

**И. БАБУШКИНА  
С. ОКУЛОВ**

**ПРАКТИКУМ**

## ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ

# ПРОГРАММИРОВАНИЮ

# Object Pascal

## Object Passing

# Object Pascal

## Object Pascal

# Object Pascal

## Object Pascal

# Object Pascal

Object Pascal

## Object Pascal

# Object Pascal

# Object Pascal

Object Pascal

Object Pascal  
Object Based

## Object Paste



**И. БАБУШКИНА, С. ОКУЛОВ**

# **ПРАКТИКУМ**

## **ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПРОГРАММИРОВАНИЮ**

4-е издание (электронное)



**Москва**  
**БИНОМ. Лаборатория знаний**  
**2 0 1 5**

УДК 519.85(023)

ББК 22.18

Б12

**Бабушкина И. А.**

**Б12** Практикум по объектно-ориентированному программированию [Электронный ресурс] / И. А. Бабушкина, С. М. Окулов. — 4-е изд. (эл.). — Электрон. текстовые дан. (1 файл pdf : 369 с.). — М. : БИНОМ. Лаборатория знаний, 2015. — Систем. требования: Adobe Reader XI ; экран 10".

ISBN 978-5-9963-2542-9

Практикум содержит материал для проведения занятий по объектно-ориентированному программированию в среде Delphi. Изложены основы теории объектно-ориентированного программирования, на базе которой изучаются возможности Delphi и отрабатываются технологии разработки различных приложений. Разобрано около 50 упражнений, приведено более 140 заданий для самостоятельной работы и 60 вопросов для контроля.

Для учителей информатики, преподавателей высших учебных заведений, старшеклассников школ с углубленным изучением информатики, студентов, обучение которых связано с Computer Science.

**УДК 519.85(023)**

**ББК 22.18**

**Деривативное электронное издание на основе печатного аналога:** Практикум по объектно-ориентированному программированию / И. А. Бабушкина, С. М. Окулов. — 2-е изд. — М. : БИНОМ. Лаборатория знаний, 2009. — 366 с. : ил. — ISBN 978-5-9963-0219-2.

**В соответствии со ст.1299 и 1301 ГК РФ при устранении ограничений, установленных техническими средствами защиты авторских прав, правообладатель вправе требовать от нарушителя возмещения убытков или выплаты компенсации**

**ISBN 978-5-9963-2542-9 © БИНОМ. Лаборатория знаний, 2004**

# Введение

Среди современных информационных технологий программирование занимает особое место как в Computer Science, так и в образовательной информатике. Обучение программированию — сложнейший процесс. Вспомним, как быстро сошла на нет эйфория первых лет внедрения обучения информатике в школу с основным упором на программирование и содержание обучения постепенно изменилось. Может быть, одна из причин, а их, разумеется, много, заключается именно в сложности этого процесса. Еще одно обстоятельство этого угасания интереса к программированию как к образовательной дисциплине заключается, видимо, в отсутствии литературы для преподавателя. Классические книги как, например, многотомник Д. Кнута «Искусство программирования для ЭВМ», достаточно трудны и не всегда согласуются с реалиями конкретной деятельности. Появившиеся в огромном количестве в последние годы книги содержат, в основном, описания конкретных систем программирования и могут служить скорее справочным материалом, а не конкретным руководством к действию. Преподавателю придется изучать самостоятельно материал, а затем разрабатывать систему занятий. В данной книге сделана попытка интегрировать в единое целое эти два аспекта: ее можно использовать и при самостоятельном изучении среды программирования Delphi, и при подготовке занятий; это позволяет авторам надеяться, что проделанная работа принесет коллегам пользу.

Данная книга не родилась в одночасье. Еще в 1990 г. при подготовке учителей информатики в Вятском государственном педагогическом университете была осознана необходимость изучения объектно-ориентированного программирования. Вначале изучение предмета осуществлялось на базе спецкурса. Издание пособия [8] в какой-то мере подводило итог этой работы. С 1995 г. был введен обязательный курс по объектно-ориентированному программированию в среде Delphi. Материал книги достаточно полно отражает его содержание. Курс является, на наш взгляд, завершающим в базовой, фундаментальной подготовке по программированию, которая формируется в результате изучения трех курсов: практикума по решению за-

дач<sup>1</sup>, основ программирования<sup>2</sup> и объектно-ориентированного программирования. Завершает подготовку (на старших курсах) изучение ряда спецкурсов по различным аспектам этой ключевой деятельности в Computer Science. Данная книга рассчитана на подготовленного читателя, тем не менее она может оказаться по силам и школьнику. В виде спецкурса материал практикума изучался в физико-математическом лицее г. Кирова. Курс информатики в лицее построен так, что к 10-му классу большинство учащихся достаточно уверенно работают в среде Pascal, а основой среды Delphi, как известно, является Object Pascal — объектно-ориентированная версия языка программирования Pascal. Таким образом, обучение по своему содержанию логично, изучаются фундаментальные основы настоящей Computer Science, этих основ не так много, и в обучении в максимальной степени задействован компьютер с его огромным дидактическим потенциалом. Можно с полной уверенностью сказать, и практика это подтверждает, что результатом «проведения» ученика по данной схеме является то, что компьютер становится для него обычным инструментом, независимо от вида его будущей деятельности. Учащийся абсолютно уверенно работает и с технологиями, которые им ранее не изучались. Это является высшей оценкой результатов деятельности учителя, ибо в учащегося заложена не сумма фактографического материала, а умение учиться и осваивать новое. Сформулируем следующую, может быть абсурдную с точки зрения ряда специалистов образовательной информатики, мысль — ядро информатики не зависит от конкретной информационной технологии, будь то текстовый процессор, программа на языке Pascal или табличный процессор. Соответственно его, это ядро, можно изучать по-разному, но главное — изучается одно и то же. Обучение программированию отличается только тем, что оно ближе всего к этим фундаментальным основам Computer Science и, кроме того, активность учащегося и задействованность компьютера более высокая.

В первой главе практикума изложены основы теории объектно-ориентированного программирования, а затем изучаются возможности Delphi и отрабатываются технологии разработки различных приложений. «Жесткой» разбивки на аудиторные

---

<sup>1</sup> Окулов С. М. и др. Задачи по программированию. 1000 задач с решениями. — М.: БИНОМ. Лаборатория знаний, 2004.

<sup>2</sup> Окулов С. М. Основы программирования. — М.: БИНОМ. Лаборатория знаний, 2004.

занятия нет. С одной стороны, это недостаток, а с другой — преподаватель может гибко планировать занятия, используя материал данного учебника в зависимости от объема выделяемого на изучение дисциплины времени и уровня подготовленности учащихся.

Каждая тема содержит дозированную порцию теоретического материала, а затем следует разбор практических задач по нему с необходимыми комментариями и упражнениями. При разборе упражнений необходимо быть особенно внимательным, следовать указаниям каждого пункта работы и следить за правильностью получаемых результатов. Изложение темы заканчивается заданиями для самостоятельной работы и вопросами для повторения. В общей сложности в практикуме разобрано порядка 50 упражнений, приведено более 140 заданий для самостоятельной работы и 60 вопросов для проверки усвоения материала. По мере прохождения практикума задания становятся все сложнее, и к моменту завершения работы с ним приобретает достаточно высокий профессиональный уровень владения объектно-ориентированной технологией.

Мы надеемся, что практикум будет полезен не только преподавателям, но и всем желающим освоить современное программирование.

# Объектно-ориентированное программирование

## 1.1. Из истории развития языков программирования

Любая методология программирования имеет свою концептуальную основу. Эволюция развития языков программирования направлена на ускорение процесса создания надежных программных средств.

Все программы в ЭВМ на самом низком, аппаратном, уровне приводятся в действие только командами машинного языка. История развития языков программирования началась с появления ассемблера, который представляет собой символическое представление машинного языка. Программа на ассемблере работает на уровне аппаратных средств, входящих в программную модель микропроцессора. При разработке алгоритма программы и его реализации на ассемблере программист должен продумать размещение данных в памяти, обеспечить эффективное использование ограниченного количества регистров, продумать организацию связи с операционной системой и другими программами.

Следующий этап развития языков программирования связан с появлением подпрограмм, а также созданием технологии структурного программирования. В основе структурного программирования лежит декомпозиция (разбиение на части) программы с целью выделения подзадач и реализацией их в виде подпрограмм.

Дальнейший рост сложности программного обеспечения потребовал развития структурированных типов данных, а также развития модульного программирования. Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля ограничен рамками модуля.

Программы продолжали усложняться, в связи с чем потребовалось уже не случайное объединение данных и алгоритмов их обработки в единое целое, а смысловое: необходимо было создать модульное программирование нового уровня, когда основ-

ной акцент делается на смысловую связь структур данных и алгоритмов их обработки. Это привело к развитию языков объектно-ориентированной технологии решения задач.

**Объектно-ориентированное программирование (ООП)** — это технология, основанная на представлении программ в виде совокупности объектов, каждый из которых является реализацией собственного класса, которые в свою очередь образуют иерархию на принципах наследования.

Основное достоинство ООП — сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями, по сравнению с модульным программированием. Это достигается посредством более полной локализации данных и интегрирования их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы.

## 1.2. Объектная декомпозиция

При использовании технологии ООП решение представляется в виде результата взаимодействия отдельных элементов некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи. Каждый элемент системы, получая сообщение, выполняет заранее определенную последовательность действий (например, обрабатывает полученные данные, изменяет свое состояние, пересылает полученные данные другому элементу системы). Передавая сообщения от одного элемента системы к другому, система выполняет поставленную перед ней задачу.

Элементы системы, параметры и поведение которой определяются условием задачи, обладающие самостоятельным поведением (т. е. «умеющие» выполнять некоторые действия, зависящие от полученных сообщений и состояния элемента), получили название *объектов* [4, с. 25].

Процесс представления предметной области в виде совокупности объектов, обменивающихся сообщениями, называется *объектной декомпозицией*.

**Упражнение 1.2.1.** Выполните объектную декомпозицию программы, которая по запросу пользователя рисует точку, окружность или квадрат.

## Решение

По правилам объектной декомпозиции разрабатывается имитационная модель программы. Для этого необходимо проанализировать все происходящие в системе процессы и выделить элементы, обладающие собственным поведением, воздействующие на другие элементы и/или являющиеся объектами такого воздействия.

Основная цель системы — нарисовать фигуру, выбранную пользователем. Действия пользователя — это либо выбор фигуры, либо изменение параметров фигуры (цвет, размер, координаты), либо команда нарисовать выбранную фигуру с заданными параметрами. Для выполнения этих команд можно воспользоваться следующими объектами: Менеджер (получает, анализирует и обрабатывает команды пользователя) и три объекта — фигуры (каждая со своими параметрами).

Фигуры получают следующие сообщения: нарисовать, изменить цвет контура, изменить размер, изменить координаты. Все эти сообщения инициируются Менеджером в соответствии с командой пользователя. Получив от пользователя команду Завершить, Менеджер прекращает выполнение программы.

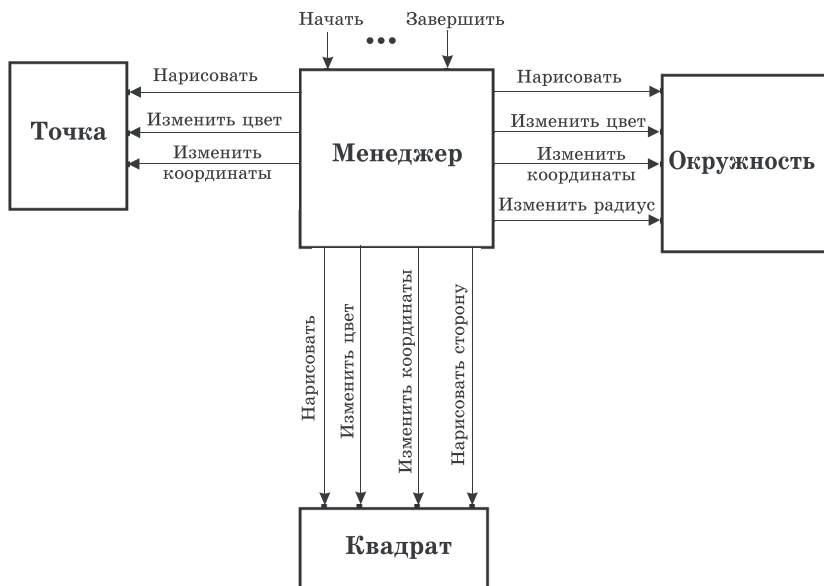


Рис. 1.2.1

**Упражнение 1.2.2.** Выполните объектную декомпозицию системы «Телефонный справочник».

**Решение**

Опишем назначение системы. В системе хранятся данные о родственниках, друзьях, знакомых или коллегах по работе: фамилия, имя, отчество, дата рождения, адрес, домашний телефон, характер знакомства, место работы, должность, рабочий телефон. Система при запуске выдает информацию о тех людях, чей день рождения приходится на текущую дату, осуществляет поиск данных по произвольному формату, позволяет добавлять, редактировать, удалять записи.

Для выполнения выделенных задач можно воспользоваться следующими объектами: Менеджер (получает, анализирует и обрабатывает команды пользователя), Поисковик (осуществляет поиск записей по определенным данным), ОбработатьЗапись (добавляет, редактирует, удаляет запись), Файл (для хранения записей, получает сообщения от объектов Поисковик, ОбработатьЗапись), ОткрытиеФайла (отвечает за существование файла, если файл не существует, то он создается).



**Рис. 1.2.2**

В объектно-ориентированном программировании разрабатываемая система состоит из объектов, которые взаимодействуют через передачу сообщений.

Каждый объект, получив сообщение, должен определенным образом реагировать на них, выполняя заранее определенные для каждого типа сообщения действия. Например, если объект `ОбработатьЗапись` будет активизирован, то он должен будет проанализировать, какое именно действие нужно выполнить. Если принято сообщение `Редактировать`, то объект должен сохранить запись в файл, на место, указанное курсором.

**Состояние** объекта характеризуется набором конкретных значений некоторого перечня всех возможных свойств данного объекта; например, состояние объекта `Файл` характеризуется значениями «активизирован» — «не активизирован». Это состояние объекта необходимо для выполнения всех действий над записной книжкой: если `Файл` находится в состоянии «не активизирован», то ни одно сообщение не сможет быть обработано.

Набор значений свойств задается на этапе проектирования и не изменяется в процессе функционирования, изменяются лишь конкретные значения.

**Поведение** объектов характеризуется определенным набором реакций на получаемые сообщения и зависит от состояния объекта.

Если объект может обладать некоторым состоянием, то, соответственно, может возникнуть необходимость в получении информации об этом состоянии. Для получения такой информации объекту посылается сообщение-запрос. В ответ на запрос объект должен переслать отправителю требуемую информацию. В таких случаях говорят, что над объектом выполнена операция селекции.

Обращение к объекту для изменения его состояния возбуждает выполнение операции модификации. Отправитель сообщения-команды, реакцией на которую должна быть модификация объекта, может ожидать завершения операции, а может продолжить выполнение своей программы.

Если объект содержит несколько однотипных компонент, например массив чисел, то операция, требующая последовательной обработки этих компонент, называется итерацией. Поэлементно могут выполняться как операции селекции, так и операции модификации.

### 1.3. Основные элементы ООП

ООП характеризуется четырьмя основополагающими идеями (абстрагирование, инкапсуляция, модульность, иерархия) и тремя дополнительными (типизация, параллелизм, сохранемость).

**Абстрагирование** — это один из главных способов решения сложных задач.

В результате объектной декомпозиции были выделены объекты. Абстракция предназначена для выделения существенных характеристик каждого объекта, отличающих его от всех других видов объектов и, таким образом, четко определяются его концептуальные границы с точки зрения наблюдателя.

Для представления абстракций объектов используется специальный определяемый программистом тип данных — класс.

**Класс** — это структурный тип данных, который включает описание полей данных, а также процедур и функций, работающих с этими полями данных. Процесс объединения данных с действиями над этими данными в единый пакет при наличии специальных правил доступа к элементам пакета получил название **инкапсуляция**.

