среде **Delphi**?
2. Какие компоненты отвечают за работу с НД? Сравните их.
3. Назовите способы работы с записями набора данных.
4. Дайте определение понятия «курсор НД».
5. Какие методы для изменений курсора набора данных существуют?
6. Каким образом определяются начало и конец набора данных?
7. Поясните соотношение порядка следования и порядка сортировки записей.
8. Каким образом осуществляется навигация по набору данных вверх и вниз?
9. Каким образом выполняются спонтанные перемещения по НД?
10. Опишите реализацию способа перебора записей при отключенной сортировке по исходному индексу или при сортировке по другому индексу.
11. Каким образом реализуется способ временной фильтрации данных?
12. Какие события возникают при изменении курсора набора данных?
13. В каких случаях необходимо временное отключение визуализации НД?
14. Каким образом выполняется временное отключение визуализации данных?
15. Какое свойство разрешает или запрещает изменять записи в НД? Охарактеризуйте его.
16. Какое свойство определяет, возможен ли автоматический перевод НД в состояние **dsEdit**? Дайте его характеристику.
17. Каким образом можно изменить текущую запись?
18. Охарактеризуйте метод **Edit**.
19. Укажите характеристики метода **SetFields**.
20. Каким образом происходит добавление новой записи?
21. Дайте характеристику метода **Insert**.

22. Опишите метод **Append**.

23. Каким образом работает метод **InsertRecord**?

24. Охарактеризуйте метод **AppendRecord**.

25. Каким образом происходит запоминание записей?

26. В каких случаях запоминание записей может завершиться неудачно?

27. С помощью каких действий выполняется отмена сделанных изменений в НД?

28. Каким образом производится оценка изменения записи?

29. Дайте характеристику события **OnUpdateData**.

30. Опишите событие **OnValidate**.

31. Каким образом происходит удаление записи?

32. Какие способы удаления записей в различных СУБД вам известны?

33. Составьте программный код для удаления всех записей из таблицы базы данных.

34. Предложите способы удаления некоторых записей из таблицы базы данных.

35. Приведите пример удаления части записей методом **Locate**.

36. Какие варианты внесения изменений в НД в приложении существуют?

37. Назовите методы реализации интерфейса при внесении изменений в набор данных.

38. В каких случаях используются закладки на записях в НД?

39. Охарактеризуйте функции и процедуры для работы с закладками.

Лабораторная работа 4

ФИЛЬТРАЦИЯ И ПОИСК ЗАПИСЕЙ В НАБОРАХ ДАННЫХ. ВИЗУАЛЬНЫЕ СВОЙСТВА И СОБЫТИЯ TDBGrid

Цель работы: ознакомиться со свойствами, методами и событиями для фильтрации записей; ознакомиться со средствами поиска записей в наборах данных; разработать приложение для фильтрации записей, а также для их поиска в наборах данных; разработать приложения с возможностью изменения визуальных свойств **TDBGrid** и организацией динамической сортировки набора данных.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Рассмотрим порядок реализации таких неотъемлемых функций базы данных, как фильтрация и поиск записей. Какой метод применить в том или ином случае, выбирает сам программист. Однако существуют некоторые практические рекомендации для решения этих вопросов. Также в данной лабораторной работе приведены полезные свойства компонента **TDBGrid**, позволяющие организовать удобный интерфейс работы с данными.

4.1. ФИЛЬТРАЦИЯ ЗАПИСЕЙ В НАБОРАХ ДАННЫХ

Стандартными средствами в компонентах **TADOTable** и **TADOQuery** для организации фильтрации набора данных являются свойства **Filtered**, **Filter**, **FilterOptions**, событие **OnFilterRecord**. Помимо этого для фильтрации может использоваться предложение **WHERE** оператора **SELECT**, указанное в свойстве **SQL** компонента **TADOQuery**[*].

Рассмотрим средства для организации фильтрации НД и особенности реализации фильтрации по текстовым полям и полям типа **TDateField**, **TDateTimeField**.

4.1.1. Свойство Filtered

[*] Более подробно об этом см.: Фленов М. Библия Delphi. 3-е изд. СПб. : БХВ-Петербург, 2011.

Свойство

```
property Filtered: Boolean;
```

установленное в значение **True**, инициирует фильтрацию, условие которой содержится как строковое значение в свойстве **Filter** или записано в обработчике события **OnFilterRecord**. Если в событии **OnFilterRecord** и свойстве **Filter** установлены разные условия фильтрации, то выполняются оба условия. Например, если в НД одновременно установлены следующие фильтры:

```
ADOTable1.Filter = '[GoodsCost] > 50 ';


procedure TForm1.ADOTable1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
   Accept := DataSet['GoodsType'] = 'Молочные продукты';
end;
```

то задание свойству **ADOTable1.Filtered** значения **True** приведет к двум фильтрациям: в результирующем НД будут показаны только те записи, у которых поле **GoodsCost** содержит значения больше **50**, а поле **GoodsAmount** – значение **Молочные продукты**.

Установка свойства **Filtered** в значение **False** приведет к отмене фильтрации, условия которой указаны в событии **OnFilterRecord** или/и свойстве **Filter**.

4.1.2. Событие OnFilterRecord

Событие

```
property OnFilterRecord: TFilterRecordEvent;
```

возникает, когда свойство **Filtered** установено в значение **True**.

Обработчик события **OnFilterRecord** имеет два параметра: имя фильтруемого набора данных и переменную **var Accept**, указывающую условия фильтрации записей в НД. В отфильтрованный НД включаются только те записи, для которых параметр **Accept** имеет значение **True**.

В условиях фильтрации могут участвовать любые поля НД, в том числе и не входящие ни в один индекс. Такой способ фильтрации позволяет реализовать сложные логические конструкции условий фильтрации.

Однако следует помнить о том, что при указании условий фильтрации набора данных в обработчике **OnFilterRecord** в нем последовательно перебираются все записи таблицы базы данных при их анализе на соответствие условию фильтрации. Это делает использование события **OnFilterRecord** предпочтительным для небольших объемов записей и значительно ограничивает его применение при больших объемах данных.

Всякий раз при обработке события **OnFilterRecord** НД из состояния **dsBrowse** переводится в состояние **dsFilter**, что предотвращает модификацию НД во время фильтрации. После завершения текущего вызова обработчика события **OnFilterRecord** набор данных переводится в состояние **dsBrowse**.