Итак, сочетание данных с допустимыми действиями над этими данными приводит к «рождению» нового «кирпичика» программирования — класса. Действия — это процедуры и функции, описанные в классе, они получили название методов.

Класс представляет собой структуру, динамически размещаемую в памяти. Экземпляр класса называется объектом. Прежде чем программа сможет использовать объект какого-либо класса, его необходимо создать. Объекты создаются и уничтожаются с помощью специальных методов, которые называются **constructor** (конструктор) и **destructor** (деструктор).

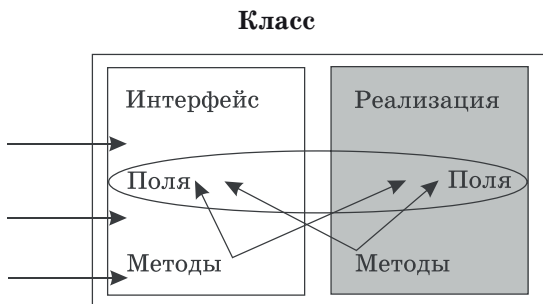
«Рожденный ползать летать не может» — и объект действует только так, как это в нем заложено, и только над тем, что в нем описано. Обращение к данным объекта не через его методы недопустимо.

Для корректной работы абстракции доступ к ее внутренней структуре должен быть ограничен. Для этого вводятся две части в описании абстракции (рис. 1.3.1).

**Интерфейс** — это совокупность доступных извне элементов реализации абстракции, т. е. основные характеристики состояния и поведения.

**Реализация** — это совокупность недоступных извне элементов реализации абстракции, т. е. внутренняя организация абстракции и механизмы реализации ее поведения.

Наличие интерфейса обеспечивает уменьшение возможности «разрушения» (несанкционированного изменения значений полей) объекта извне. При этом сокрытие особенностей реализации упрощает внесение изменений в реализацию класса как в процессе отладки, так и при модификации программы. Таким образом, класс определяет существование глобальной области данных внутри объекта, доступной методам объекта. С другой стороны, доступ к объекту регламентируется и должен выполняться через специальный интерфейс.



**Рис. 1.3.1**

Для описания нового класса в языке Object Pascal определен следующий синтаксис:

```
Type <имя_объявляемого_класса>=class (<имя_класса_родителя>)
    Private
        <скрытые_элементы_класса>
    Protected
        <защищенные_элементы_класса>
    Public
        <общедоступные_элементы_класса>
    Published
        <опубликованные_элементы_класса>
end;
```

Директивы **private**, **protected**, **public**, **published** предназначены для ограничения доступа к элементам класса.

Секция **private** содержит внутренние элементы, обращение к которым возможно только в пределах модуля, содержащего объявление класса.

Секция **protected** содержит защищенные элементы, которые доступны в пределах модуля, содержащего определение класса, и внутри классов-потомков.

Секция **public** содержит общедоступные элементы, к которым возможно обращение из любой части программы.

Секция **published** содержит опубликованные элементы, которые по ограничению доступа аналогичны **public**. Для визуальных компонент, (внесенных на панель компонент), информация об элементах, размещенных в этой секции, становится доступной через инспектор объектов.

Потомки класса могут менять область доступности всех элементов родительского класса, кроме элементов, объявленных в секции **private**, так как последние им недоступны.

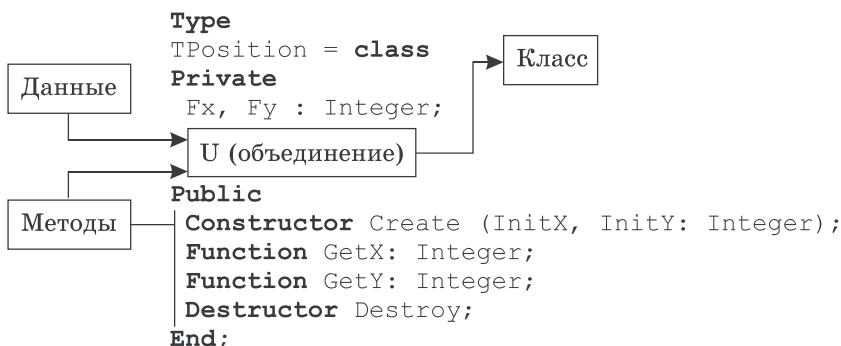
Все объекты Delphi являются динамическими, т. е. размещаемыми в динамической области памяти. Соответственно переменная типа класса по смыслу представляет собой указатель на объект. Вызовы конструктора и деструктора являются обязательными, так как конструктор выполняет размещение объекта в памяти, а деструктор — выгрузку из нее.

**Упражнение 1.3.1.** Разработать класс, переменные которого используются для описания положения геометрической фигуры на экране.

### Решение

Проектируемый класс должен содержать поля для сохранения положения элемента на экране — координаты местоположения  $x$  и  $y$ , при этом возможными действиями являются инициализация элемента, получение координат, разрушение объекта.

Определим класс Position:



Для хранения значений координат введем два поля `Fx` и `Fy` (в языке Object Pascal принято соглашение названия полей начинать с символа `F` (от слова `Field` — поле)). Это внутренние данные класса, чтобы обеспечить их целостность, опишем их в разделе `private`.

Constructor `Create` предназначен для создания экземпляра класса (объекта), а также для определения начальных значений его полей.

Destructor `Destroy` предназначен для удаления объекта из динамической памяти.

Подчеркнем, выделим мысль о том, что согласно идеологии объектно-ориентированного программирования все действия с данными, определенными в классе, осуществляются только путем использования методов объекта. Методы `GetX` и `GetY` по запросу обращаются к соответствующему полю объекта и возвращают координаты положения объекта.

Итак, объединение данных с действиями над этими данными порождает новый тип, а процесс называется инкапсуляцией.

Создавая объекты типа `TPosition`, инициализируя их в соответствии с условием, получим разные положения на экране, причем параметры будут храниться внутри объектов.

Каждая переменная типа `class` включает набор полей, объявленных в классе. Совокупность значений, содержащихся в этих полях, моделирует конкретное состояние объекта предметной области. Изменение этих значений в процессе работы отражает изменение состояния моделируемого объекта.

Воздействие на объект выполняется посредством изменения его полей или вызова его методов. Доступ к полям и методам объекта осуществляется, за исключением специальных случаев, с указанием имени объекта (при этом используются составные имена):

```
<имя_объекта>.<имя_поля>;
```

или

```
<имя_объекта>.<имя_метода>;
```

Все методы объекта обязательно имеют доступ ко всем полям своего объекта. В языке Object Pascal это достигается через неявную передачу в метод специального параметра `Self` — адреса области данных конкретного объекта. Таким образом, уменьшается количество параметров, явно передаваемых в метод.

**Модульность** — это свойство программы, связанное с декомпозицией ее на ряд отдельных фрагментов, которые компилируются по отдельности, но могут устанавливать связи между собой. Связи между модулями — это их представление друг о друге.

Доступ к данным объекта не через его методы запрещен! Кроме того, объекты должны ограничивать свои операции только их собственными данными и не должны быть связанными ни с какими глобальными переменными, а также не должны изменять их.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Модули исполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Для описания небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям.

В традиционном структурном программировании модульность — это искусство раскладывать программы на части так, чтобы в один контейнер попадали подпрограммы, использующие друг друга или изменяемые вместе. В ООП ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.

Особенности системы, подверженные изменениям, следует скрывать в отдельном модуле. В качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к данным из модуля должен осуществляться только через процедуры данного модуля. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями.

Модульность — это свойство системы, которая была разложена на внутренне связанные, но слабо связанные между собой модули.

Правила разделения системы на модули.

1. Распределение классов и объектов по модулям должно учитывать то, что модули служат в качестве элементарных и неделимых блоков программы.
2. Многие компиляторы создают отдельный сегмент кода для каждого модуля, поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и управление страницами виртуальной памяти.

**Иерархия.** Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры.

Иерархия — расположение частей или элементов целого от высшего к низшему, это упорядочение абстракций, расположение их по уровням. Одним из важных видов иерархии является *наследование*.

Программист для решения определенного класса задач может строить иерархию классов, в которой, и это самое главное, каждый следующий производный класс имеет доступ (наследует) к данным и действиям всех своих предшественников (прародителей). Потомок получает в свое распоряжение все, что принадлежало его предку. Потомок может добавить новые методы, свойства или поля и изменить реализацию любого метода, но не может их уничтожить. Согласно определению наследования, поля и методы предка доступны его потомку. Если в потомке создается одноименное поле или метод, можно говорить о перекрытии полей и методов.

В ООП используют два вида иерархии.

Иерархия «целое—часть» показывает, что некоторые абстракции включены в некоторую абстракцию как ее части, например, строение цветка описывается следующими частями: цветоножке, пестик, тычинки, цветоножка, завязь, лепестки. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне — при декомпозиции предметной области на объекты, на физическом уровне — при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессорной системе).

Иерархия «общее—частное» — показывает, что некоторая абстракция является частным случаем другой абстракции, например, ель — это разновидность хвойных деревьев, а деревья — это часть растительного мира планеты. Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

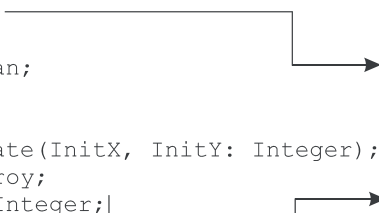
### **Упражнение 1.3.2.** Разработайте класс Точка.

#### **Решение**

Определим новый класс TPoint (точка). Точка определяется координатами  $x$  и  $y$ . Объекты типа TPoint можно сделать видимыми или невидимыми, задать цвет изображения, переместить и т. д.

Полная структура класса TPoint имеет вид:

```
TPoint = class
  Private
    Fx, Fy: Integer;
    Fvisible: Boolean;
    Fcolor: Word;
  Public
    Constructor Create(InitX, InitY: Integer);
    Destructor Destroy;
    Function GetX: Integer;
    Function GetY: Integer;
    Function IsVisible: Boolean;
    Procedure Show;
    Procedure Hide;
    Procedure Move(NewX, NewY: Integer);
end;
```



Совпадают с данными и методами класса TPosition

Поэтому класс TPoint опишем следующим образом:

```
TPoint = class (TPosition)
  Private
    Fvisible: Boolean;
    Fcolor: Word;
  Public
    Constructor Create(InitX, InitY: Integer);
    Destructor Destroy;
    Function IsVisible: Boolean;
    Procedure Show;
    Procedure Hide;
    Procedure Move(NewX, NewY: Integer);
end;
```

TPosition, указанный в скобках после зарезервированного слова class, сообщает компилятору, что TPoint является «потомком» класса TPosition и соответственно наследует все поля и методы этого класса (в частности, поля Fx и Fy, методы GetX и GetY).

Класс TPoint описывает новые поля, определяющие видимость (FVisible) и цвет (FColor), и методы определить видимость (IsVisible), отобразить (Show), спрятать (Hide), переместить (Move).

Конструктор Create и деструктор Destroy переопределяются.

Отметим, что новый класс автоматически получает все данные и методы своих предков, то есть экземпляр класса TPoint содержит все данные (поля) и методы типа TPosition. Таким образом, в иерархичном дереве классов по мере удаления от корня будут встречаться все более сложные классы, экземплярами которых будут объекты с более сложной структурой и поведением.

Доступ к полям и методам, описанным в классе-родителе, осуществляется так же, как к собственным.

**Типизация.** Напомним, тип — это точная характеристика свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов. Типизация — это ограничение, которое накладывается на класс объектов и препятствует взаимозаменяемости различных классов или сильно сужает возможность такой замены.

Использование принципа типизации обеспечивает:

- раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
- упрощение документирования;
- возможность генерации более эффективного кода.

В ООП возможна статическая и динамическая связь имени объекта и его типа. В первом случае это означает определение типов переменных во время компиляции. Во втором — тип выражения определяется во время исполнения приложения. Из принципов динамической связи и наследования вытекает очень важное свойство, присущее объектам — **полиморфизм**. Полиморфизм — это выделение некоторого действия, т. е. действие должно иметь имя, и создание средств использования действия объектами иерархии, причем каждый класс реализует это действие так, как оно для него подходит.

Итак, при создании иерархии классов может обнаружиться, что некоторые свойства объектов, сохраняя название, изменяются по сути.

Для реализации таких иерархий должен быть предусмотрен полиморфизм, обеспечивающий возможность задания различных реализаций некоторого единого по названию метода для классов различных уровней иерархии. В ООП такой полиморфизм называется *простым*, а методы, имеющие одинаковое название, — статическими полиморфными. В ранее рассмотренных упражнениях статическим полиморфным методом является, например, конструктор Create.

Совокупность полиморфных методов с одним именем для иерархии классов образует единый полиморфный метод иерархии, в котором реализация полиморфного метода для конкретного класса представляет отдельный аспект.

*Сложный полиморфизм.* Полиморфными объектами, или полиморфными переменными, называются переменные, которым в процессе выполнения программы может быть присвоено значение, тип которого отличается от типа переменной.

В языках со строгой типизацией такая ситуация может возникнуть

- при передаче объекта типа класса-потомка в качестве фактического параметра подпрограмме, в которой этот параметр описан как параметр типа класса-родителя (явно — в списке параметров или неявно — в качестве внутреннего параметра, используемого при вызове методов — Self);
- при работе с указателями, когда на объект класса-родителя присваивается адрес объекта класса-потомка.

Тип полиморфного объекта становится известным только на этапе выполнения программы, соответственно, при вызове полиморфного метода для такого объекта нужный аспект также должен выполняться на этапе выполнения. Для этого в языке должен быть реализован механизм позднего связывания, позволяющий определять тип объекта и аспект полиморфного метода, к которому идет обращение в программе, на этапе ее выполнения.

С помощью механизма позднего связывания реализуется оперативная перестройка программы в соответствии с типами используемых объектов.

Рассмотрим это свойство на практике.

**Упражнение 1.3.3.** Разработайте класс Окружность.

**Решение**

Определим новый класс TCircle (окружность). Окружность определяется центром с координатами  $x$ ,  $y$  и радиусом  $r$ . Объекты этого типа можно сделать видимыми или невидимыми, задать цвет изображения, переместить и т. д. В связи с этим можно определить новый класс как потомок класса TPoint:

```
TCircle = class (TPoint)
```

```
Private
```

```
Fr: Integer; {радиус}
```

```
Public
```

```
Constructor Create(InitX, InitY, InitR: Integer);
```

```
Destructor Destroy;
```

```
Function GetR: Integer; {возвращает значение радиуса}
```

```
Procedure Show; {спрятать}
```

```
Procedure Hide; {отобразить}
```

```
Procedure Move(NewX, NewY: Integer); {переместить}
```

```
End;
```

Классы TPoint и TCircle связаны отношением наследования и содержат методы Hide (спрятать), Show (отобразить) и Move (переместить). Очевидно, что методы Show и Hide для каждого класса свои, но логика метода Move совпадает:

Переместить:

Спрятать объект;	{вызов метода Hide}
Изменить координаты объекта;	{x:=NewX; y:=NewY}
Отобразить объект;	{вызов метода Show}

Поэтому естественным было бы желание определить метод Move только в TPoint так, чтобы класс-потомок TCircle унаследовал его без определения. Но в методе Move ссылки на методы Hide и Show формируются на стадии компиляции. Это жесткая, статическая связь, и без ее «разрыва», т. е. реализации механизма более позднего формирования ссылок на методы не на стадии компиляции, а на стадии выполнения (динамическое связывание), реализовать это невозможно. Добавление к заголовку метода зарезервированного слова virtual объявляет его виртуальным, т. е. связь с этим методом устанавливает на стадии выполнения программы. Перепишем описание классов следующим образом:

```
TPoint = class (TPosition)
...
Procedure Show; virtual;
Procedure Hide; virtual;
...
end;
TCircle = class (TPoint)
...
Procedure Show; override;
Procedure Hide; override;
...
end;
```

Директива **override** используется для переопределения функциональности метода-предка, она необходима для поддержки полиморфной иерархии.

Реализация динамической связи для объектов, имеющих хотя бы один виртуальный метод, осуществляется с помощью таблицы виртуальных методов (ТВМ). Она содержит адреса виртуальных методов. Для каждого класса во время компиляции программы строится одна ТВМ (рис. 1.3.2).

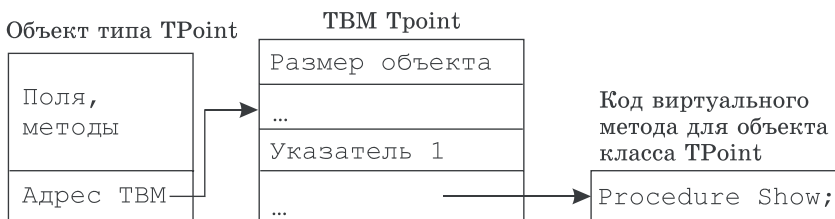


Рис. 1.3.2

Формирование связи между экземпляром класса (объектом) и TBM осуществляет конструктор.

Отметим, что методы, работающие с полиморфными объектами — это всегда методы классов-предков, описывающие общие моменты поведения объектов. В сложной иерархии, таким образом, можно выделить семейство классов со схожим поведением объектов. Они образуют поддеревья, в корне которых находится класс, определяющий общие моменты поведения.

Итак, мы смогли исключить метод Move из описания класса TCircle, сделав его полиморфным. Все объекты классов TPoint и TCircle будут использовать его, причем так, как им это необходимо.

Сформулируем правила, которые важно выполнять при работе с виртуальными методами:

- если в некотором классе метод описан как виртуальный, то все производные классы, включающие метод с тем же именем, должны описать этот метод как полиморфный (override). Нельзя заменить виртуальный метод статическим;
- порядок расположения, количество и типы формальных параметров в одноименных виртуальных методах должны оставаться неизменными.

В дополнении к виртуальным методам, для реализации полиморфизма в Object Pascal используются динамические методы. По возможностям наследования и перекрытия они аналогичны виртуальным методам, но доступ к ним выполняется через таблицу динамических методов (ТДМ). ТДМ хранит адреса только тех динамических методов, которые определены в данном классе. Такой подход позволяет снизить расход памяти при большом количестве этих методов и самих классов.

На каждый динамический метод приходится только одна ссылка, представленная индексом, по которому и происходит поиск метода для вызова.

Для объявления метода динамическим используется директива `dynamic`. Перекрытие динамических методов производится так же, как и виртуальных — с использованием ключевого слова `override`.

### *Абстрактные методы*

**Абстрактные** методы используются при объявлении методов, реализация которых откладывается. Такие методы в классе описываются служебным словом `abstract` и обязательно переопределяются в потомках класса.

Класс, в состав которого входят методы с отложенной реализацией, называется абстрактным. Создавать объекты абстрактных классов запрещается.

**Упражнение 1.3.4.** Разработайте родительский класс для рисования геометрических фигур.

#### **Решение**

Выделим минимальный объем свойств и методов, которые определяют все геометрические фигуры. Во-первых, это точка, относительно которой будет определяться положение фигуры на экране. Во-вторых, это цвет отображаемой геометрической фигуры. Кроме того, определим методы: скрыть, отобразить, переместить геометрическую фигуру. Суть метода переместить остается прежней:

Скрыть;

Задать новое расположение геометрической фигуры;

Отобразить;

Методы скрыть и отобразить для каждой геометрической фигуры будут определять по-своему, поэтому необходимо объявить их виртуальными и абстрактными:

#### **Type**

```
TPosition = class
```

```
private
```

```
  Fx, Fy : integer;
```

```
public
```

```
  constructor Create (InitX, InitY: Integer);
```

```
  function GetX: integer;
```

```
  function GetY: integer;
```

```
  destructor Destroy;
```

```
end;
```

```
TGeometricalFigure = class (TPosition)
private
    Fcolor: word;
public
    constructor Create(InitX, InitY: integer; InitColor: word);
    destructor Destroy;
    function GetColor: word;
    procedure SetColor(NewColor: word);
    procedure Show; virtual; abstract;
    procedure Hide; virtual; abstract;
    Procedure Move(NewX, NewY: integer);
end;
```

**Параллелизм** — свойство нескольких абстракций одновременно находиться в активном состоянии, т. е. выполнять некоторые операции.

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере.

**Процесс** — это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система, имеет много таких потоков: длительность существования одних недолго, а другие живут в течение всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

**Сохраняемость** — это способность абстракции существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

Любой программный объект существует в памяти и живет в течение некоторого времени. Спектр сохраняемости объектов охватывает:

- временные объекты, хранящие промежуточные результаты вычисления выражений;
- локальные объекты, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до ее завершения;

- глобальные объекты, существующие, пока программа загружена в память;
- сохраняемые данные, которые сохраняются в файлах внешней памяти между сеансами выполнения программы.

## 1.4. Композиция и наполнение

В результате объектной декомпозиции второго и далее уровней могут получиться объекты, находящиеся между собой в отношении включения. Классы для реализации таких объектов могут строиться двумя способами: с использованием наследования или композиции.

Наследование применяется тогда, когда разрабатываемый класс имеет с исходным сходную структуру и элементы поведения. В тех случаях, когда сходное поведение не просматривается или наследование по каким-то причинам нецелесообразно, можно использовать композицию классов.

Композицией называется такое отношение между классами, когда один является частью второго. Композиция реализуется включением в класс поля, являющегося объектом другого класса. Такие поля называют объектными.

Включение объектов в некоторый класс можно реализовать и с использованием указателей на объекты, что позволяет включить 0 или более объектов (если они собраны в массив или списковую структуру). Такая реализация класса называется наполнением.

**Упражнение 1.4.1.** Приложение «Бильярд». Спроектируйте классы для написания компьютерной игры в бильярд.

### Решение

Для реализации объектов потребуются классы: `TBilliardTable` (бильярдный стол) и `TCircle` (шар).

Для описания шаров воспользуемся классом `TCircle` (упр. 1.3.3), дополнив его новыми характеристиками: `color` (цвет) и `dx`, `dy` — шаг смещения:

### Type

```
TBall = class (TCircle)
private
  dx, dy : integer;
  color: word;
public
  procedure SetColor(NewColor: word);
  function GetColor: word;
```

```

constructor Create(InitX, InitY, InitR, Initdx, Initdy:
integer; InitColor: word);
end;

```

Бильярдный стол — это прямоугольник, из которого вырезали по сторонам шесть окружностей (луз). На форме располагаются N окружностей (шаров), образующих первоначальную конфигурацию:

### Type

```

TCollection = array[1..100] of TBall;
TBilliardTable = class
private
  x, y, width, height: word;
    {координаты верхнего левого угла и размеры бильярдного стола}
  color: word;                                {цвет стола}
  N: byte;                                    {количество шаров}
  Balls: TCollection;                        {массив указателей на шары}
  Pockets: array[1..6] of TCircle; {массив указателей на лузы}
  procedure SetConfiguration; abstract;
    {начальная расстановка шаров}
public
  constructor Create(InitX, InitY, InitWidth, InitHeight,
NewColor: word; InitN: byte); {конструктор определяет размеры
                               бильярдного стола, расположение луз,
                               вызывает процедуры SetConfiguration и Draw
                               для первоначальной расстановки шаров и изображения стола}
  procedure Draw;                {изображение бильярдного стола}
  procedure Test;                {проверка шаров на столкновение
шаров между собой и со стенками стола, направление движения
шаров при этом изменяется, а также проверка
                               на попадание шаров в лузы}
  procedure Solve;              {выполняет перемещение шаров,
                               уменьшая скорость движения шаров; вызывает процедуру Test }
  destructor Destroy;
end;

```

Таким образом, используя механизм наполнения, в класс TBilliardTable включены объектные поля Balls и Pockets.

## Задания для самостоятельного выполнения

- 1.1. Составьте объектную модель для автоматизации работы библиотеки.
- 1.2. Составьте объектную модель телефонного справочника.
- 1.3. Составьте объектную модель «Процессор», эмулирующую работу процессора ЭВМ.
- 1.4. Составьте объектную модель известной игры «Жизнь», придуманной Дж. Г. Конуэем. Игра моделирует жизнь гипотетической колонии живых клеток. Клетки рождаются, выживают и погибают по определенным правилам. Все поле жизни клеток представляет собой квадратную решетку, каждая ячейка которой может вмещать одну клетку.
  - ☐ Соседями клетки считаются все клетки, находящиеся в восьми ячейках, расположенных с данной по горизонтали, вертикали и диагонали.
  - ☐ Если у клетки меньше двух соседей, то она погибает от одиночества.
  - ☐ Если у клетки больше трех соседей, то она погибает от перенаселения.
  - ☐ Если рядом с пустой ячейкой окажутся ровно три соседние клетки, то в этой ячейке рождается новая клетка.
  - ☐ Гибель и рождение клеток происходит в момент смены поколений.Исходными данными служит начальное расположение клеток.
- 1.5. Составьте объектную модель, демонстрирующую поведение системы «Солнце — Земля — Луна».
- 1.6. Составьте объектную модель, демонстрирующую строение и функционирование Солнечной системы.

## Вопросы для повторения

1. В чем состоит концептуальное отличие структурного программирования от объектно-ориентированного программирования?
2. Что такое класс? Объясните различие терминов класс и объект.
3. Перечислите этапы объектной декомпозиции.
4. Что такое полиморфизм? Приведите пример, демонстрирующий необходимость определения полиморфного метода.
5. В чем сходство и отличие методов, описанных как виртуальные и динамические?
6. Для чего используются абстрактные методы?

# Введение в среду программирования Delphi

## 2.1. Историческая справка

Во времена DOS, которые уже стали историей, программисты стояли перед нелегким выбором между продуктивным, но неэффективным Basic и эффективным, но непродуктивным ассемблером. Появление компилятора Turbo Pascal во многом разрешило их проблемы. Программисты, работающие под Windows 3.1, оказались перед схожей проблемой: что выбрать — мощный, но требующий знаний C++, или простой, но крайне ограниченный Visual Basic (VB). С появлением в 1995 г. Delphi 1 возник новый подход к разработке приложений в среде Windows: простой язык, визуальная разработка приложений, создание откомпилированных выполняемых файлов, динамических библиотек и многое другое. Delphi 1 был первым инструментом разработки Windows-приложений, объединившим в себе оптимизирующий компилятор, визуальную среду программирования и мощные возможности для работы с базами данных.

В 1996 г. появилась версия Delphi 2, предназначенная для разработки приложений для 32-разрядных операционных систем Windows 95 или Windows NT.

Delphi 3 вышла в 1997 г. В этой версии Delphi был расширен набор инструментов для разработки Windows-приложений, упрощено использование таких технологий, как ActiveX и COM, поддержка баз данных с многоуровневой архитектурой.

В 1998 г. появилась версия Delphi 4, главной задачей которой стало упрощение разработки приложений. Появились новые средства навигации в программах и используемых классах. Визуальная среда разработки перепроектирована и дополнена возможностью пристыковывать панели инструментов и окна, что сделало процесс разработки более удобным. Были расширены средства поддержки корпоративных многопользовательских решений.

Выход версии Delphi 5 в свет состоялся во второй половине 1999 г. Ее характеризуют:

- улучшение графической среды разработки и отладчика, пакет поддержки групповой разработки программ и инструменты трансляции;

- набор новых функций, предназначенных для упрощения разработки приложений для Internet;
- стабильность работы системы.

## 2.2. Основные элементы среды программирования Delphi

Для того чтобы запустить Delphi, выполните команду:  
Пуск — Программы — Borland Delphi 5 — Delphi 5 (рис. 2.2.1).

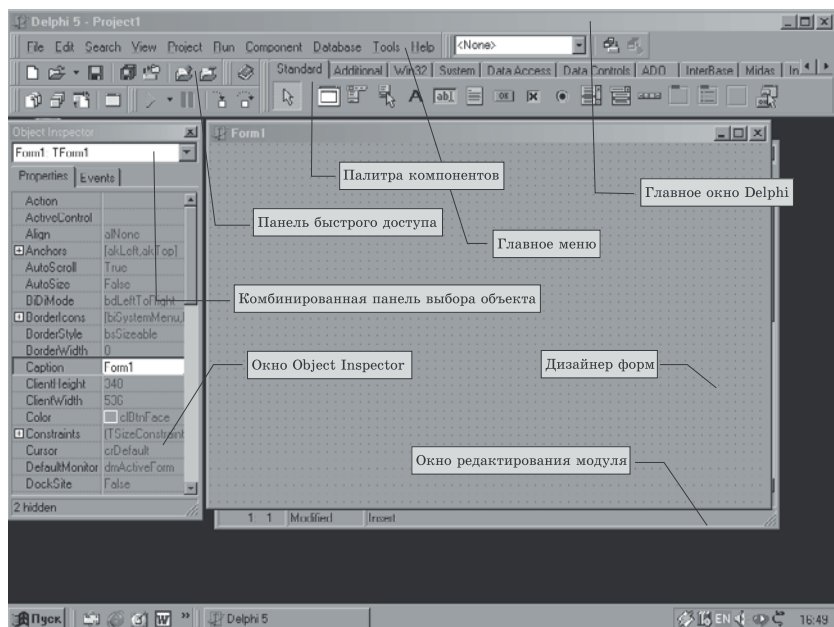




Рис. 2.2.1

В интегрированную среду разработки Delphi входит несколько основных элементов.

**Главное окно Delphi.** Главное окно Delphi содержит главное меню, панель быстрого доступа и палитру компонентов.

Панель быстрого доступа (или палитра инструментов) содержит кнопки, выполняющие некоторые команды меню, например запуск приложения (  ) или переключение между дизайнером форм и окном редактирования модуля (  ).

Палитра компонентов содержит пиктограммы, которые представляют компоненты библиотеки визуальных компонент (VCL).

Главное меню позволяет управлять всеми аспектами работы в Delphi из одного места, так что в Delphi не требуется индивидуального меню для каждого окна.

**Окно Object Inspector.** Отображает свойства (или события) одного или нескольких выбранных компонентов. Окно инспектора объектов состоит из следующих элементов.

Комбинированная панель выбора объекта — это поле, расположенное вверху окна инспектора объектов. На рис. 2.2.1 в комбинированной панели указан объект `Form1: TForm1`.

Страница свойств (Properties). Для каждого компонента определен список свойств, изменение которых приводит к изменению внешнего вида объекта или к изменению реакции на внешние воздействия. Левая колонка страницы свойств содержит имена свойств, а правая — их значения.

Страница событий (Events). Каждый компонент способен реагировать на события, список которых отображен на странице событий инспектора объектов. Левая колонка страницы событий содержит названия, а правая — имя процедуры-обработчика события.

**Дизайнер форм.** Форма — это визуальное изображение окна приложения. Простые приложения имеют только одну форму, а более сложные приложения могут обладать множеством таких форм. Точечная сетка, отображаемая в процессе проектирования приложения, помогает выравнивать помещаемые на форму компоненты. В скомпилированном приложении сетка не отображается.

**Окно редактирования модуля.** Содержит текст модуля на языке Object Pascal, связанный с каждой формой приложения. Delphi автоматически создает этот программный код. Это окно также используется для редактирования других модулей приложения.

## 2.3. Создание приложения

При запуске Delphi автоматически создает новое приложение, с которым Вы можете начать программирование новой задачи. Для удобства работы файлы каждого создаваемого проекта лучше хранить в отдельной папке.

### 2.3.1. Сохранение приложения

Выберите команду File — Save All.

В появившемся диалоговом окне сохранения файла создайте новую папку Exercise1, в котором будут храниться файлы первого проекта, созданного в Delphi. В строке Имя файла введите имя Main. Щелкните на кнопке Сохранить для сохранения файла модуля, Main.pas (рис. 2.3.1.1), и файла формы, Main.dfm.