Для примера проведем фильтрацию НД **Q_GoodsCatalogD** по стоимости товара, которая должна превышать значение, указанное пользователем в фильтре. Для реализации интерфейса фильтра установим на форме **Form4** компонент **TGroupBox**, который позволяет группировать визуальные компоненты, как и компонент **TPanel**, но обладает возможностью задания заголовка, расположенного в верхней части компонента. Разместим на нем компоненты **TEdit** с именем **E_MinCost** и **TCheckBox**. Изменим свойства у компонентов (рис. 4.1) и запишем код обработчиков событий, которые при изменении значения (**E_MinCost.Text**) обновляли бы фильтр НД:

```
// Обработчик события OnClick компонента CheckBox1
procedure TForm4.CheckBox1Click(Sender: TObject);
begin
// Отключить наложенный фильтр
  DataModule2.Q_GoodsCatalogD.Filtered := false;
// Задать параметр фильтрации в зависимости от состояния
// свойства Checked компонента CheckBox1
  DataModule2.Q_GoodsCatalogD.Filtered := CheckBox1.Checked;
end;
```

```
// Обработчик события OnChange компонента E_MinCost
procedure TForm4.E_MinCostChange(Sender: TObject);
begin
// Отключить наложенный фильтр
  DataModule2.Q_GoodsCatalogD.Filtered := false;
// Проверить длину текста в свойстве Text компонента E_MinCost
  if Length(E_MinCost.Text) > 0 then
// Задать параметр фильтрации в зависимости от состояния
// свойства Checked компонента CheckBox1
    DataModule2.Q_GoodsCatalogD.Filtered := CheckBox1.Checked;
end;

// Обработчик события OnFilterRecord
// компонента НД Q_GoodsCatalogD
procedure TDataModule2.Q_GoodsCatalogDFilterRecord(DataSet:
  TDataSet; var Accept: Boolean);
begin
// Проверить, существует ли форма Form4
  if Assigned(Form4) then
// Если эта форма существует, то назначить отклик для записи
// согласно заданному условию
    Accept := DataSet['gcCost'] > (strtofloat(Form4.E_MinCost.Text))
  else
// Если форма не создана, то вывести запись
  Accept := true;
end;
```

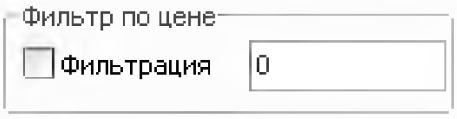


Рис. 4.1. Пример настройки
визуальных свойств компонентов
для фильтрации НД

Выражение **Assigned(Form4)** выступает в качестве элемента проверки существования объекта. Такая проверка необходима, поскольку при запуске программы не все объекты создаются одновременно. В нашем примере форма **Form4** была создана после контейнера **DataModule2**. И если мы не проведем проверку и попытаемся обратиться к еще не созданному объекту, то это приведет к ошибке.

4.1.3. Свойство Filter

Свойство

```
property Filter: String;
```

позволяет указать условия фильтрации. В этом случае НД будет отфильтрован, как только его свойство **Filtered** станет равным значению **True**. Синтаксис фильтра похож на синтаксис предложения **WHERE SQL**-оператора **SELECT** с тем исключением, что в фильтре указываются не имена переменных программы, а имена полей и литералы, т. е. явно заданные значения.

При составлении строки фильтрации можно применять следующие операторы отношения:

– < – меньше чем;
– > – больше чем;
– >= – больше или равно;
– <= – меньше или равно;
– = – равно;
– <> – не равно,

а также логические операторы **and**, **not** и **or**:

```
([GoodsType] = 'Молочные продукты') and ([GoodsCost] < 40)
```

Строку фильтрации можно ввести во время выполнения приложения.

В качестве примера рассмотрим общий код обработчика события, который будет применять фильтр в зависимости от состояния компонента **CheckBox1**:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
```

```
    ADOTable1.Filtered := False;
    ADOTable1.Filter := Edit1.Text;
    ADOTable1.Filtered := CheckBox1.Checked;
  end;
```

В этом примере фильтрация включается при проставлении отметки в поле компонента **CheckBox1** (**CheckBox1.Checked** имеет значение **True**), а когда пользователь снимает отметку (**CheckBox1.Checked** установлен в значение **False**), то фильтрация выключается. Однако при этом нужно следить, чтобы введенная строка соответствовала требованиям, предъявляемым к синтаксису строки **Filter**.

Другим способом реализации фильтра является обработчик, считывающий значения фильтрации и преобразующий их к формату строки **Filter**. Например, может быть создана форма, где значения, по которым осуществляется фильтрация, вводятся в поля компонентов **Edit1** и **Edit2**. После этого приложение автоматически формирует строку условия фильтрации и заносит ее в свойство **Filter**. Эта строка для наглядности показывается в форме приложения в компоненте **Label1**.

4.1.4. Особенности фильтрации по текстовым полям

При организации фильтрации по текстовым полям могут быть использованы различные методы.

Так, если мы хотим найти товар, в наименовании которого содержится некоторый фрагмент строки **STRValue** для события **OnFilterRecord**, то в обработчике необходимо записать следующий код:

```
Accept := Pos(STRValue, DataSet['GoodsType']) > 0;
```

В этом фрагменте кода используется функция

```
function Pos(const substr: string; const str: string): Integer;
```

которая возвращает значение больше нуля (это позиция начала фрагмента **substr**), если фрагмент **substr** найден в строке **str**.

Если мы хотим задать фильтр на основе свойства **Filter**, который отображал бы записи в соответствии с установленным шаблоном, то необходимо использовать ключевое слово **LIKE**. При этом нужно знать, что символ _ (подчеркивание) заменяет любой одиночный символ, а символ **%** (процент) – любую последовательность из символов.

Так, например, если мы хотим найти все товары, названия которых начинаются с определенного фрагмента **STRValue**, то нам необходимо задать фильтр

```
[GoodsName] like 'STRValue%'
```

В случае динамического назначения фильтра, например при нажатии кнопки, присваивание строки фильтрации свойству **Filter** будет выглядеть следующим образом:

```
ADOTable1.Filter := '[GoodsName] like ' + QuotedStr(STRValue + '%');
```

В этом коде можно заметить функцию

```
function QuotedStr(const S: string): string;
```

которая позволяет заключить указанную строку текста **S** в одинарные кавычки.

Зачастую необходимо сформировать фильтр, который отображал бы все записи, включающие в себя фрагмент **STRValue**. В таком случае фильтр будет следующим:

```
[GoodsName] like '%STRValue%'
```

В обработчике события нажатия кнопки этот фильтр примет вид

```
ADOTable1.Filter := '[GoodsName] like ' + QuotedStr('%' + STRValue + '%');
```

При построении универсального фильтра для поиска фрагмента текста на основе события **OnChange** компонента **TEdit** с именем **Edit1** нам понадобится следующий обработчик:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
// Отключить наложенный фильтр
  ADOTable1.Filtered := False;
// Проверить длину введенного значения для фильтрации
  if length(Edit1.Text) > 0 then
    begin
// Сформировать строку фильтра
ADOTable1.Filter := '[GoodsName] like ' + QuotedStr('%' + Edit1.Text + '%');
```

```
// Активировать фильтрацию
   ADOTable1.Filtered := True;
  end;
end;
```