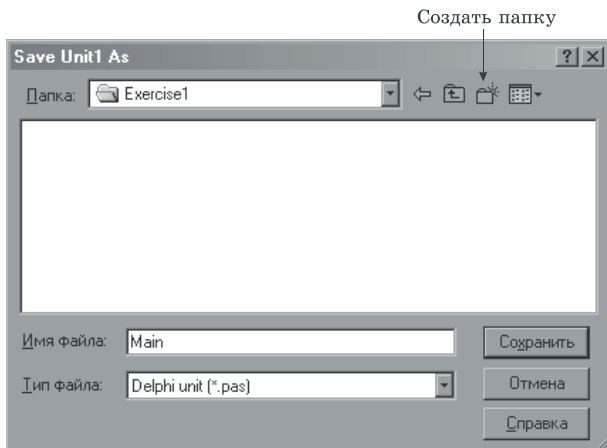


Рис. 2.3.1.1

**Примечание.** Для того чтобы создать папку, щелкните левой клавишей мыши на значке Создать папку в верхней части диалогового окна сохранения файла.

Затем Delphi с помощью еще одного диалогового окна сохранения файла запросит имя проекта. В строке Имя файла введите Exercise1 и выберите Сохранить. Вы создадите файл проекта — Exercise1.dpr.

Используя программу Проводник Windows, убедитесь в том, что в папке Exercise1 были созданы три указанных выше файла.

### 2.3.2. Запуск приложения

Выберите команду Run из меню Run или щелкните на кнопке Run панели быстрого доступа.

Обратите внимание на внешний вид формы. Во время выполнения приложения исчезла точечная сетка, отображаемая в процессе проектирования. Курсор мыши имеет вид стрелки.

Закройте приложение, для этого щелкните дважды на кнопке системного меню в верхнем левом углу окна или на кнопке закрытия окна, расположенной в верхнем правом углу. Если Вы привыкли работать с клавиатурой, то для закрытия приложения и возврата в среду программирования Delphi нажмите Alt+F4.

Если случайно приложение осталось открытым, многие команды Delphi будут недоступными (например окно инспектора объектов).

### 2.3.3. Изменение свойств

Свойства влияют на то, как объект выглядит, и на его невидимые черты (поведение).

Изменение свойств в ходе проектирования — это простой ручной процесс, который включает выбор желаемого объекта, работу со страницей свойств в окне инспектора и изменение значений желаемых свойств.

**Свойство Name.** В Delphi для каждого объекта имеется свойство Name. Когда компонент помещается в форму, Delphi автоматически присваивает ему уникальное имя. Наш компонент форма имеет имя Form1.

В ходе разработки программы происходит частое обращение к объектам по их именам, поэтому осмысленное присвоение имен компонентам избавляет от многих неприятностей.

Назовем форму MainF (описана в модуле Main, буква F говорит о том, что данный объект — это форма (Form)). В окне инспектора объектов выберите свойство Name и в правом столбце наберите MainF. Это изменение сразу же отображается в комбинированной панели выбора объекта инспектора объектов.

**Свойство Color** определяет цвет. В Delphi предусмотрены предопределенные цветовые константы, которые соответствуют многим общеупотребительным цветам. Например, при выборе констант clRed или clYellow цвет формы изменяется соответственно, на красный или желтый. Кроме того, определены константы для представления системных цветов экранных элементов Win32. Например, константы clActiveCaption и clHighLightText соответствуют цветам активных заголовков и выделенного цвета в Win32.

Установите цвет формы равным clGreen.

**Свойство Caption** определяет заголовок формы.

Измените значение этого свойства на значение — Мое первое приложение.

**Свойства Height и ClientHeight** задают высоту формы и высоту рабочей области (исключая рамку и заголовок формы) соответственно. Эти свойства связаны между собой: при изменении значения одного из свойств изменяется и другое. **Свойства Width и ClientWidth** задают ширину формы и ширину рабочей области формы (исключая рамку формы) соответственно. Значения этих четырех свойств можно задать, уменьшая (или увеличивая) размеры формы, используя манипулятор мышь или вводя необходимые значения в правый столбец инспектора объектов.

**Свойства Top и Left** определяют расположение формы на экране, задавая расстояние от верхней границы экрана до верхнего края формы и от левой границы экрана до левого края формы соответственно.

Кроме свойств, изменение значений которых приводит к видоизменению внешнего вида приложения во время проектирования, существуют свойства, изменение значения которых видимо только после запуска приложения.

**Свойство Cursor** определяет графический вид курсора.

В инспекторе объектов измените свойство **Cursor**, выбрав из списка любое значение. Запустите приложение, посмотрите, как изменился вид курсора. Значение свойства **Cursor**, равное **stHelp**, придает курсору вид стрелки со знаком вопроса. Закройте приложение.

Используя свойства **Hint** и **ShowHint**, можно отобразить подсказку. Установите значение свойства **Hint** в «Это форма», а значение **ShowHint** — в **True**. Запустите приложение. Окно подсказки появляется, когда курсор мыши помещается на форму и на мгновение останавливается. Закройте приложение.

## 2.3.4. Обработка событий

Архитектуру программы, выполняющейся в операционной системе Windows, достаточно сложно нарисовать в виде визуальной схемы. Например, пусть программа только что начала работать. На экране показывается главное меню программы, и пользователь нажимает на кнопку мыши. Нажатие на кнопку мыши является событием, начинает работать определенный объект. Пользователь мог бы не нажимать кнопку мыши, и тогда работа программы пошла бы по другому сценарию. То есть ситуация типа: может быть, а может и не быть.

В вероятностной структуре (это структура, в которой не все маршруты, трассы кодов жестко закреплены; в этой архитектуре элементы программ могут включаться при возникновении некоторого события, которое может произойти, а может и не произойти) сильно увеличивается количество возможных состояний и сценариев работы программы. Здесь пользователь определяет порядок работы программы.

Приложения Delphi используют управляемые событиями методы для организации взаимодействия между программой и пользователем. Большая часть кода, которую Вы будете писать в Delphi, будет инициироваться событиями. В принципе событием может быть изменение любой величины. Происхождение события заставляет работать определенный объект, при этом объект получает заранее определенные параметры для своей настройки на работу. В Delphi процедура, инициируемая событием, называется *обработчиком события*.

События делятся на три основные категории: события мыши, события клавиатуры и системные события.

#### 2.3.4.1. События мыши

Для формы событие **OnClick** возникает в том случае, если пользователь нажимает левую кнопку мыши в то время, когда курсор мыши находится на поле формы.

**Упражнение 2.3.4.1.1.** Напишите приложение, которое при нажатии левой кнопки мыши перекрашивает форму в красный цвет.

##### Решение

Создайте обработчик события **OnClick**: выберите страницу **Events** инспектора объектов и выполните двойной щелчок мышью в правой колонке, напротив события **OnClick**. На переднем плане появится окно редактирования модуля с помещенной сразу в нужное место модуля заготовкой обработчика события **OnClick** (рис. 2.3.4.1.1).

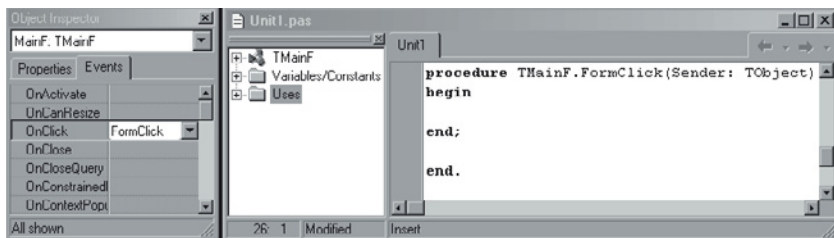


Рис. 2.3.4.1.1

В обработчике события указан параметр `Sender`, в котором хранится имя объекта, породившего данное событие (в примере `Sender` будет содержать ссылку на объект `MainF` — форму).

Для изменения цвета формы в обработчике события `OnClick` напомним следующий оператор:

```
procedure TMainF.FormClick(Sender: TObject);  
begin  
    MainF.Color := clRed; (*)  
end;
```

Поясним оператор (\*). `MainF` — это название объекта формы. `Color` — это свойство формы. Обращение к свойствам объекта происходит так же, как и в структуре данных записи, через точку.

**Эксперимент.** Запустите приложение. Щелкните левой кнопкой мыши на форме. Убедитесь, что цвет формы стал красным.

Что произойдет при повторном щелчке левой кнопкой мыши на форме? Возникнет событие `OnClick`, в результате которого выполнится оператор (\*), т. е. цвет формы остается красным. ♦

**Упражнение 2.3.4.1.2.** Напишите приложение, в котором при каждом щелчке левой кнопкой мыши произвольным образом изменяется цвет формы.

### Решение

Модифицируем код модуля `Main.pas`.

Для получения различных цветов для формы воспользуемся функцией `RGB`. Функция `RGB(Red, Green, Blue)` принимает три параметра - уровни интенсивности красного, зеленого и синего цветов соответственно и возвращает цвет `Win32` как целое значение. Для каждого уровня интенсивности существует 256 возможных значений. Например, `RGB(255, 0, 0)` возвращает цветовое значение для красного цвета, а `RGB(255, 0, 255)` — сиреневого цвета.

Итак, щелчок левой кнопкой мыши по форме порождает событие `OnClick` формы, т. е. приводит к обращению к процедуре `TMainF.FormClick`, в результате чего выполняется оператор (\*) — изменение цвета формы на красный. Изменим оператор (\*) в соответствии с формулировкой задачи:

```
procedure TMainF.FormClick(Sender: TObject);  
begin  
    Color := RGB(Random(256), Random(256), Random(256)); (**)  
end;
```

Сравните правые части операторов (\*) и (\*\*). Оператор (\*) изменяет цвет конкретного объекта типа TMainF (под именем MainF). Оператор (\*\*) позволяет изменить цвет любого объекта типа TMainF, следовательно, этот оператор более общий, и далее будем использовать его при обращении к форме.

**Эксперимент.** Запустите приложение. Убедитесь, что после каждого щелчка по форме левой кнопкой цвет изменяется произвольным образом. ♦

**Упражнение 2.3.4.1.3.** Напишите приложение, в котором при нажатии на левую кнопку мыши происходит смена цвета с зеленого на красный, и, наоборот, с красного — на зеленый.

### Решение

При щелчке по левой кнопке мыши происходит событие OnClick. В соответствии с условием задачи в обработчике события OnClick формы необходимо проанализировать свойство Color формы: если его значение соответствует красному, то изменить его на зеленый (clGreen), иначе присвоить значение красный. На языке Object Pascal это записывается так:

```
procedure TMainF.FormClick(Sender: TObject);  
begin  
  if Color=clRed then Color:=clGreen else Color:=clRed;  
end;
```

**Эксперимент.** Сохраните приложение, запустите его. Проверьте выполнение условия задачи. ♦

**Событие OnDblClick** происходит, если пользователь выполняет двойной щелчок левой кнопки мыши.

**Упражнение 2.3.4.1.4.** Напишите приложение, в котором при двойном щелчке левой кнопкой мыши изменяется вид курсора.

### Решение

Каждое значение свойства Cursor имеет свой числовой эквивалент в пределах от -21 (crHandPoint) до 0 (crDefault). При каждом двойном щелчке левой кнопкой мыши значение свойства Cursor будем увеличивать на единицу, при достижении значения 0 — восстановим значение -21. Введите в обработчик события формы OnDblClick следующий оператор:

```
procedure TMainF.FormDblClick(Sender: TObject);  
begin  
  if Cursor=0 then Cursor:=-21 else Cursor:=Cursor+1;  
end;
```

**Эксперимент.** Сохраните проект и запустите его. Объясните, почему при каждом двойном щелчке левой кнопкой мыши сначала изменяется цвет формы, а затем — вид курсора. ♦

**Событие OnMouseDown** возникает, если пользователь нажимает на правую, левую или среднюю кнопку мыши. Заголовок обработчика этого события имеет вид:

```
procedure TMainF.FormMouseDown(Sender: TObject;  
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

**Событие OnMouseUp** происходит, если пользователь освобождает кнопку мыши, которая была нажата. Заголовок обработчика этого события имеет вид:

```
procedure TMainF.FormMouseUp(Sender: TObject;  
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

**Событие OnMouseMove** происходит, если пользователь перемещает указатель мыши. Заголовок обработчика этого события имеет вид:

```
procedure TMainF.FormMouseMove(Sender: TObject;  
Shift: TShiftState; X, Y: Integer);
```

В параметре `Button` обработчики этих событий получают одно из значений `mbLeft`, `mbRight` или `mbMiddle`, в зависимости от того, какая кнопка мыши была нажата — левая, правая или средняя.

Параметр `Shift` указывает состояние клавиш `Alt`, `Ctrl`, `Shift` и кнопок мыши. Тип `TShiftState` описан так:

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft,  
ssRight, ssMiddle, ssDouble);
```

Значения `ssShift`, `ssAlt`, `ssCtrl` принадлежат множеству `Shift`, если клавиши `Shift`, `Alt`, `Ctrl`, соответственно, были нажаты; `ssLeft`, `ssMiddle`, `ssRight` — если нажаты левая, средняя или правая кнопки мыши; `ssDouble` — если произошел двойной щелчок.

В параметрах `X` и `Y` процедура получает координаты указателя мыши, выраженные в пикселях относительно формы.

**Упражнение 2.3.4.1.5.** Напишите приложение, позволяющее рисовать отрезки на форме.

### Решение

Закройте предыдущий проект, выполнив команду `Close All` меню `File`. Создайте новое приложение, выполнив последовательность команд: `File/ New Application`. Сохраните новое прило-

жение в папке Exercise2. Файл модуля — под именем Main.pas, файл проекта — Exercise2.dpr.

Измените свойства формы следующим образом:

Name	MainF
Caption	Отрезки

Все графические операции в Delphi выполняются с использованием **свойства Canvas**. Это свойство объектного типа. Класс TCanvas — сердцевина графической подсистемы Delphi. Он объединяет в себе контекст конкретного устройства GDI (интерфейс графических устройств) и «рабочие инструменты» (перо, кисть, шрифт, набор функций по рисованию типовых геометрических фигур). В процессе конструирования приложения свойство Canvas недоступно в окне инспектора объектов, оно доступно только во время выполнения приложения. Некоторые свойства и методы класса TCanvas представлены в приложении 1.

Чтобы нарисовать на форме пунктирную линию красного цвета от точки с координатами (10,10) до точки (300,300), создайте обработчик события формы Click и введите следующие операторы:

```
with Canvas do  
begin  
  Pen.Color:=ClRed;  
  Pen.Style:=PsDash;  
  MoveTo(10,10);  
  LineTo(300,300);  
end;
```

Поясним код обработчика события. С помощью свойства Pen можно установить параметры изображения линии, такие, как цвет (Color), тип (Style), толщина (Width).

Метод MoveTo(x,y) перемещает графический курсор в точку с координатами (x,y).

Метод LineTo(x,y) рисует отрезок от текущего положения графического курсора до точки с координатами (x,y).

**Эксперимент.** Сохраните проект. Убедитесь, что после щелчка мышью по форме на канве отображается один отрезок, соединяющий пиксели с координатами (10, 10) и (300, 300). ♦

Модифицируем программу таким образом, чтобы можно было рисовать произвольные отрезки. Отрезок — это часть прямой, заключенная между двумя точками. Пусть нажатие кнопки

мыши послужит началом рисования отрезка (переместим графический курсор в точку, на которую указывает курсор мыши), освобождение кнопки мыши — завершением рисования отрезка (нарисуем линию до точки, на которую указывает курсор). Создайте обработчик события `OnMouseDown`:

```
procedure TMainF.FormMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    Canvas.MoveTo(X, Y);
end;
```

Создайте обработчик события `OnMouseUp`:

```
procedure TMainF.FormMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    Canvas.LineTo(X, Y);
end;
```

***Эксперимент.*** Сохраните приложение. Попробуйте с помощью приложения написать свое имя. ♦

**Упражнение 2.3.4.1.6.** Создайте приложение, позволяющее изображать кривые линии.

### Решение

Закройте предыдущий проект, выполнив команду `Close All` меню `File`. Создайте новое приложение, выполнив последовательность команд: `File/ New Application`. Сохраните новое приложение в папке `Exercise3`. Файл модуля — под именем `Main.pas`, файл проекта — `Exercise3.dpr`.

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Кривые линии

Кривая линия — это изображение следа курсора мыши. Перемещение курсора мыши возбуждает событие `OnMouseMove`. Создайте обработчик этого события и введите следующий оператор:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
begin
    Canvas.LineTo(x, y);
end;
```