4.1.5. Особенности фильтрации по полям типа TDateField, TDateTimeField

В зависимости от настроек региональных параметров в операционной системе и версии СУБД **Microsoft Access** могут возникнуть ситуации, когда нам нужно применить фильтр по дате больше указанной:

```
ADOQuery1.Filter := 'myDateField > ' +
  QuotedStr(DateToStr(DateTimePicker1.Date));
```

но он не работает. Проблема здесь может заключаться в способе представления даты, который может отличаться в зависимости от версии СУБД и региональных настроек. Для решения этой проблемы можно воспользоваться функцией

```
function FormatDateTime(const Format: string; DateTime:
  TDateTime): string;
```

позволяющей задать формат, в который будет преобразована дата. Спецификаторы формата полностью аналогичны тем, что представлены в табл. 2.3 (см. лабораторную работу 2). В случае явного приведения формата мы будем иметь следующий код фрагмента присвоения:

```
ADOQuery1.Filter := 'myDateField > ' +
  QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker1.Date));
```

Используя такой механизм, мы можем спроектировать фильтрацию, позволяющую проверять вхождение даты в определенный диапазон.

В качестве примера разберем тип фильтрации для поля **myDateField**, в котором содержится дата поступления товара. Нам необходимо разместить следующие компоненты (рис. 4.2):

– **TCheckBox** (**CheckBox1**), который отвечает за включение фильтрации;

– **TComboBox** (**ComboBox1**), который задает тип фильтрации (условия);

– **TDateTimePicker** (**DateTimePicker1** и **DateTimePicker2**), которые указывают даты выполнения фильтрации.

Рис. 4.2. Пример размещения компонентов

Сначала настроим компоненты.

Компоненту **ComboBox1** для свойства **Items** зададим следующие значения: <, <=, =, >, >=, <>, **В диапазоне**, **Вне диапазона**, которые запишем в каждой строке при редактирования свойства **Items** компонента **TComboBox**. Для свойства **Style** зададим значение **csDropDownList**, говорящее о том, что будет выводиться фиксированный список и мы не можем редактировать значения с клавиатуры во время работы программы. Также зададим свойству **ItemIndex** значение **6** (по умолчанию будет фильтрация по диапазону).

Для компонентов **DateTimePicker1** и **DateTimePicker2** в свойстве **Time** выставим значения **0:00:00**. В случае если фильтрация проводится по полю **TDateTime** (дата и время), необходимо задать значения **0:00:00** и **23:59:59** для компонентов **DateTimePicker1** и **DateTimePicker2** соответственно.

Теперь можно приступить к написанию кода:

```
// Обработчик события OnChange компонента ComboBox1
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
// Проверить, какой режим фильтрации предполагается
  if (ComboBox1.Text = 'В диапазоне') or
    (ComboBox1.Text = 'Вне диапазона') then
// В случае если режим на основе диапазонов, отобразить
// компонент
```

```
      DateTimePicker2.Visible := true
    else
// В противном случае скрыть компонент
      DateTimePicker2.Visible := false;
// Вызвать обработчик компонента CheckBox1 для обновления
// фильтра
    CheckBox1Click(self);
end;

// Обработчик события OnClick компонента CheckBox1
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
// Отключить наложенный фильтр
    ADOQuery1.Filtered := false;
// Проверить, скрыт ли компонент DateTimePicker2
    if not DateTimePicker2.Visible then
// Если компонент DateTimePicker2 скрыт, то сформировать
// фильтр на основе условий сравнения, которые берутся
// из компонента ComboBox1
      ADOQuery1.Filter := 'myDateField ' + ComboBox1.Text +
      QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker1.Date))
    else
// Если компонент DateTimePicker2 отображается, то
// выполнить дополнительную проверку и задать
// соответствующий тип фильтра
    if (ComboBox1.Text = 'В диапазоне') then
      ADOQuery1.Filter := '(myDateField >= ' +
      QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker1.Date)) +
```

```
        ') and (myDateField <= ' +
        QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker2.Date)) + ')'
      else
      if (ComboBox1.Text = 'Вне диапазона') then
        ADOQuery1.Filter := '(myDateField < ' +
        QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker1.Date)) +
        ') or (myDateField > ' +
        QuotedStr(FormatDateTime('dd/mm/yy', DateTimePicker2.Date)) +
')';
    // Установить активность фильтра в зависимости от состояния
    // компонента CheckBox1
      ADOQuery1.Filtered := CheckBox1.Checked;
    end;
```

4.2. Поиск записей в наборах данных

Поиск записей в наборах данных может осуществляться различными методами: **Locate**, **FindFirst FindLast**, **FindNext**, **FindPrior**, **Lookup**. Рассмотрим эти методы более подробно.

4.2.1. Метод Locate

Метод **Locate** ищет первую запись, удовлетворяющую критерию поиска, и если такая запись найдена, то делает ее текущей. В этом случае в качестве результата возвращается значение **True**. Если поиск был неуспешен, то возвращается значение **False**:

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

Приведем параметры метода **Locate**:

– список **KeyFields** определяет поле или несколько полей, по которым ведется поиск, в виде строкового выражения. В случае нескольких поисковых полей их названия разделяются точкой с запятой;

– критерии поиска задаются в вариантном массиве **KeyValues** так, что **i**-е значение в параметре **KeyValues** ставится в соответствие **i**-му полю в параметре **KeyFields**. В случае поиска по одному полю в массиве **KeyValues** указывается одно значение;

– параметр **Options** позволяет указать необязательные значения режимов поиска: **loCaseInsensitive** – поиск ведется без учета регистра букв, т. е. если в параметре **KeyValues** указано **хлеб**, а в некоторой записи в данном поле встретилось **Хлеб** или **ХЛЕБ**, то запись считается удовлетворяющей условию поиска; **loPartialKey** – запись считается удовлетворяющей условию поиска, если она содержит часть поискового запроса, например удовлетворяющими запросу **Ка** будут признаны записи со значениями в искомом поле **Картошка**, **Капуста** и т. д.

Метод **Locate** производит поиск по любому полю. В случае когда условию поиска удовлетворяет несколько записей, текущей станет логически самая первая из них (в порядке сортировки записей в НД, определяемом текущими настройками, по умолчанию это индекс). При этом поле или поля, по которым производится поиск, могут не только входить в текущий индекс, но и не быть индексными вообще. В случае если поля поиска входят в какой-либо индекс, метод **Locate** использует этот индекс при поиске. Если искомые поля входят в несколько индексов, то сложно предположить, какой из них будет применен. Соответственно трудно предсказать, какая запись, удовлетворяющая критерию поиска, будет сделана текущей записью, особенно в случае если поиск ведется не по текущему индексу.

Осуществим поиск по полям **GoodsType**; **GoodsName** (индексированному и неиндексированному) при различных текущих индексах в НД. Пусть поисковый контекст задается значением **Хлебные изделия**, **Батон** в режиме частичного совпадения значений:

```
procedure TForm1.Locate1ButtonClick(Sender: TObject);
begin
ADOTable1.Locate('GoodsType; GoodsName',
    VarArrayOf(['Хлебные изделия', 'Батон']), [loPartialKey]);
end;
```

При различных текущих индексах в момент выполнения поиска его результаты также могут быть различными.

А теперь проведем поиск по полю **GoodsName**, которое входит в два индекса, при различных текущих индексах в НД. Пусть поис-

ковый контекст задается значением **Ба** при режиме частичного совпадения значений:

```
procedure TForm1.Locate2ButtonClick(Sender: TObject);
begin
  ADOTable1.Locate('GoodsName', 'Ба', [loPartialKey]);
end;
```

В этом случае результат поиска при различных текущих индексах будет одинаков.

Пусть заранее неизвестно, по какому полю необходимо производить поиск. Тогда поместим в форму **Form1** компонент **RadioGroup1**, в котором перечислим поля поиска, для чего воспользуемся свойством **Items**, и компонент **Edit1** для ввода условий поиска. Зададим следующий обработчик для кнопки **LocateButton**:

```
procedure TForm1.Locate3ButtonClick(Sender: TObject);
var
  Pole: ShortString;
begin
  case RadioGroup1.ItemIndex of
    0:   Pole := 'GoodsType';
    1:   Pole := 'GoodsName';
    2:   Pole := 'GoodsMeasure ';
  end;
  if not ADOTable1.Locate(Pole, Edit1.Text, [loCaseInsensitive,
  loPartialKey])
then     ShowMessage('Запись не найдена');
end;
```

Преимуществом такого подхода является то, что вместо выполнения нескольких методов **Locate** для поиска по каждому полю выбирается один метод **Locate** независимо от поля, по которому производится поиск. В место компонента **TRadioGroup** можно использовать компонент **TComboBox** с фиксированным списком.

4.2.2. Методы FindFirst, FindLast, FindNext, FindPrior

Как было рассмотрено в п. 4.1.1, набор данных может быть отфильтрован с использованием свойства **Filtered**. Условие фильтрации задается свойством **Filter** или описывается в обработчике события **OnFilterRecord**. Свойство **Filtered** указывает, выполнять ли фильтрацию (значение **True**) или нет (значение **False**). В этом случае в наборе данных показываются все записи, а не только записи, удовлетворяющие условию фильтрации.

Для НД, в котором определены условия фильтрации, но сама фильтрация в текущий момент не включена, среда **Delphi** предоставляет следующую возможность: в не отфильтрованном в данный момент НД навигация обеспечивается только между теми записями, которые удовлетворяют условию фильтрации в текущий момент, когда свойство **Filtered** имеет значение **False**. Для этой цели используются методы **FindFirst**, **FindLast**, **FindNext**, **FindPrior**.

Условие фильтрации можно сделать совпадающим с условием поиска, указанным в параметре **KeyValues** метода **Locate**. При этом поиск с помощью указанных методов имеет существенное преимущество перед поиском с помощью метода **Locate**: если в методе **Locate** можно указывать только значения, то в условии фильтрации (свойство **Filter**) можно дополнительно задавать и логические условия, например:

```
([GoodsName] like 'Mo%') and ([GoodsCost] > 40)
```

т. е. включать в НД только записи о товарах, названия которых начинаются на **Mo**, при этом не всех, а только тех из них, у которых цена больше **40**.

Методы **FindFirst**, **FindLast**, **FindNext**, **FindPrior**, **Found** выполняют следующие действия (табл. 4.1).

Таблица 4.1

Методы поиска записей в НД

Метод	Выполняемые действия
function FindFisrt: Boolean	Переводит курсор на первую запись, удовлетворяющую фильтру
function FindLast: Boolean	Переводит курсор на последнюю запись, удовлетворяющую фильтру
function FindNext: Boolean	Переводит курсор на следующую запись, удовлетворяющую фильтру

function FindPrior: Boolean	Переводит курсор на предыдущую запись, удовлетворяющую фильтру
property Found: Boolean	Возвращает значение **True**, если последнее обращение к одному из методов **FindFirst**, **FindLast**, **FindNext**, **FindPrior** привело к нахождению нужной записи

В качестве примера предоставим пользователю возможность перемещаться на первую, последнюю, следующую, предыдущую запись, удовлетворяющую условию «Компонент **Edit1** содержит название товара (или его фрагмент)». При этом свойство **Q_GoodsCatalogD.Filtered** установлено в значение **False**, а в свойстве **Q_GoodsCatalogD.Filter** указано условие фильтрации. В наборе данных показываются все записи, и он остается в неотфильтрованном состоянии:

```
// Изменение условия фильтрации
procedure TForm1.Edit1Change(Sender: TObject);
begin
// Проверить длину  фрагмента для поиска
  if length(Edit1.Text) > 0 then
// Если длина  фрагмента для поиска больше нуля, то задать
// фильтр
    DataModule2.Q_IncomingGoodsD.Filter := '[gcName] like ' +
QuotedStr('%' + Edit1.Text + '%');
  else
// В противном случае задать пустой фильтр
  DataModule2.Q_IncomingGoodsD.Filter := '';
end;

// Нажата кнопка Первая
procedure TForm1.FindFirstButtonClick(Sender: TObject);
begin
  Label1.Caption := '  ';
  DataModule2.Q_IncomingGoodsD.FindFirst;
```

```
end;

// Нажата кнопка Последняя
procedure TForm1.FindLastButtonClick(Sender: TObject);
begin
  Label1.Caption := '  ';
  DataModule2.Q_IncomingGoodsD.FindLast;
end;

// Нажата кнопка Следующая
procedure TForm1.FindNextButtonClick(Sender: TObject);
begin
  Label1.Caption := '  ';
  if not DataModule2.Q_IncomingGoodsD.FindNext then
    Label1.Caption := 'Нет такой записи';
end;

// Нажата кнопка Предыдущая
procedure TForm1.FindPriorButtonClick(Sender: TObject);
begin
  Label1.Caption := '  ';
  if not DataModule2.Q_IncomingGoodsD.FindPrior then
    Label1.Caption := 'Нет такой записи';
end;
```

При запуске этой программы из среды разработки **RAD Studio** и использовании предложенного способа поиска при выполнении методов **FindNext** и **FindPrior** возможно появление окна, которое говорит об ошибке (рис. 4.3). Однако этого не стоит бояться, поскольку так срабатывает внутренний обработчик **ADO** компонентов в отладочном

режиме, а при запуске программы в нормальном режиме (вне среды разработки) это окно появляться не будет.