**Эксперимент.** Сохраните приложение и запустите его. Убедитесь, что при перемещении курсора мыши на форме изображается линия, началом которой служит левый верхний угол формы. Кроме этого не удастся завершить рисование кривой. ♦

Устраним этот недостаток. Обработчик события OnMouseMove в параметре Shift получает состояние клавиш Ctrl, Alt, Shift и кнопок мыши. Для определенности положим, что линию будем рисовать в том случае, когда нажата левая кнопка мыши. Измените обработчик события следующим образом:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
begin
  if ssLeft in Shift then Canvas.LineTo(x, y)
                                {если нажата левая кнопка мыши, то рисуем линию}
  else Canvas.MoveTo(x, y);
                                {иначе перемещаем графический курсор без следа}
end;
```

**Эксперимент.** Сохраните проект и запустите его. Проверьте, что линия рисуется только тогда, когда перемещение мыши происходит с нажатой левой кнопкой. ♦

**Упражнение 2.3.4.1.7.** Напишите приложение, которое изображает прямоугольники (эллипсы).

### Решение

Закройте предыдущий проект, выполнив команду Close All меню File. Создайте новое приложение, выполнив последовательность команд: File/ New Application. Сохраните новое приложение в папке Exercise4. Файл модуля — под именем Main, файл проекта — Exercise4.

Измените свойства формы следующим образом:

Name	MainF
Caption	Прямоугольники

В классе TCanvas определен метод Rectangle(x1,y1,x2,y2), в котором параметры задают координаты противоположных вершин прямоугольника (т. е. принадлежащих одной из его диагоналей). Нажатие кнопки мыши (событие OnMouseDown) фиксирует начало рисования прямоугольника и соответственно определяет координаты первой точки, а освобождение кнопки (событие OnMouseUp) — окончание рисования прямоугольника и соответственно определяет координаты второй точки. Создайте обработчики этих событий и введите следующие операторы:

```

procedure TMainF.FormMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  x1:=x;
  y1:=y;
end;
procedure TMainF.FormMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Rectangle(x1,y1,x,y); {Canvas.Ellipse(x1,y1,x,y)}
end;

```

Переменные  $x1$  и  $y1$  являются глобальными, но их использование ограничивается пределами данного модуля, поэтому опишите их в разделе `private` класса `TMainF`.

*Эксперимент.* Удостоверьтесь в правильности работы программы. ♦

**Упражнение 2.3.4.1.8.** Напишите приложение, которое позволяет рисовать ломаные линии.

### Решение

Создайте новое приложение и сохраните его в папке `Exercise5`. Файл модуля — под именем `Main.pas`, файл проекта — `Exercise5.dpr`.

Измените свойства формы следующим образом:

Name	MainF
Caption	Ломаные линии

Объединение отрезков  $A_1A_2, A_2A_3, \dots, A_{n-1}A_n$  образуют ломаную  $A_1A_2A_3\dots A_n$ . Начало ломаной — это точка  $A_1$ , конец — точка  $A_n$ . Для рисования ломаной нужно задать ее начало, затем нарисовать звенья (отрезки, причем конец одного является началом другого) и, наконец, завершить ломаную (отметить ее конец).

Начало (конец) рисования ломаной линии свяжем с событием `OnDblClick`. Введем логическую переменную `Draw`: ее значение, равное `True`, будет обозначать рисование ломаной, `False` — завершение ломаной. В разделе `interface` модуля введите раздел констант (после раздела описания переменных — `Var`):

```

const Draw: Boolean = False;

```

Создайте обработчик события `OnDblClick`:

```

procedure TForm1.FormDblClick(Sender: TObject);
begin

```

```
Draw:=not Draw;  
end;
```

В обработчике события `OnMouseDown` будем рисовать звенья ломаной, если значение переменной `Draw` равно `True` и перемещать графический курсор, в противном случае:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:  
TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
  if Draw then Canvas.LineTo(x,y)  
  else Canvas.MoveTo(x,y);  
end;
```

*Эксперимент.* Сохраните проект, используя написанное приложение, попробуйте изобразить треугольник, четырехугольник и пятиугольник. ◆

## Задание для самостоятельного выполнения

- 2.1. Используя код упражнения 2.3.4.1.5, напишите приложения для рисования разноцветных отрезков.
- 2.2. Напишите приложения для рисования эллипсов с закрашенной внутренней областью.
- 2.3. Модифицируйте код приложения из упражнения 2.3.4.1.5 так, чтобы при рисовании отрезок прорисовывался.
- 2.4. Измените код модуля, написанного в упражнении 2.3.4.1.8 для рисования замкнутых ломаных.

### 2.3.4.2. События клавиатуры

Обработку клавиатуры можно выполнить, используя комбинацию трех событий.

**Событие `OnKeyDown`** происходит при нажатии любой клавиши, включая функциональные и специальные:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;  
Shift: TShiftState);
```

Событие `OnKeyPress` возникает при нажатии клавиши, генерирующей символы ASCII, включая управляющие клавиши:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
```

Событие `OnKeyUp` происходит при освобождении любой клавиши:

```
procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;  
Shift: TShiftState);
```

Каждый из обработчиков событий получает, по крайней мере, один параметр, называемый *Key*, который представляет нажатую клавишу. В обработчиках событий *OnKeyDown* и *OnKeyUp* параметр *Key* является беззнаковым значением типа *Word*, которое представляет код виртуальной клавиши *Windows*. В обработчике события *OnKeyPress* параметр *Key* — это значение типа *Char*, представляющее символ *ASCII*. Несмотря на то, что обе переменные называются *Key*, они представляют различную информацию. Все *ASCII*-символы имеют соответствующие коды виртуальных клавиш, но многие виртуальные клавиши не имеют *ASCII*-эквивалента, например клавиша *F1*.

Параметр *Shift* имеет тип *TShiftState* (см. события мыши).

**Упражнение 2.3.4.2.1.** Напишите приложение, закрывающееся при одновременном нажатии клавиш *Alt* и *X*.

### Решение

Создайте новое приложение и сохраните его в папке *Exercise6*. Файл модуля — под именем *Main.pas*, файл проекта — *Exercise6.dpr*.

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Для закрытия приложения нажмите Alt + X

Приложение должно закрываться, если одновременно нажаты обе клавиши, для обработки воспользуемся событием *OnKeyDown*. Создайте обработчик этого события.

Воспользуемся значением параметра *Shift* для определения нажатия клавиши *Alt*:

```
if ssAlt in Shift then Close;
{Close — это метод формы, закрывающий ee}
```

**Эксперимент.** Убедитесь, что нажатие клавиши *Alt* приведет к закрытию формы. ♦

Определим код клавиши *X*. Добавьте в обработчик события *OnKeyDown* оператор:

```
Caption:=IntToStr(Key);
```

Здесь *IntToStr(x)* — это функция, которая преобразует целое число *x* в значение строкового типа, *Key* — код нажатой клавиши.

**Эксперимент.** Запустите приложение. Определите виртуальный код клавиши с латинской буквой «x». ♦

Итак, виртуальный код клавиши с латинской буквой «х» равен 88. Для решения задачи напомним следующий код обработчика события OnKeyDown формы:

```
procedure TMainF.FormKeyDown(Sender: TObject; var Key:
Word; Shift: TShiftState);
begin
  if (ssAlt in Shift) and (Key=88) then Close;
end;
```

**Эксперимент.** Сохраните приложение. Убедитесь, что при одновременном нажатии клавиш Alt и латинской буквы «х» независимо от выбранного языка приложение закрывается. Объясните это. ◆

**Упражнение 2.3.4.2.1.** Напишите программу, которая определяет нажатую на клавиатуре клавишу и выводит в заголовок формы информацию об этой клавише.

Например, при нажатии строчной латинской буквы 'а' в заголовке должна быть надпись:

*«Код виртуальной клавиши=65 а — строчная латинская буква ASCII-код=97»*

(аналогично для русских букв, цифр, специальных символов — запятая, точка, скобки и т. д.); при нажатии клавиши стрелка вправо:

*«Код виртуальной клавиши = 39»*

(аналогично для всех специальных клавиш — F1, Insert и т. д.).

### Решение

Создайте новое приложение и сохраните его в папке Exercise7. Файл модуля — под именем Main.pas, файл проекта — Exercise7.dpr.

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Информация о клавишах клавиатуры

При нажатии любой клавиши клавиатуры возникают события OnKeyDown и OnKeyUp. Для получения информации о коде виртуальной клавиши создадим обработчик события OnKeyDown:

```
procedure TMainF.FormKeyDown(Sender: TObject; var Key:
Word; Shift: TShiftState);
begin
  Caption:='код виртуальной клавиши='+IntToStr(Key);
end;
```

**Эксперимент.** Сохраните приложение. Убедитесь, что при нажатии любой клавиши в заголовке формы отображается виртуальный код этой клавиши. ♦

Для отображения информации о символах воспользуемся обработчиком события `OnKeyPress`, параметр `Key` которого содержит символьное значение, представляющее символ ASCII. Создайте обработчик события `OnKeyPress` и введите оператор:

```
Procedure TMainF.FormKeyPress(Sender: TObject; var Key: Char);
Begin
  Caption:=Key;
End;
```

**Эксперимент.** Сохраните приложение и запустите его. При нажатии на клавишу «F» в заголовке формы отображается символ F. Попробуйте объяснить это. ♦

Это объясняется тем, что при нажатии на клавишу вначале происходит событие `OnKeyDown`, после обработки которого возникает событие `OnKeyPress`, и в результате этого код виртуальной клавиши затирается изображением символа. Измените код следующим образом:

```
procedure TMainF.FormKeyPress(Sender: TObject; var Key:
Char);
begin
  case Key of
    'a'..'z': Caption:=Caption+' '+Key+' – строчная латинская
буква';
    'A'..'Z': Caption:=Caption+' '+Key+' – заглавная латинская
буква';
    '0'..'9': Caption:=Caption+' '+Key+' – цифра';
    'а'..'я': Caption:=Caption+' '+Key+' – строчная русская
буква';
    'А'..'Я': caption:=Caption+' '+Key+' – заглавная русская
буква';
    else Caption:=Caption+' специальный символ';
    end;
    Caption:=Caption+' ASCII-код='+IntToStr(Ord(Key));
end;
```

**Эксперимент.** Сохраните приложение. Проверьте, является ли написанная программа решением поставленной задачи. ♦

## Задания для самостоятельного выполнения

- 2.5. Напишите программу отображения графиков функций:  
 $y=x^2$ ;  $y=\sin(x)$ ;  $y=\operatorname{tg}(x)$ .
- 2.6. Напишите программу отображения графика функции  
 $y=a*\sin(bx)$ . При нажатии на клавиши влево/вправо происходит изменение параметра  $b$ , при нажатии на клавиши вверх/вниз — параметра  $a$ . В заголовок формы выведите название графика функции, например,  $y=3\sin(-2x)$ .
- 2.7. Напишите программу, которая подсчитывает количество нажатий клавиш управления курсором. Например, 'Влево — 1, Вверх — 0, Вправо — 5, Вниз — 37'.
- 2.8. Напишите программу, которая при нажатии на клавиши управления курсором перемещает форму в соответствующем направлении; при нажатии комбинации клавиши <Shift> и клавиши управления курсором — изменяет размеры формы.

### 2.3.4.3. Системные события. Отладка приложения: точки прерывания

События мыши и клавиатуры вызываются воздействиями пользователя на программу. Системные события исходят непосредственно от Windows.

**Событие OnCreate** наступает, когда форма создается.

**Событие OnShow** происходит, когда форма отображается (показана).

**Событие OnActive** происходит, когда форма становится активным окном программы (событие генерируется при получении формой фокуса ввода. Это происходит, когда пользователь возвращается в форму из другой формы того же приложения).

**Событие OnPaint** происходит, когда форму необходимо отобразить заново (перерисовать).

**Событие OnResize** происходит, когда размеры формы изменяются.

**Упражнение 2.3.4.3.1.** Для демонстрации системных событий напомним приложение, содержащее модальное окно.

#### Решение

Создайте новый проект. Сохраните новое приложение в папке Exercise8, файл модуля — под именем Main.dpr, файл проекта — Exercise8.dpr.

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Главная форма

Создайте обработчики событий `OnCreate`, `OnShow` и `OnActivate`, каждый из которых в заголовок формы выводит свое название, например, в процедуре `Form1.FormCreate` напишите

```
Caption:='FormCreate';
```

В обработчике `OnPaint` формы напишите следующие операторы:

```
With Canvas Do
```

```
Begin
```

```
Font.Color:=rgb(Random(255),Random(255),Random(255));
Font.Size:=90;                               {Size=-Height*72/PixelsPerInch}
Font.Style:=[fsBold, fsUnderLine];
Brush.Color:=clBtnFace;
TextOut(30,100,'Delphi')
```

```
end;
```

Свойство `Font` (шрифт), как и свойство `Canvas`, обладает набором собственных свойств и методов. Так, свойство `Size` задает размер символов, а `Style` — стиль начертания, здесь задан полужирный с подчеркиванием.

Добавим в проект еще одну форму, выполнив команду `New Form` меню `File`.

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Вторая форма

Сохраните модуль формы под именем `Modal`.

Напишите обработчики событий `OnCreate`, `OnShow` и `OnActivate`, подобные тем, что написали для первой формы.

В обработчике события `OnPaint` второй формы напишите оператор:

```
Caption:=IntToStr(Random(1000));
```

при возникновении этого события в заголовке формы будет отображаться случайное число из интервала от 0 до 1000.

Подключите модуль второй формы к первой, выполнив команду `Use unit...` меню `File`, в появившемся диалоговом окне выберите `Modal`. Для того чтобы вторая форма отображалась во время выполнения приложения, в обработчике события `OnDoubleClick` первой формы напишите оператор

```
ModalF.Show;
```

Приложение написано. Сохраните файлы проекты, выполнив команду `Save All ...` меню `File`.

Установим точки прерывания.

*1-й способ.* Установите точку прерывания, щелкнув мышью на левом краю окна редактирования. При этом выбранная для остановки строка выделяется красной полосой, на левом краю строчки появляется маленький значок (рис. 2.3.4.3.1).

*2-й способ.* Выполните команду Run/Add Breakpoint/Source Breakpoint... Появится диалоговая панель редактирования точек прерывания Edit Breakpoint с несколькими полями, среди которых не только координаты файла и номер строки (где будет задана точка прерывания), но и некоторые другие полезные данные. Например, Вы можете задать параметр Condition, где ввести выражение, при истинности которого точка прерывания «сработает», иначе выполнение приложения не будет прервано при прохождении через эту строку. Дополнительно можно задать количество проходов, после которых точка прерывания переходит в активное состояние.

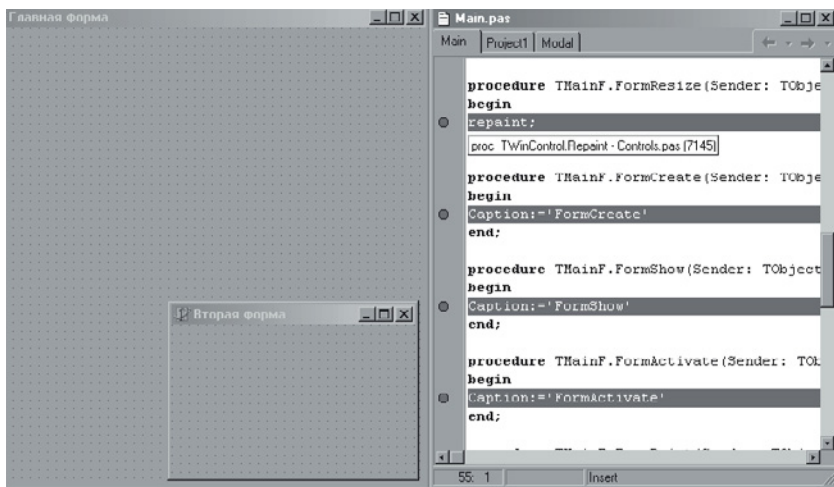


Рис. 2.3.4.3.1

Установите точки останова в обработчике каждого события и расположите формы и окно редактирования так, как показано на рис. 2.3.4.3.1.