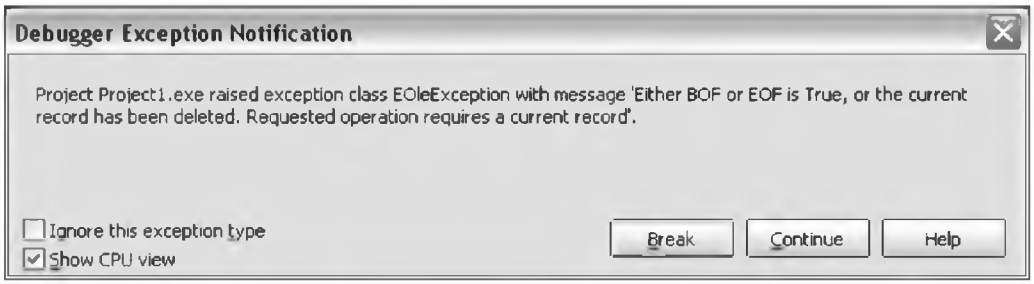


Рис. 4.3. Окно сообщения об ошибке

Заметим, что фильтрация записей с использованием события свойства **Filter** может применяться только на небольших объемах данных (из-за того, что при этом используется последовательный метод доступа к записям в таблице базы данных), поэтому аналогичные ограничения накладываются и на поиск записей с использованием методов **FindFirst**, **FindLast**, **FindNext**, **FindPrior**.

4.2.3. Метод Lookup

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

Метод **Lookup** осуществляет поиск только на точное соответствие критерия поиска и значения полей записи. Такой режим, как **loPartialKey** метода **Locate** (поиск по частичному соответствию значений), в этом методе отсутствует:

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

Метод **Lookup** имеет следующие параметры:

– параметр **KeyFields** указывает список полей, по которым необходимо осуществить поиск. При наличии в этом списке более чем одного поля соседние поля разделяются точкой с запятой;

– параметр **KeyValues** указывает поисковые значения полей, список которых содержится в параметре **KeyFields**. Если имеется несколько поисковых полей, то каждому **i**-му полю в списке

KeyFields ставится в соответствие **i**-е значение в списке **KeyValues**. При наличии одного поля его поисковое значение можно указывать в качестве параметра **KeyValues** непосредственно, в случае нескольких полей их необходимо приводить к типу вариантного массива при помощи функции **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 в результате поиска запись не найдена, то метод **Lookup** возвращает значение **Null**, что выявляется при помощи следующей конструкции:

```
if VarType(LookupResults) = varNull then
{Какие-либо действия}
```

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

– если указано одно поле, то результатом будет значение соответствующего типа или значение **Null**, если поле в найденной записи содержит пустое значение;

– если указано несколько полей, то результатом будет вариантный массив, число элементов которого меньше или равно числу результирующих полей. Меньшее количество элементов может быть потому, что некоторые поля найденной записи могут содержать пустые значения.

Рассмотрим два примера.

Пусть имеется одно результирующее поле (результат – значение типа **Variant**). Будем осуществлять поиск в НД **Q_IncomingGoodsD** (приход товаров) по полю **gcName** (наименование товара). Поисковое значение будем вводить в компонент **Edit1**. В качестве результата будем выдавать значение поля **scName** (название фирмы-поставщика) найденной записи:

```
procedure TForm1.Lookup1ButtonClick(Sender: TObject);
var
  LookupResults: Variant; // Результат
begin
// Выполнить поиск
```

```
  LookupResults := DataModule2.Q_IncomingGoodsD.Lookup('gcName',
     Edit1.Text, 'scName');
// Обнулить выводимый результат
   Label1.Caption := ' ';
// Содержит ли результат пустое значение или Null?
   case VarType(LookupResults) of
     varEmpty:    Label1.Caption := 'Пустой результат';
     varNull:    Label1.Caption := 'Запись не найдена'
   else
// Нет, результат содержит какое-то значение
// Привести вариантный тип  к строковому
     Label1.Caption := LookupResults;
   end; // Case
end;
```

А теперь пусть имеется несколько результирующих полей (результат – вариантный массив). Если переменная типа **Variant** является вариантным массивом, то функция **VarIsArray(LookupResults)** возвращает значение **True**. При работе с переменным числом возвращаемых полей верхнюю и нижнюю границы массива **LookupResults** можно определить при помощи функций **VarArrayLowBound (LookupResults, I)** и **VarArrayHighBound (LookupResults, I)**. Тип **i**-го элемента вариантного массива можно задать как **VarType(LookupResults[i])**.

Будем осуществлять поиск в НД **Q_GoodsCatalogD** (товары) по полю **gcName** (наименование товара). Поисковое значение зададим на основе выбранной записи в НД **Q_IncomingGoodsD**, при этом для проверки название товара будем выводить в компонент **Edit1**. В качестве результата будем выдавать значения полей **tgName** (наименование типа товара), **gcMeasure** (единица измерения), **gcCost** (цена за единицу измерения) найденной записи:

```
procedure TForm4.Lookup2ButtonClick(Sender: TObject);
var
   LookupResults: Variant; // Результат
```

```
begin
// Вывести названия товара выбранной записи
  Edit1.Text := DataModule2.Q_IncomingGoodsDgcName.Value;
// Выполнить поиск
 LookupResults := DataModule2.Q_GoodsCatalogD.Lookup('gcName',
  DataModule2.Q_IncomingGoodsDgcName.Value, 'tgName; gcMeasure;
  gcCost');
// Отобразить значения результирующих полей в метках TLabel
  Label1.Caption := '  ';
  Label2.Caption := '  ';
  Label3.Caption := '  ';
// Проверить, является ли результат вариантным массивом
  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 // Then
  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, то параметр **VarType(LookupResults)** возвращает значение **varNull**. Если поиск по какой-либо причине не был произведен, то этот параметр возвращает значение **varEmpty**.

Если какое-либо из возвращаемых полей содержит пустое значение (**Null**), то соответствующий элемент вариантного массива также будет содержать пустое значение. В этом случае обращение к нему вызовет исключительную ситуацию, поэтому необходима предварительная проверка возвращаемого значения.

4.2.4. Инкрементальный локатор

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

Приведенные в п. 4.2.2 обработчики события нажатия кнопки **FindButton** используют локаторы. Однако вне рассмотрения остался еще один режим, при котором по вводу каждого символа в компонент **TEdit** курсор переходит на запись, расположенную ближе всего к искомой. Чем больше введено символов, тем ближе курсор к искомой записи. Такой локатор называется *инкрементальным*. Для его реализации используется обработчик события **OnChange** компонента **TEdit**, возникающего при любом изменении значения в компоненте.

В качестве примера рассмотрим создание инкрементального локатора для поля **gcName** компонента НД **Q_GoodsCatalogD**. Разместим на форме **Form4** компонент **TEdit**, зададим ему имя **E_GoodsNameSearch** и запишем для него обработчик события:

```
procedure TForm4.E_GoodsNameSearchChange(Sender: TObject);
begin
  DataModule2.Q_GoodsCatalogD.Locate('gcName',
    E_GoodsNameSearch.Text, [loPartialKey, loCaseInsensitive])
end;
```

Теперь по мере ввода названия товара курсор НД будет перемещаться к искомой записи.

4.3. Управление отображением данных в компоненте TDBGrid

Компонент **TDBGrid** используется для показа содержимого записей НД в табличном формате, когда строки соответствуют записям НД, а столбцы – полям записи. Свойство **DataSource** компонента **TDBGrid** содержит имя компонента **TDataSource**, который ссылается на соответствующий НД – компоненты **TADOTable** или **TADOQuery**.

Рассмотрим основные моменты, связанные с настройкой визуальных характеристик отображаемых полей в компоненте **TDBGrid**. Поскольку все поля в этом компоненте задаются в виде столбцов, то покажем, что представляют собой столбцы, и опишем их основные свойства.

4.3.1. Понятие столбцов в компоненте TDBGrid

Для определения состава столбцов в компоненте **TDBGrid** можно использовать редактор столбцов **Column Editor**. В этом случае учитываются только столбцы, созданные в редакторе столбцов компонента **TDBGrid**, и принимаются во внимание только их характеристики (в частности, порядок следования). Фактически в редакторе столбцов определяются постоянные столбцы.

Если редактор столбцов не применялся, то берутся поля, объявленные при помощи редактора полей НД (компоненты **TField**). Столбцы компонента **TDBGrid**, основанные на использовании компонентов **TField**, называются *динамическими столбцами*. При этом вид столбцов определяется соответствующими характеристиками компонентов **TField**, а порядок следования столбцов – порядком их определений. Для изменения заголовков столбцов в компоненте **TDBGrid** следует изменить значения свойств **DisplayLabel** для каждого поля.

В случае когда компоненты **TField** для НД не создавались, порядок следования полей и их характеристики соответствуют тем, что были заданы при определении структуры данной таблицы базы данных в момент ее создания.

Многие свойства компонента **TDBGrid** можно переустанавливать во время выполнения приложения, что предоставляет разработчикам большую гибкость при реализации интерфейса.

Для добавления нового столбца в **Columns Editor** нужно нажать кнопку **Add**. В результате будет создан столбец, который не связан ни с каким полем набора данных. Чтобы поставить такому столбцу в соответствие какое-либо поле НД, в инспекторе объектов следует рас-

крыть список в свойстве **Field Name** и выбрать нужное поле. В этом случае столбец будет назван так же, как поле.

При необходимости показа в компоненте **TDBGrid** столбцов, соответствующих всем или большинству полей набора данных, следует нажать кнопку **Add All Fieldes**. Тогда в список полей будут включены столбцы, соответствующие всем полям НД. Столбцы, которые не должны показываться в компоненте **TDBGrid**, удаляются при помощи кнопки **Delete**.

Используя кнопки **Move Up** и **Move Down**, можно изменить порядок следования столбцов. Свойства столбцов определяют особенности их отображения в компоненте **TDBGrid** (табл. 4.2).

Таблица 4.2

Основные свойства столбцов компонента TDBGrid

Свойство	Назначение
Alignment	Определяет выравнивание значений в столбце. По умолчанию соответствует свойству **Field.Alignment**
Color	Определяет цвет фона столбца. По умолчанию соответствует свойству **TDBGrid.Color**
FieldName	Определяет поле таблицы базы данных, ассоциированное с данным постоянным столбцом
ReadOnly	Определяет возможность редактирования столбца из компонента **TDBGrid** (значение **True**) или невозможность этого (значение **False** по умолчанию)

Окончание табл. 4.2

Свойство	Назначение
Width	Определяет ширину столбца в пикселях. По умолчанию значение извлекается из свойства **Field.DisplayWidth**, где ширина задается в символах
Font	Определяет тип, размер и цвет шрифта для вывода значений в столбце. По умолчанию соответствуют свойству **TDBGrid.Font**
Title.Alignment	Определяет выравнивание заголовка. По умолчанию выполняет выравнивание влево
Title.Caption	Определяет надпись заголовка. По умолчанию соответствует свойству **Field.DisplayLabel** или имени поля таблицы базы данных
Title.Font	Определяет тип, размер и цвет шрифта заголовка. По умолчанию соответствует свойству **TDBGrid.TitleFont**

Если необходимо изменить свойства постоянного столбца компонента **TDBGrid**, то к нему можно обратиться как к

DBGrid1.Columns.Items[i] или как к **DBGrid1.Columns[i]**, где **i** принадлежит диапазону **[0..DBGrid1.Columns. Count – 1]**.

Рассмотрим пример реализации динамического изменения свойств постоянных столбцов компонента **TDBGrid**. Для этого разместим на форме **Form1** невизуальные компоненты, обеспечивающие диалоги выбора шрифта **TFontDialog** и цвета **TColorDialog** (они находятся на вкладке **Dialogs** палитры компонентов среды **Delphi**), а также кнопки **TButton**, нажатие которых инициирует изменение свойств постоянных столбцов (**Шрифт столбцов**, **Шрифт заголовков**, **Шрифт столбца**, **Фон столбца**). Зададим обработчики событий нажатия кнопок, реализующих возможности настройки отображения в компоненте **TDBGrid**:

```
// Обработчик нажатия кнопки Шрифт столбцов
procedure TForm1.ColumnsFontButtonClick (Sender: TObject);
var
  i: Integer;
begin
// В диалоге выбора шрифта текущим будет шрифт
// нулевого столбца
  FontDialog1.Font := DBGrid1.Columns.Items[0].Font;
  if FontDialog1.Execute then
    for i := 0 to DBGrid1.Columns.Count – 1 do
      DBGrid1.Columns.Items[i].Font := FontDialog1.Font;
end;


// Обработчик нажатия кнопки Шрифт заголовков
procedure TForm1.CaptionsColumnsFontButtonClick (Sender:
TObject);
var
  i: Integer;
begin
// В диалоге выбора шрифта текущим будет шрифт заголовка
```

```
// нулевого столбца
  FontDialog1.Font := DBGrid1.Columns.Items[0].Title.Font;
  if FontDialog1.Execute then
    for i := 0 to DBGrid1.Columns.Count – 1 do
      DBGrid1.Columns.Items[i].Title.Font := FontDialog1.Font;
end;

// Обработчик нажатия кнопки Шрифт столбца
procedure TForm1.SelectedColumnFontButtonClick (Sender: TObject);
begin
// В диалоге выбора шрифта текущим будет шрифт текущего
// столбца
FontDialog1.Font :=
DBGrid1.Columns.Items[DBGrid1.SelectedIndex].Font;
  if FontDialog1.Execute then
    DBGrid1.Columns.Items[DBGrid1.SelectedIndex].Font :=
      FontDialog1.Font;
end;