**Эксперимент.** Запустите приложение. Проследите за выполнением приложения по шагам. Первая остановка произойдет на процедуре `MainF.FormCreate` (это говорит о том, что произошло событие создания первой формы), продолжите выполнение приложения, выполнив команду `Run` меню `Run`.

Установите, какие события возникают при запуске приложения и в какой последовательности.

Попробуйте переместить форму по экрану, минимизировать и восстановить, изменить ее размеры. Какие события при этом происходят, как изменяется вид окна?

Для вызова второй формы дважды щелкните левой кнопкой мыши по форме. Поэкспериментируйте со второй формой: переместите вторую форму, измените ее размеры, сверните и раскройте окно формы. Что при этом происходит с формами? Попробуйте это объяснить.

Закройте приложение. Добавьте в обработчик события `OnResize` первой формы оператор: `RePaint` (метод `Repaint` порождает событие `OnPaint`, что заставляет форму немедленно перерисоваться).

Запустите приложение и поэкспериментируйте с ним. Какие изменения произошли в поведении формы? Попробуйте объяснить их. ♦

События, генерируемые операционной системой при закрытии окна:

**Событие `OnDeActivate`** происходит, когда форма теряет фокус ввода.

**Событие `OnHide`** происходит, когда форма становится невидимой.

**Событие `OnCloseQuery`** происходит при вызове метода `Close`, или при выборе команды `Close` из системного меню формы, или при щелчке на кнопке закрытия окна. Это событие используется для определения условий, при которых форма может закрыться.

```
procedure TMainF.FormCloseQuery(Sender: TObject;  
var CanClose: Boolean);
```

Логический параметр `CanClose` определяет, может ли форма закрыться или нет; его значение по умолчанию равно `True`.

Например, Вы можете использовать обработчик `OnCloseQuery`, чтобы спросить пользователя, действительно ли он хочет немедленно закрыть форму:

```
procedure TMainF.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
```

```
begin
```

```
  if MessageDlg('Вы действительно хотите завершить  
    работу?', mtConfirmation, [mbOk, mbNo], 0) = mrNo then  
    CanClose:=False;
```

```
end;
```

В этом фрагменте кода использована функция `MessageDlg`, отображающая диалоговое окно в центре экрана. Первый параметр этой функции — это сообщение, которое выводится в окне. Второй параметр — тип окна — может принимать следующие значения: `mtWarning` (предупреждение), `mtError` (ошибка), `mtInformation` (информация), `mtConfirmation` (подтверждение). Третий параметр определяет, какие кнопки должны появиться в диалоговом окне, его значения — `mbYes`, `mbNo`, `mbOK`, `mbCancel`, `mbAbort`, `mbRetry`, `mbIgnore`, `mbAll`, `mbHelp`. Четвертый параметр — для определения темы помощи, которая должна появиться, когда пользователь щелкает на кнопке помощи или нажимает `F1` во время отображения диалогового окна.

**Событие `OnClose`** происходит, когда имело место событие `OnCloseQuery`, в результате чего форма закрывается.

```
procedure TMainF.FormClose(Sender: TObject; var Action: TCloseAction);
```

Значение параметра `Action` определяет, закрывается ли форма фактически. Параметр может принимать следующие значения:

<code>caNone</code>	Форме не разрешается закрываться.
<code>caHide</code>	Форма не закрывается, а просто становится скрытой. Ваше приложение может иметь доступ к скрытой форме.
<code>caFree</code>	Форма закрывается, и вся память, занятая формой, освобождается.
<code>caMinimize</code>	Форма минимизируется, прежде чем закрыться. Это значение параметра установлено по умолчанию для дочерних окон MDI-приложений.

**Событие `OnDestroy`** происходит, когда форма готова исчезнуть навсегда.

### **Задания для самостоятельного выполнения**

- 2.9. Напишите приложение, в заголовке окна которого описан его размер (высота и ширина).
- 2.10. Напишите приложение, в окне которого изображен прямоугольник, стороны которого расположены на расстоянии 10 пикселей от границ окна, независимо от размеров окна.
- 2.11. Создайте приложение с заставкой. Приложение должно состоять из двух форм. На первой форме появляется сообщение о создателе приложения и о том, что при двойном щелчке по форме появится заставка. Окно заставки черного цвета, без заголовка, занимает весь экран. В окне заставки в произвольном месте формы должны появляться эллипсы и прямоугольники разного цвета и размера. Заставка должна исчезать при нажатии любой клавиши клавиатуры.

### **Вопросы для повторения**

1. Каким образом можно изменить свойства компонента?
2. В чем отличие программы, работающей в системе DOS, от программы, работающей в системе Windows?
3. Какие события формы Вы знаете?
4. В каком порядке выполняются события клавиатуры: OnKeyPress, OnKeyUp, OnKeyDown?
5. Благодаря какому классу языка программирования Object Pascal возможна работа с графикой в Delphi? Перечислите свойства и методы этого класса.
6. Какие события происходят при создании формы и в какой последовательности?
7. Какие события происходят при закрытии формы и в какой последовательности?

## Файлы, составляющие приложения Delphi

Приложение Delphi хранится в нескольких различных файлах. Каждое имя файла заканчивается расширением, которое определяет содержимое файла.

Для исследования файлов, составляющих проект Delphi, рассмотрим файлы первого созданного приложения. Запустите Delphi. Откройте файл проекта `Exercisel`, для этого выполните последовательность команд:

**File — Open Project** В диалоговом окне открытия файла выделите файл `Exercisel`. Щелкните на кнопке **Open**.

**View — Units** и затем из списка модулей появившегося диалогового окна выберите все файлы. Щелкните на кнопке **OK**. В окне редактирования модуля отображается два файла. Чтобы выбрать тот или иной, необходимо щелкнуть на соответствующей закладке в верхней части окна.

### 3.1. Файл проекта (.dpr)

Файл проекта представляет собой программу, написанную на языке Object Pascal. Эта программа автоматически создается Delphi. Главный файл проекта изначально называется `Project1.dpr`. Каждое приложение имеет единственный файл проекта. Проект Delphi является Pascal-программой. Проект имеет, по крайней мере, два назначения — объявить модули приложения и запустить приложение. Delphi автоматически создает исходный код файла проекта и управляет им. Вам редко придется модифицировать операторы этого файла.

Рассмотрим структуру файла проекта `Exercisel.dpr`.

```
program Exercisel;

uses
  Forms,
  Main in 'Main.pas' {MainF};

{$R *.RES}

begin
  Application.Initialize;
```

```
Application.CreateForm(TMainF, MainF);  
Application.Run;  
end.
```

В первой строке файла содержится имя проекта — Exercise1. Затем в объявлении `uses` определяются модули, используемые этой программой. В данном случае существует два таких модуля — стандартный модуль `Forms`, который обеспечивает возможности форм Delphi, а также модуль `Main`. Вторая строка объявления `uses` означает, что модуль `Main` расположен в файле `Main.pas`, а `MainF` — это имя объекта формы, описанной в этом модуле.

Директива `{$R *.RES}` является указанием компилятору на необходимость подключения к программе файла ресурсов. Обычно файлы ресурсов содержат только пиктограмму программ, хотя, конечно, они могут иметь и другие типы ресурсов.

И, наконец, файл проекта завершается тремя операторами, расположенными между ключевыми полями `begin...end`. Каждый из них реализует обращение к одному из методов объекта `Application`. В объекте `Application` собраны данные и подпрограммы, необходимые для нормального функционирования Windows-программы в целом. Delphi автоматически создает объект-программу `Application` для каждого нового проекта.

Для того чтобы создать конкретное Windows-приложение с помощью Object Pascal, необходимо создать объект-потомок `TApplication`. Методы и свойства, описанные в `TApplication`, осуществляют создание, выполнение, поддержку и разрушение приложения. Во время написания Windows-приложения `TApplication` является посредником между программистом и окружением Windows.

Метод `Initialize` — первый метод, вызываемый для каждого проекта Delphi, — осуществляет ряд вспомогательных действий, необходимых для работы под управлением операционной системы Windows.

Метод `CreateForm` создает новую форму, тип которой описан в параметре `FormClass`, и связывает ее с переменной, указанной параметром `Reference`:

```
procedure CreateForm(FormClass: TFormClass; var Reference);
```

По умолчанию форма, созданная первым вызовом `FormCreate`, становится главной формой приложения.

Метод `Run` выполняет приложение. Этот метод содержит цикл обработки сообщений и не прекращает свою работу, пока

цикл обработки сообщений не закончен (т. е. реализует бесконечный цикл получения и обработки поступающих от Windows сообщений о действиях пользователя. Когда пользователь щелкнет на кнопке Close, Windows передаст программе специальное сообщение, которое в конечном счете заставит программу прекратить работу и освободить назначенные ей системные ресурсы).

Кроме этих трех методов в классе TApplication описаны многие другие полезные методы и свойства, необходимые при создании проекта.

#### Некоторые свойства класса TApplication:

ExeName	HintHidePause	Icon
Hint	HintPause	ShowHint
HintColor	HintShortPause	Title

#### Некоторые методы класса TApplication:

BringToFront	Minimize
MessageBox	Restore

### Задания для самостоятельного выполнения

- 3.1. Используя справочную систему Delphi, определите назначение каждого из приведенных в таблице свойств.
- 3.2. Модифицируйте код файла проекта. Измените цвет, интервал появления всплывающей подсказки.
- 3.3. Используя справочную систему Delphi, определите назначение каждого из приведенных в таблице методов. Напишите демонстрационное приложение, отображающее работу этих методов.

## 3.2. Файл модуля (.pas)

Модули — это программные единицы, предназначенные для размещения фрагментов программ. С помощью содержащегося в них программного кода реализуется вся поведенческая сторона программы. Эти файлы содержат исходный код приложения на языке Object Pascal. В типичном приложении Delphi для каждой формы создается один файл с расширением pas:

```
unit Main;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls,  
Forms, Dialogs;
```

**type**

```
TMainF = class (TForm)
  procedure FormClick(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
```

**var**

```
MainF: TMainF;
```

**implementation**

```
{ $R *.DFM }
```

```
procedure TMainF.FormClick(Sender: TObject);
```

**begin**

```
  Color:=clRed
```

```
end;
```

```
end.
```

В первой строке Main.pas указывается имя модуля — Main. Delphi присваивает модулю то имя, которое указывается при сохранении файла модуля проекта.

Далее следует директива `interface`, которая объявляет описания, видимые пользователю. Ко всем элементам, описанным в этом разделе, разрешено обращение из других модулей, использующих данный модуль. В этом разделе возможно подключение модулей, описание новых типов, констант, переменных, процедур и функций.

Интерфейсная часть модуля формы содержит три части — подключение модулей, создание нового типа и объявление переменной объектного типа, представляющей форму.

Модуль Main использует много различных модулей, все они представлены в операторе `uses`. Такие модули, как Windows и Messages, обеспечивают необходимые данные и программный код для интерфейса прикладных программ Windows, а модули Classes, Graphics, Controls — для формы, компонентов и других элементов программы.

Далее идет раздел объявления типов, определяемый ключевым словом `type`. Тип TMainF задает описание класса (class). Класс TMainF в Main.pas наследует члены класса TForm, который поддерживает в Delphi функциональные возможности формы. Таким образом, класс TMainF является потомком класса TForm.

Класс TMainF объявляет процедуру FormClick(Sender: TObject), которая обрабатывает действие — щелчок мышью на форме.

Классы могут иметь дополнительные объявления, которые бывают либо собственными (private), либо общедоступными (public), т. е. доступны также операторам других модулей.

Завершает раздел интерфейса описание переменных, предвзяемое словом var. Модули могут содержать одну или несколько переменных, но могут обходиться и без них. В модуле Main — только одна переменная:

```
var MainF: TMainF;
```

Эта строка объявляет объект-переменную MainF класса TMainF, т. е. MainF — это занимающий определенный участок памяти экземпляр класса. В этом примере объект представляет собой главное окно программы, которая содержит обработчик события, выполняющий действия при щелчке на форме.

За разделом интерфейса идет раздел реализации (implementation), в котором описаны элементы, скрытые от пользователя. Этот раздел может также содержать разделы подключения модулей, описание типов, переменных, констант, кроме этого он содержит программный код подпрограмм, указанных в интерфейсной части модуля. К элементам, указанным только в разделе implementation, запрещены обращения из других модулей, использующих данный модуль.

Первая строка раздела implementation содержит директиву

```
{ $R *.DFM }
```

которая открывает и читает файл с расширением .DFM. Звездочка указывает Delphi на необходимость поиска файла с именем, совпадающим с названием файла модуля, но имеющим расширение .DFM. Этот файл содержит свойства формы, измененные с помощью окна Object Inspector.

В последней строке файла находится ключевое слово end, за которым следует точка.

Итак, в секции интерфейсных объявлений описываются программные элементы, которые будут «видны» другим программным модулям, а в секции реализаций раскрывается механизм работы этих элементов. Разделение модуля на две секции обеспечивает удобный механизм обмена алгоритмами между отдельными частями одной программы. Он также реализует средство обмена программными разработками между отдельными программистами. Получив откомпилированный «посторонний» мо-

дуль, программист получает доступ только к интерфейсной части, в которой, как уже говорилось, содержатся объявления элементов. Детали реализации объявленных процедур, функций и классов скрыты в секции реализаций и недоступны другим модулям.

### Задания для самостоятельного выполнения

- 3.4. Разработайте модуль, содержащий иерархию классов: список, стек, очередь.
- 3.5. Запустите приложение, созданное в упр. 2.3.4.1.5. Нарисуйте несколько отрезков. Сверните окно приложения. Восстановите окно. Отрезки исчезли. Это происходит потому, что объект Canvas «не умеет» сохранять изображения. Устраните этот недостаток, используя стек, реализованный при выполнении предыдущего задания.

## 3.3. Файл формы (.DFM)

Файл формы используется для сохранения информации о внешнем виде главной формы, содержит значения в двоичном формате, представляющие свойства формы, а также свойства любых компонент, находящихся на форме. В файлах .DFM также фиксируются взаимоотношения между событиями и обработчиками событий. Delphi копирует эту информацию в исполняемый Exe-файл. Чтобы посмотреть информацию файла формы в текстовом виде, выполните команду View as Text контекстного меню формы (рис. 3.3.1):

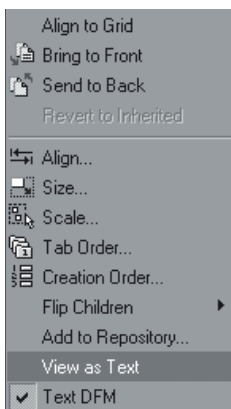


Рис. 3.3.1

```
object MainF: TMainF
  Left = 200
  Top = 108
  Width = 544
  Height = 375
  Hint = 'Это форма'
  Caption = 'Мое первое приложение'
  Color = clYellow
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  ShowHint = True
  OnClick = FormClick
  PixelsPerInch = 96
  TextHeight = 13
end;
```

Чтобы вернуться к графическому режиму отображения формы, выполните команду View as Form контекстного меню.

### 3.4. Дополнительные файлы приложения Delphi

Мы рассмотрели основные файлы проекта. Рассмотрим дополнительные файлы, создаваемые Delphi. Используя Проводник Windows, откройте папку Exercise1 (рис. 3.4.1).

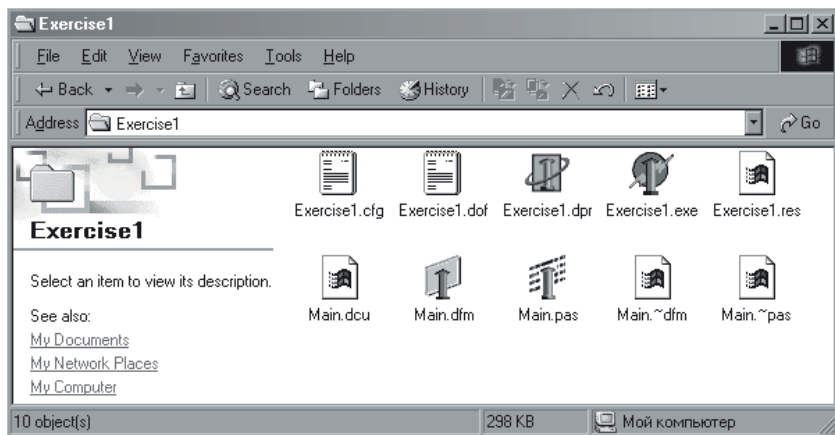


Рис. 3.4.1

**Файл .DCU** содержит скомпилированный код модуля. Например, файл `main.dcu` содержит программный код и данные, объявленные в модуле `main.pas`. Вы можете безо всякого риска удалять файлы `.dcu`, поскольку Delphi создаст их снова при компиляции приложения.

**Файл .RES.** Файл ресурсов содержит в двоичном формате такие ресурсы, как пиктограмма программы и другие растровые изображения. Для того чтобы создавать и модифицировать файлы ресурсов, воспользуйтесь командой Image Editor меню Tools. Никогда не вносите изменений в файл ресурсов проекта, имя которого совпадает с именем проекта, но заканчивается расширением `.RES`. При перекомпиляции Delphi все внесенные вами ресурсы исчезнут.

**Файл .DSK** сохраняют конфигурацию рабочей области приложения, если включен режим Desktop AutoSave с помощью команды environment меню option. Файлы `.DSK` сохраняют информацию об окнах Delphi и порядке их расположения, а также каталоги хранения файлов проекта. Файлы `.DSK` сохраняют также маршруты проекта, поэтому если вы переносите свои проекты в другие папки или сохраняете модули проекта в нескольких папках, то не следует удалять файлы `.dsk`.

**Файл .DOF** сохраняет параметры, установленные с помощью команды Option... меню Project. Содержание файла — текущие установки проекта, такие, как параметры компилятора, командной строки, директории и другие. Если вы удалите файл с расширением `dof`, Delphi создаст его снова, используя работающие по умолчанию параметры. Будьте осторожны при удалении файла опций проекта, так как могут возникнуть проблемы для приложений с нестандартной настройкой.

**Эксперимент.** Используя диалоговое окно Project Options, измените иконку приложения, отображаемую в Проводнике Windows; внесите информацию о названии компании, описание файла, комментарии. Проследите изменения в файле опций. Для создания иконки воспользуйтесь программой Image Editor, поставляемой вместе с Delphi. ♦

**Исполняемый файл (.EXE).** Файлы, заканчивающиеся на `.~*`, являются резервными копиями модифицированных или сохраняемых файлов. Можно спокойно удалять эти файлы в любое время, хотя они могут понадобиться, чтобы восстановить потерянные или разрушенные программы или вернуться к предыдущим версиям.

**Вопросы для повторения**

1. Какие файлы составляют проект Delphi? Какие файлы проекта нельзя удалять ни в коем случае? Объясните.
2. Объект Application. Для чего он предназначен? Перечислите основные свойства и методы этого объекта.
3. Опишите общую структуру модуля в Object Pascal.
4. Каким образом обеспечивается сокрытие информации при описании класса?
5. Для чего предназначен файл формы?

# Введение в визуальное проектирование

## 4.1. Визуальное проектирование

Delphi, являясь визуальной средой разработки приложений, ориентирована на тех программистов, которые из готовых компонент «собирают» конкретные приложения для конечных пользователей. Визуальные средства Delphi построены на концепции двойного инструментария (Two-Way Tools), позволяющей изменять свойства объектов как в процессе визуального конструирования на этапе «сборки» (Design time), так и программно, в процессе работы приложения (Run time). В Delphi эта концепция реализуется с помощью компонентов.

С другой стороны, являясь расширяемым объектно-ориентированным инструментарием, Delphi позволяет создавать собственные компоненты.

Компонент — это оформленный специальным образом класс. Его свойства могут меняться на этапе «сборки» программы с помощью Инспектора объектов (Object Inspector). Все изменения сразу же отображаются на экране монитора, поэтому такой подход позволяет быстро создавать интерфейсную часть приложения. Как известно, эта часть занимает обычно до 80% работы программиста.

Компоненты Delphi объединены в библиотеку VCL — Visual Component Library (библиотека визуальных компонентов). Все компоненты Delphi можно классифицировать следующим образом:

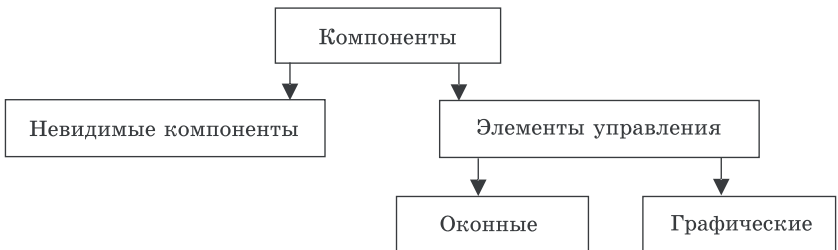


Рис. 4.1.1

Невидимые (невизуальные) компоненты представляют собой, как правило, компоненты, с помощью которых осуществ-

ляется доступ к системным ресурсам, например к системному таймеру. Они отображаются во время конструирования интерфейса, но не видны во время работы приложения.

Элементы управления дают возможность пользователю во время работы программы получать информацию и/или управлять работой программы.

Оконные элементы, визуальные компоненты управления могут принимать фокус ввода (становятся выделенными), они являются окнами системы Windows и обладают всеми свойствами окна, т. е. имеют уникальный идентификатор Windows и получают сообщения от операционной системы.

Графические компоненты отличаются от предыдущих тем, что не имеют идентификатора и, соответственно, не могут получать сообщения от Windows. Они отнимают у системы гораздо меньше ресурсов, чем все другие.

Иерархия классов Delphi представляет собой удачно подобранную иерархию базовых классов. Во многом это объясняется тем, что традиционно в среде Windows было достаточно сложно реализовывать пользовательский интерфейс. Событийная модель в Windows всегда была сложна для понимания и отладки. Но именно разработка интерфейса в Delphi является самой простой задачей для программиста.

Среда Delphi включает полный набор визуальных инструментов для быстрой разработки приложений (Rapid Application Development — RAD), поддерживающей разработку пользовательского интерфейса и подключение к корпоративным базам данных. VCL — библиотека визуальных компонент — включает стандартные объекты построения пользовательского интерфейса, объекты управления данными, графические объекты, объекты мультимедиа, диалоги и объекты управления файлами, управление OLE.

Палитра компонентов (рис. 4.1.2) позволяет выбрать нужные объекты для размещения их на Дизайнере форм. Для использования Палитры компонент просто первый раз щелкните мышкой на одном из объектов в палитре компонент и потом второй раз — на Дизайнере форм. Выбранный объект появится на проектируемом окне, и им можно манипулировать с помощью мыши.



Рис. 4.1.2

Палитра компонентов использует постраничную группировку объектов. Вверху палитры находится набор закладок — Standard, Additional, Dialogs и т. д. Для выбора страницы палитры компонентов используются стрелки перемещения, расположенные в правом верхнем углу.

## 4.2. Компоненты Label, Edit, Button

Рассмотрим некоторые компоненты, расположенные на странице Standard.

Используя всплывающую подсказку, найдите компонент **Button**. Щелкните на нем мышью, затем щелкните мышью на Дизайнере форм.

**Примечание.** Для того чтобы появились всплывающие подсказки, вызовите контекстное меню Палитры компонентов и установите флажок Show Hints (рис. 4.2.1).

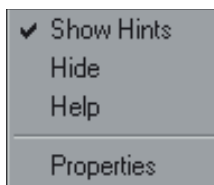


Рис. 4.2.1

В комбинированной панели выбора объекта (рис. 2.2.1) Инспектора объектов появился еще один объект — Button1. В Инспекторе объектов перечислены свойства, доступные во время проектирования приложения. Некоторые свойства (такие, как, Caption, Name, Left, Top, Width, Height, Cursor) вам уже известны, они совпадают со свойствами формы.


Перечислим свойства компонента **Button**, которые будем использовать при создании приложений:

Cancel	значение True этого свойства связывает нажатие клавиши Escape с обработчиком события OnClick;
Default	значение True этого свойства связывает нажатие клавиши Enter с обработчиком события OnClick;
Enabled	значение True этого свойства обеспечивает доступность компонента для мыши, клавиатуры и событий таймера;

Font	контролирует отображение текста, отображаемого на компоненте. Это свойство объектного типа, некоторые его элементы приведены при описании компонента формы;
Visible	определяет видимость компонента во время выполнения приложения.

#### События компонента Button:

OnEnter	случается, когда компонент получает фокус ввода;
OnExit	случается, когда фокус ввода перемещается на другой компонент.

Положите на форму компонент **Edit** (окно редактирования, ). Компонент Edit используется для получения текста от пользователя и для его отображения.

#### Свойства компонента Edit:

AutoSelect	значение True этого свойства обеспечивает выделение текста, помещенного в Edit, при получении объектом фокуса ввода;
BorderStyle	определяет границу между внутренней частью компонента и клиентской областью приложения. Имеет два значения: bsNone (нет границы) и bsSingle (есть граница);
CharCase	определяет регистр символов, отображаемых в компоненте;
Color	цвет фона компонента;
MaxLength	ограничивает количество символов, которые можно ввести в edit;
PasswordChar	используется для скрытия вводимой информации. Если значение свойства равно #0, то текст отображается нормально. Любое другое значение этого свойства определяет отображаемые в Edit символы;
ReadOnl	значение свойства, равное True, запрещает редактирование текста, отображаемого компонентом;
Text	свойство используется для чтения текста или определения нового значения текста.

При изменении содержимого компонента Edit происходит событие OnChange.

Положите на форму компонент **Label** (метка, **A**). Компонент **Label** — это графический элемент управления, предназначенный для отображения текста на форме.

#### Свойства компонента **Label**

<b>Align</b>	определяет расположение объекта: <b>alNone</b> (заданное пользователем), <b>alTop</b> (верхняя часть), <b>alBottom</b> (нижняя часть), <b>alLeft</b> (левая часть), <b>alRight</b> (правая часть), <b>alClient</b> (вся область);
<b>Alignment</b>	горизонтально выравнивание текста: <b>taLeftJustify</b> (по левому краю), <b>taRightJustify</b> (по правому краю), <b>taCenter</b> (по центру);
<b>AutoSize</b>	значение свойства, равное <b>true</b> , приводит к автоматическому изменению размеров метки в соответствии с длиной текста;
<b>Caption</b>	Определяет строковую константу, отображаемую в компоненте;
<b>LayOut</b>	вертикальное выравнивание текста в метке: <b>tlTop</b> (по верхнему краю), <b>tlCenter</b> (по центру), <b>tlBottom</b> (по нижнему краю);
<b>WordWrap</b>	значение свойства, равное <b>true</b> , обеспечивает автоматический перенос строк при отображении текста.

При необходимости узнать подробнее о свойствах, методах и событиях компонента можно воспользоваться справочной системой. Справочная система является контекстно-зависимой; при нажатии клавиши **F1** открывается подсказка, соответствующая текущей ситуации. Например, находясь в Инспекторе Объектов, выберите какое-нибудь свойство и нажмите **F1**, отобразится справка о выделенном свойстве.

#### Задания для самостоятельного выполнения

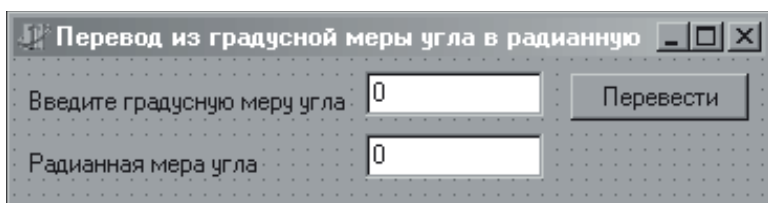
- 4.1. Проследите за изменением состояния компонентов **Label**, **Edit** и **Button** при изменении свойств, перечисленных выше.
- 4.2. Поместите в правый нижний угол формы кнопку на расстоянии 5 пикселей от ее края. Создайте обработчик события, который при изменении размеров формы сохраняет местоположение кнопки.

**Упражнение 4.2.1.** Напишите программу для перевода градусной меры угла в радианную.

## Решение

Создайте новый проект. Сохраните новое приложение в папке Calculator — файл модуля под именем Main.pas, файл проекта — Angle.dpr.

*1-й этап. Создадим визуальный интерфейс приложения (рис. 4.2.2)*



**Рис. 4.2.2**

Измените значения свойств формы следующим образом:

Name	MainF
Caption	Перевод из градусной меры угла в радианную

Поместите на форму компонент Label, измените значения свойств следующим образом:

Name	DegreeLbl
Caption	Введите градусную меру угла

Поместите на форму компонент Edit, измените значения свойств следующим образом:

Name	DegreeEdt
Text	0

Поместите на форму компонент Button, измените значения свойств следующим образом:

Name	ExecuteBtn
Caption	Перевести

Поместите на форму компонент Label, измените значения свойств следующим образом:

Name	RadianLbl
Caption	Радианная мера угла

Поместите на форму компонент Edit, измените значения свойств следующим образом:

Name	RadianEdt
Enabled	False
Text	0

Измените размеры формы в соответствии с расположенными компонентами (рис. 4.2.2).

### *2-й этап. Создание программного кода*

Создайте обработчик события OnClick кнопки ExecuteBtn. Нужно взять информацию, введенную в компонент DegreeEdt, преобразовать в соответствии с математической формулой:

Радиянная мера угла = Градусная мера угла \* ПИ / 180)  
и отобразить результат в компоненте RadianEdt.

При выполнении этих манипуляций может возникнуть проблема преобразования форматов данных. Исходные данные (градусная мера угла — это вещественное число) представлены строкой, хранящейся в компоненте DegreeEdt. Для преобразования строки в вещественное число воспользуемся процедурой Val:

```
procedure Val(S; var V; var Code: integer),
```

где S — строковое представление числа, V — параметр, через который процедура возвращает число целого или вещественного типа, Code — если строка S не является представлением числа, то параметр Code содержит номер ошибочного символа.

Итак, в обработчике события OnClick кнопки ExecuteBtn введем следующую последовательность операторов:

```
procedure TMainF.ExecuteBtnClick(Sender: TObject);
```

```
var Angle_degree, Angle_radian: real;
```

```
    Error: integer;
```

```
begin
```

```
    val(DegreeEdt.text, Angle_degree, Error);
```

```
    if Error = 0 then
```

```
begin
```

```
    Angle_radian:= Angle_degree*Pi/180;
```

```
    RadianEdt.text:=FloatToStr(Angle_Radian);
```

```
    end
```

```
    else
```

```
begin
```

```
    Application.MessageBox('Ошибка при вводе градусной меры угла',  
    'Ошибка', MB_OK);
```

```
    DegreeEdt.text:='0';
```

```
    RadianEdt.text:='0';
```

```
    end;
```

```
end;
```

Поясним операторы обработчика события OnClick.

Первый оператор преобразует введенную в строковом формате градусную меру угла в вещественное число.

Если параметр Error равен нулю, то ошибок при введении числа нет. В этом случае получаем радианную меру угла, функция FloatToStr преобразует вещественное число в строку. В противном случае воспользуемся методом объекта Application для отображения диалогового окна об ошибке и восстановим нулевые значения в компонентах DegreeEdt и RadianEdt.

Сохраните изменения, внесенные в проект.

**Эксперимент.** Запустите приложение. Убедитесь в корректной работе созданного приложения. ♦

**Упражнение 4.2.2.** Напишите программу, выполняющую арифметические действия над целыми числами.

### Решение

Создайте каталог Calculator, файл проекта сохраните под именем Calculator.dpr, а файл модуля — Main.pas.

*1-й этап.* Как уже говорилось выше, первым этапом решения задач в среде Delphi является визуальное проектирование.

Измените заголовок формы (свойство Caption) на 'Калькулятор', а значение свойства Name — на CalculatorFrm.

Поместите на форму компонент Edit. Расположите его так, как показано на рис. 4.2.3. Установите следующие значения его свойств:

Enabled:=False;

{чтобы предотвратить ввод символов,  
которые не могут присутствовать в числе }

Name:=NumberEd;

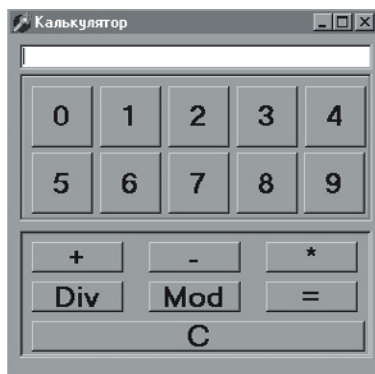


Рис. 4.2.3

Поместите на форму компонент Panel, который предназначен для логического объединения компонентов в группу.