// Обработчик нажатия кнопки Фон столбца
procedure TForm1.SelectedColumnColorButtonClick (Sender: TObject);
begin
// В диалоге выбора цвета текущим будет цвет текущего столбца
ColorDialog1.Color := DBGrid1.Columns.Items[DBGrid1.
SelectedIndex].Color;
  if ColorDialog1.Execute then
    DBGrid1.Columns.Items[DBGrid1.SelectedIndex].Color :=
      ColorDialog1.Color;
end;
```

Протестировав это приложение, мы сможем убедиться в правильности его работы.

4.3.2. Подсветка записей в компоненте DBGrid

Рассмотрим средства выделения (цветом, изменением шрифта и т. д.) какого-либо столбца в компоненте **TDBGrid**. Предварительно отметим, что на характеристики представления значений в полях компонента **TDBGrid** могут оказывать влияние такие свойства компонентов **TField**, как **DisplayFormat** и **EditFormat**, которые позволяют задавать формат поля при его показе и редактировании соответственно. Эти свойства актуальны для всех компонентов, визуализирующих значение поля (полей), а не только для столбцов компонента **TDBGrid**.

Перечисленные в п. 4.3.1 средства настройки отображения носят безусловный характер, т. е. они позволяют изменять характеристики показа всех значений в конкретном столбце. Однако может потребоваться выделение не целого столбца, а определенных полей столбца, отвечающих какому-либо условию. Например, для бухгалтерских и складских программ часто принято выделять красным цветом отрицательные остатки или остатки, находящиеся ниже установленного лимита.

Способ прорисовки данных в компоненте **TDBGrid**: стандартный или по особому сценарию – определяется свойством

```
property DefaultDrawing: Boolean;
```

Если прорисовка управляется со стороны самого приложения, то алгоритм прорисовки должен содержаться в обработчиках событий **OnDrawColumnCell** или **OnDrawDataCell** (этот обработчик введен для обеспечения совместимости с ранними версиями **Delphi**).

При автоматической прорисовке (свойство **DefaultDrawing** установлено в значение **True**) метод **Paint** использует цвета и шрифт свойства **Canvas** для прорисовки полей. Событие

```
property OnDrawColumnCell: TDrawColumnCellEvent;

TDrawColumnCellEvent = procedure(Sender: TObject; const Rect:
TRect;

DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;
```

имеет следующие параметры:

– **Rect** определяет область поля, где происходит прорисовка. Тип

```
TRect = record
        case Integer of
                0: (Left, Top, Right, Bottom: Integer);
                1: (TopLeft, BottomRight: TPoint);
        end;
end;
```

задает прямоугольник, в котором прорисовывается поле компонента **TDBGrid**. Координаты этого прямоугольника указываются или как четыре целых числа, содержащих границы прямоугольника в пиксельном исчислении, или как два значения типа **Point**, соответствующих левому верхнему и правому нижнему углам прямоугольника:

```
TPoint = record
        X: LongInt;
        Y: LongInt
end;
```

– **DataCol** показывает порядковый номер текущего столбца, начиная с нулевого столбца;

– **Column: TColumn** задает текущий столбец;

– **State: TGridDrawState** характеризует состояние поля. Тип

```
TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
```

определяет множество возможных состояний поля компонента **TDBGrid**.

Если поле находится в состоянии **gdFocused**, т. е. является текущим, то в обработчике события **OnDrawColumnCell** происходит прорисовка инверсной полосы.

Для вывода полей по умолчанию используется метод

```
procedure DefaultDrawColumnCell(const Rect: TRect; DataCol:
Integer; Column: TColumn; State: TGridDrawState);
```

Отображение как целых строк, так и отдельных полей можно изменять, используя информацию, содержащуюся в передаваемых в обработчик **OnDrawColumnCell** параметрах.

Пусть нам необходимо вывести белым шрифтом на красном фоне те строки компонента НД **Q_IncomingGoodsD**, у которых поле **igAmount** содержит значение меньше **15**. Приведем программный код обработчика **OnDrawColumnCell**:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const
 Rect: TRect; DataCol: Integer; Column: TColumn; State:
 TGridDrawState);
begin
  with DBGrid1.Canvas do
  begin
// Проверить, не меньше ли значение поля igAmount 15, а также
// выделена ли строка
  if (DataModule2.Q_IncomingGoodsDigAmount.Value <= 15) and not
    (gdSelected in State) then
// Если условия выполняются, то задать формат вывода
  begin
// Установка красного фона
      Brush.Color := clRed;
// Установка белого шрифта
      Font.Color := clWhite;
    end;
  end; // With
// Отрисовка поля
  DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

В отличие от предыдущего примера, в приведенном ниже примере выделяются не строки, а поля:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const
 Rect: TRect; DataCol: Integer; Column: TColumn; State:
 TGridDrawState);
```

```
begin
  with DBGrid1.Canvas do
  begin
// Проверить название поля igAmount и факт его выделения
  if (Column.FieldName = 'igAmount') and not(gdSelected in State) then
// Проверить, не меньше ли значение поля igAmount 15
      if (DataModule2.Q_IncomingGoodsDigAmount.Value <= 15) then
// Если условие выполняется, то задать формат вывода
      begin
// Установка красного фона
        Brush.Color := clRed;
// Установка белого шрифта
        Font.Color := clWhite;
      end;
    end; // With
// Отрисовка поля
  DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

4.3.3. Изменение сортировки полей во время выполнения приложения на основе события OnTitleClick

Часто при просмотре данных в компоненте **DBGrid** появляется необходимость изменить сортировку НД по какому-нибудь столбцу. А поскольку **ADO**-компоненты обладают свойством **Sort**, то сортировка НД не представляет большой сложности. Необходимо только определить способ ее выполнения. Наиболее оптимальным решением является написание обработчика события **OnTitleClick**, которое возникает, когда мы щелкаем мышью по заголовкам столбцов.

Для реализации механизма сортировки на основе заголовков столбцов нужно объявить глобальную переменную **CSort** целочисленного типа, которая будет показывать, по какому столбцу выполня-

лась сортировка в прошлый раз. В блоке **var** запишем объявление следующего вида:

```
CSort: Integer = -1; // Инициализировать значение –1, так как
// не было сортировки
```

Теперь зададим обработчик события **OnTitleClick** для **DBGrid1**:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
var
  sorttype: string;
begin
 if DataModule2.Q_IncomingGoodsD.FieldByName(Column.FieldName).
    FieldKind <> fkCalculated then
 begin
// Сбросить параметры для колонки, с которой работали
// в прошлый раз
    if CSort >= 0 then
    begin
      DBGrid1.Columns[CSort].Title.Font.Color := clBlack;
      DBGrid1.Columns[CSort].Title.Font.Style :=
Column.Title.Font.Style – [fsBold];
    end;
// Установить параметр сортировки как пустое значение
// (соответствует сортировке по возрастанию ASC)
    sorttype := '';
// Запомнить индекс сортируемого столбца
    CSort := Column.Index;
// Проверить, не выполнялась ли сортировка по убыванию
// для данного поля
    if DataModule2.Q_IncomingGoodsD.Sort <> Column.FieldName +
'DESC'
```

```
    then begin
  // Если проверка не выполнялась, то сверить, сортировали ли
  // по этому полю или нет
    if DataModule2.Q_IncomingGoodsD.Sort = Column.FieldName then
  // Если сортировка по данному полю уже была, то поменять
  // тип сортировки
        sorttype := ' DESC';
  // Задать стили шрифта заголовка для отображения вида
  // сортировки
      Column.Title.Font.Style := Column.Title.Font.Style + [fsBold];
      if sorttype = '' then
        Column.Title.Font.Color := clGreen // По возрастанию
      else
        Column.Title.Font.Color := clBlue; // По убыванию
  // Установить сортировку для поля с заданным параметром
  DataModule2.Q_IncomingGoodsD.Sort := Column.FieldName + sorttype;
    end
    else
  // Если в прошлый раз использовалась сортировка по убыванию,
  // то отключить сортировку
      DataModule2.Q_IncomingGoodsD.Sort := '';
  end;
end;
```

В представленном исходном коде стоит обратить внимание на пробелы, которые добавляются к строковым переменным, например: **sorttype := '_DESC';**.

Описанный механизм сортировки выясняет поле, по которому необходимо выполнить сортировку в момент работы программы. В связи с этим его можно использовать как универсальное средство во время разработки разнообразных программных систем, использующих отображение данных с возможностью сортировки.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться со способами фильтрации набора данных (см. п. 4.1.1).
2. Реализовать фильтрацию по стоимости товара (см. п. 4.1.2).
3. Выполнить фильтрацию с помощью свойства **Filter** (см. п. 4.1.3).
4. Провести фильтрацию по текстовому полю с использованием обработчика **OnChange** компонента **TEdit** (см. п. 4.1.4).
5. Спроектировать фильтрацию, позволяющую выбирать дату из определенного диапазона (см. п. 4.1.5).
6. Реализовать поиск записей по методу **Locate** (см. п. 4.2.1).
7. Организовать поиск записей с использованием методов **FindFirst**, **FindLast**, **FindNext**, **FindPrior** (см. п. 4.2.2).
8. Выполнить поиск с использованием метода **Lookup** (см. п. 4.2.3).
9. Реализовать инкрементальный локатор (см. п. 4.2.4).
10. Организовать изменение типов отображаемых шрифтов в компоненте **TDBGrid** (см. п. 4.3.1).
11. Реализовать подсветку записей в **DBGrid** (см. п. 4.3.3).
12. Выполнить сортировку записей набора данных с использованием события **OnTitleClick** (см. п. 4.3.4).
13. Ответить на контрольные вопросы.
14. Составить отчет, который должен содержать титульный лист, цель и ход выполнения лабораторной работы, сформулированные выводы.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Перечислите средства для фильтрации записей в наборах данных.
2. С какой целью используется свойство **Filtered**?
3. Каким образом осуществляется фильтрация набора данных с использованием **SQL**-запроса?
4. Укажите область использования ключевого слова **Accept**.
5. Какие изменения будут происходить с НД при установке свойства **Filtered** в значение **True**?
6. Охарактеризуйте свойство **Filter**.
7. Опишите событие **OnFilterRecord**.
8. Какие поля НД могут использоваться в условиях фильтрации?
9. Каким образом осуществляется фильтрация в обработчике **OnFilterRecord**?

10. В каких случаях следует применять событие **OnFilterRecord**?

11. Какие изменения происходят с НД при обработке события **OnFilterRecord**?

12. Охарактеризуйте известные вам способы фильтрации НД.

13. Каким образом можно организовать фильтрацию по нескольким полям?

14. Для каких целей применяется компонент **TGroupBox**?

15. Для чего используется функция **Assigned**?

16. Какие операторы отношения и логические операторы можно использовать при организации фильтрации?

17. Укажите особенности реализации фильтрации по текстовым полям.

18. Объясните необходимость применения ключевого слова **LIKE**.

19. Охарактеризуйте функцию **QuotedStr**.

20. С какой целью используется функция **FormatDateTime**?

21. Для чего предназначен метод **Locate**?

22. Перечислите параметры метода **Locate** и дайте их характеристику.

23. Приведите фрагмент поиска по двум полям для метода **Locate**.

24. Сравните метод **Locate** с другими методами поиска записей.

25. Укажите назначение методов **FindFirst**, **FindLast**.

26. Опишите методы **FindNext**, **FindPrior**.

27. Для чего используется свойство набора данных **Found**?

28. Дайте характеристику метода **Lookup**.

29. Какие параметры имеются у метода **Lookup**? В чем состоит их назначение?

30. Какие поля можно указывать в качестве поисковых в методе **Lookup**?

31. Какой результат может возвратить метод **Lookup**?

32. Что представляет собой локатор записей в наборе данных?

33. Дайте определение инкрементального локатора.

34. Для каких целей предназначен редактор столбцов?

35. Охарактеризуйте способы изменения местоположения столбца.

36. Каким образом можно изменить заголовки столбцов в компоненте **TDBGrid**?

37. Перечислите свойства столбцов, которые можно изменять программно.

38. Какие способы прорисовки данных в компоненте **TDBGrid** вам известны?

39. Каким образом осуществляется автоматическая прорисовка данных?

40. Опишите процесс изменения сортировки полей во время выполнения приложения.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Бекаревич, Ю. Самоучитель Access 2010 / Ю. Бекаревич, Н. Пушкина. – СПб. : БХВ-Петербург, 2011. – 432 с.

2. Кошелев, В. Е. Access 2007. Эффективное использование / В. Е. Кошелев. – М. : Бином-Пресс, 2007. – 590 с.

3. Культин, Н. Программирование в Delphi 2010. Самоучитель / Н. Культин. – СПб. : БХВ-Петербург, 2010. – 448 с.

4. Осипов, Д. Л. Базы данных и Delphi. Теория и практика / Д. Л. Осипов. – СПб. : БХВ-Петербург, 2011. – 752 с.

5. Сурядный, А. С. Microsoft Access 2010. Лучший самоучитель / А. С. Сурядный. – СПб. : Астрель ВКТ, 2012. – 448 с.

6. Сухарев, М. Delphi. Полное руководство. Включая версию 2010 / М. Сухарев. – М. : Наука и техника, 2010. – 1040 с.

7. Фленов, М. Библия Delphi / М. Фленов. – 3-е изд. – СПб. : БХВ-Петербург, 2011. – 674 с.