Объемный вид компоненту Panel придает фаска (Bevel), состоящая из двух частей — внутренней (Inner) и внешней (Outer). На рис. 4.2.4 значение свойства BevelInner установлено в bvLowered, а BevelOuter — в bvRaised.



Рис. 4.2.4

Чтобы компонент Panel1 выглядел так, как на рис. 4.2.3, установите следующие значения свойств:

```
BevelInner:=bvRaised;  
BevelOuter:=bvRaised;  
BevelWidth:=1;                                     {ширина фаски}  
BorderStyle:=bsSingle;                             {стиль рамки}  
Caption:='';
```

Поместите на панель 5 компонентов Button. Для этого, удерживая клавишу Shift, щелкните на компоненте Button в палитре компонент — компонент стал выделенным. Щелкните на панели пять раз — на панели появились пять кнопок (рис. 4.2.5).

Установим размер кнопок, равный 60 пикселям по ширине и 50 — по высоте. Выделите все помещенные на панель кнопки (рис. 4.2.5).

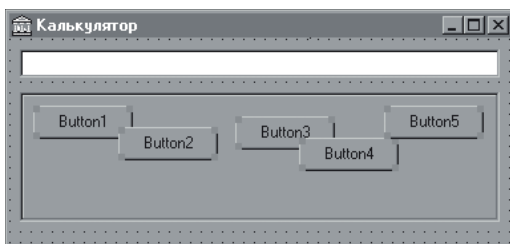


Рис. 4.2.5

Выполните команду Size меню Edit. В появившемся диалоговом окне Size установите размер объектов (рис. 4.2.6).

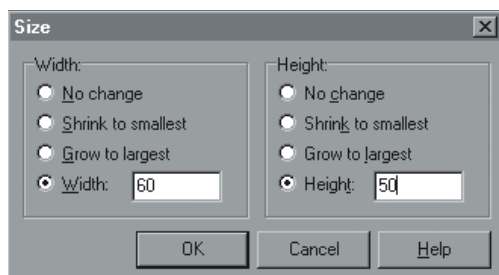


Рис. 4.2.6

**Примечание:** Значение No Change не изменяет размеров объектов; Shrink to smallest выравнивает размер по ширине (высоте) наименьшего из выделенных объектов; Grow to Largest выравнивает размер по ширине (высоте) наибольшего из выделенных объектов; Width (Height) устанавливает ширину (высоту) отмеченных объектов в указанные значения.

Чтобы выровнять кнопки относительно друг друга, выполните команду Align... меню Edit. Установите выравнивание по верхней стороне по вертикали и равные расстояния между компонентами по горизонтали (рис. 4.2.7).

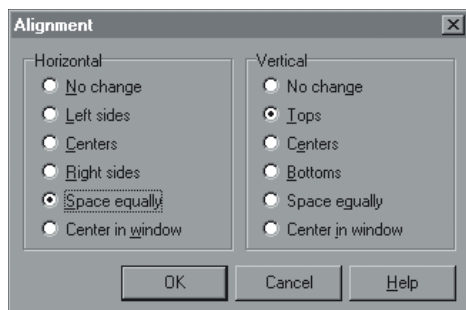


Рис. 4.2.7

**Примечание.** Значение No change не изменяет выравнивание компонентов; Left sides (Right sides, Tops, Bottoms) выравнивает левые (правые, верхние, нижние) границы выделенных компонентов; Space equally располагает компоненты на равном расстоянии друг от друга; Center in Window выравнивает компоненты по центру формы; Centers выравнивает центры компонентов.

Скопируйте выделенные пять кнопок в буфер обмена, выполнив команду Copy меню Edit. На Panel1 поместите еще пять кнопок, для этого выделите компонент Panel1 (чтобы показать объект, на который будем помещать кнопки из буфера) и выполните команду Paste меню Edit (вставка из буфера обмена). Не снимая выделения, разместите эти кнопки как показано на рис. 4.2.3.

Поместите в нижнюю часть формы еще один компонент Panel. Разместите на нем семь компонентов Button (рис. 4.2.3) и измените их свойства Caption и Name в соответствии с таблицей:

Caption	Name	Caption	Name	Caption	Name
0	ZeroBtn	5	FiveBtn	–	SubtractBtn
1	OneBtn	6	SixBtn	*	MultiplyBtn
2	TwoBtn	7	SevenBtn	Div	DivBtn
3	ThreeBtn	8	EightBtn	Mod	ModBtn
4	FourBtn	9	NineBtn	=	EqualBtn
		+	AddBtn	C	CancelBtn

Измените шрифт кнопок, для этого выделите все кнопки и измените значение свойства Font.

Итак, визуальное проектирование калькулятора закончено.

### *2-й этап. Создание программного кода*

При нажатии на клавиши '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' формируется число — к содержимому окна редактирования справа добавляется цифра, отображенная на кнопке. Создадим обработчик события OnClick кнопки OneBtn:

```
procedure TCalculatorFrm.OneBtnClick(Sender: TObject);  
begin  
    NumberEd.Text:=NumberEd.Text+OneBtn.Caption  
end; (1)
```

Но при щелчке на любой из оставшихся 9 кнопок происходит то же самое действие. Обобщим описанный выше обработчик события:

```
procedure TCalculatorFrm.OneBtnClick(Sender: TObject);  
begin  
    NumberEd.Text:=NumberEd.Text+(Sender as TButton).Caption  
end; (2)
```

Напомним, что параметр `Sender` представляет объект, который получил сообщение, например, при нажатии на клавишу '1' параметр `Sender`, несмотря на указанный в заголовке тип `TObject`, является ссылкой на объект `OneBtn` типа `TButton`. Оператор `AS` используется для приведения объектных типов, т. е. параметр `Sender` (который был сужен до класса `TObject`) будет преобразован (расширен) к типу `TButton`. Поэтому при щелчке на кнопке `OneBtn` значения `OneBtn.Caption` обработчика (1) совпадает со значением `(Sender as TButton).Caption` обработчика (2).

Таким образом, если после щелчка на кнопке `TwoBtn` обратиться к методу-обработчику события (2), то значение `(Sender as TButton).Caption` будет равняться '2'. Чтобы связать этот обработчик события с обработкой события `OnClick` компонента `TwoBtn`, необходимо в левом столбце Инспектора объектов компонента `TwoBtn` из списка методов-обработчиков событий события `OnClick` выбрать `OneBtnClick` (рис. 4.2.8).

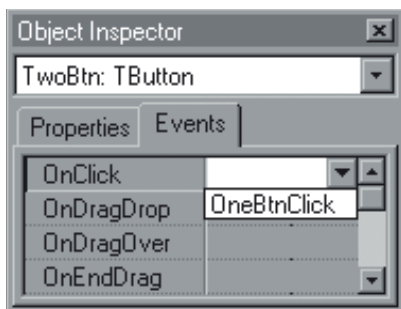


Рис. 4.2.8

Выполните аналогичные действия для каждой из оставшихся восьми кнопок. Запустите приложение.

**Эксперимент.** Запустите приложение. Убедитесь, что при щелчке по кнопкам формируется число. ◆

При щелчке на кнопках '+', '-', '\*', 'Div', 'Mod' необходимо запомнить первый операнд арифметического выражения и знак операции.

Введем переменную `operand1` (`operand1: Integer`) для хранения значения первого операнда вычисляемого выражения, опишем ее в разделе `private` класса `TCalculatorFrm`.

Каждый компонент VCL обладает свойством `Tag`, назначение которого устанавливает программист. Мы можем воспользоваться этим свойством для запоминания (определения) знака арифметической операции. Установим значение свойства `Tag` компонентов в соответствии со следующей таблицей:

Name	Tag
AddBtn	1
SubtractBtn	2
MultiplyBtn	3
DivBtn	4
ModBtn	5

При выборе знака действия запомним `Tag` соответствующей кнопки в свойстве `Tag` формы. Тогда обработчик события `OnClick` компонента `AddBtn` будет выглядеть так:

```
procedure TCalculatorFrm.AddBtnClick(Sender: TObject);
    {метод-обработчик события OnClick кнопок,
     задающих знаки арифметических действий}
begin
    Operand1:=StrToInt(NumberEd.Text);
    {переменная operand1 содержит значение первого операнда}
    NumberEd.text:=''; {очистите содержимое строки редактирования}
    Tag:=(Sender as TButton).Tag
    {в свойство Tag формы скопируем значение
     свойства Tag компонента, возбудившего событие}
end;
```

Соедините данный обработчик события с обработчиками событий `OnClick` оставшихся четырех знаков действий.

При нажатии на клавишу '=' вычисляется результат арифметической операции:

```
procedure TCalculatorFrm.EqualBtnClick(Sender: TObject);
var Operand2, Result: integer;
begin
    Operand2:=StrToInt(NumberEd.Text);
    {переменная operand2 содержит значение второго операнда}
case Tag of {в зависимости от значения свойства Tag
    формы вычислим значение арифметического выражения}
    1: Result:=Operand1+Operand2;
    2: Result:=Operand1-Operand2;
    3: Result:=Operand1*Operand2;
    4: Result:=Operand1 div Operand2;
```

```
5: Result:=Operand1 mod Operand2;  
end;  
NumberEd.Text:=intToStr (Result) ;  
                {результат выполнения арифметической операции  
                поместим в окно редактирования NumberEd}  
end;
```

*Эксперимент.* Запустите приложение. Убедитесь в правильности работы калькулятора. ♦

При нажатии на клавишу 'C' окно ввода очищается:

```
Procedure TCalculatorFrm.ClearBtnClick(Sender: TObject);  
Begin  
NumberEd.Text:='';  
end;
```

Модифицируем программу так, чтобы мы могли пользоваться калькулятором, не только при помощи манипулятора мышью, но и клавиатуры.

Рассмотрим, как приложение обрабатывает события клавиатуры. В Windows используется фокус ввода для того, чтобы определить, куда посылать события клавиатуры. Выбранный в данный момент компонент (активный компонент) всегда обладает фокусом ввода (рис. 4.2.9), и именно он принимает все события клавиатуры.

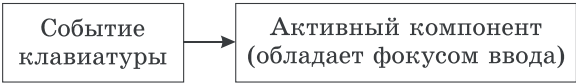


Рис. 4.2.9

Для того чтобы послать события клавиатуры сначала форме (рис. 4.2.10), а потом активному компоненту установите свойство формы KeyPreview равным True. Это может пригодиться для того, чтобы обезопасить пользователя от введения ошибочных символов (например, от введения букв).

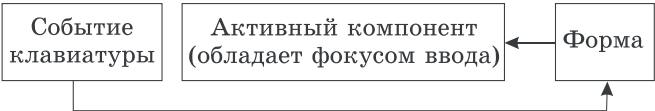


Рис. 4.2.10

Итак, установите значение свойства KeyPreview формы равным True, создайте обработчик события OnKeyPress формы:

```
procedure TCalculatorFrm.FormKeyPress(Sender: TObject;  
  var Key: Char);  
begin  
  case Key of  
    '0' .. '9': NumberEd.Text:=NumberEd.Text+Key;  
                {формируем операнд арифметической операции}  
    '+': AddBtn.Click;  
        {этот оператор возбуждает событие OnClick объекта AddBtn}  
    '=': EqualBtn.Click;  
  end;  
end;
```

*Эксперимент.* Запустите приложение. Сложите два целых числа. Убедитесь в правильности полученного результата. Допишите обработчик события OnKeyPress. ♦

### Задания для самостоятельного выполнения

- 4.3. Подсчитайте количество секунд, прошедших с начала суток, если сейчас N часов, M минут, L секунд.
- 4.4. Напишите программу для расчета платежной ведомости. Форма должна содержать 2 окна редактирования: одно — для ввода количества отработанных часов, другое — для ввода стоимости одного часа. Программа должна вычислять и показывать заработную плату. Каждый час, отработанный сверхурочно, выше недельной нормы в 37,5 часов, оплачивается по ставке, превышающей обычную в 1,5 раза.
- 4.5. Создайте программу нахождения корней квадратного уравнения.
- 4.6. «Прятки». Сценарий выполнения программы: первоначально форма имеет вид, изображенный на рис. 4.2.11; при нажатии на кнопку Close приложение закрывается; при нажатии на кнопку Hide форма изменяет вид (рис. 4.2.12) и у кнопки появляется подсказка 'Нажми на кнопку, чтобы надпись появилась'; при нажатии на кнопку Show форма приобретает первоначальный вид.
- 4.7. «Кодировщик». Заданный заглавными буквами текст зашифровать по следующему правилу: каждому символу исходного текста соответствует следующий за ним символ. Зададим следующий порядок символов: '0'..'9', 'A'..'Z', 'A'..'Я'.

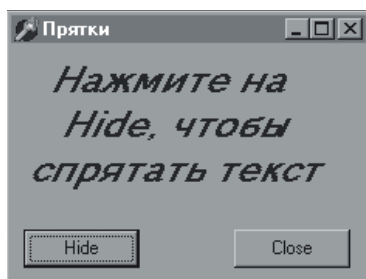


Рис. 4.2.11

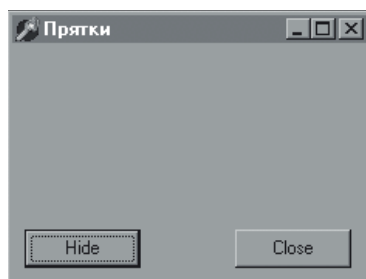


Рис. 4.2.12

- 4.8. На форме расположены строка ввода, три кнопки и три метки. В строку ввода заносится число. При нажатии на первую кнопку в первой метке выводится знак числа из строки ввода, при нажатии на вторую кнопку во второй метке выводится модуль числа; при нажатии на третью кнопку в третьей метке выводится информация о том, является число простым или нет.
- 4.9. Напишите программу, которая по введенной стоимости дописывает слово рубль, рубля или рублей, в зависимости от суммы. Например, 1 рубль, 3 рубля, 5 рублей.
- 4.10. «Угадайка». Программа с помощью датчика случайных чисел выбирает (загадывает) число в диапазоне от 1 до 50. Нужно угадать это число за три попытки. После каждой попытки сообщается, больше или меньше названное число задуманного. После третьей неудачной попытки загаданное число сообщается. На форме должна быть кнопка **новая игра**, при нажатии на которую игра возобновляется.

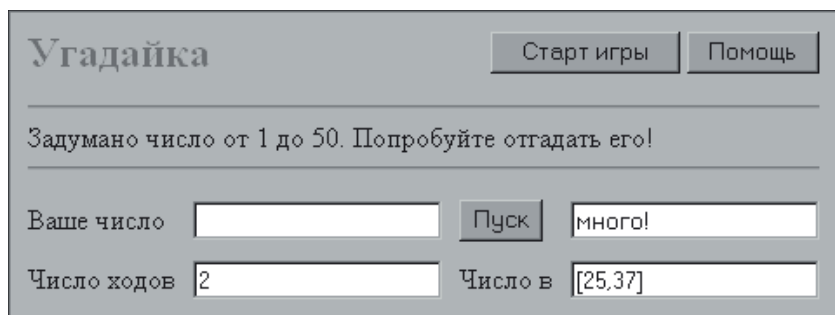


Рис. 4.2.13

- 4.11.** Составьте программу для обучения устному счету. На каждом шаге предлагаются два числа в интервале от 1 до 99 и арифметические действия (сложение или вычитание), которые следует выполнить над этими числами. Для выбора чисел и знаков действий воспользуйтесь функцией `Random`. Программа должна предлагать 10 различных заданий, после каждого из которых программа выдает сообщение о правильности выполнения задания и предлагает перейти к следующему заданию. После выполнения 10-го задания выдается информация о количестве правильно выполненных заданий и предлагается выполнить задания еще раз.
- 4.12.** Постройте кривую по заданному параметрическому представлению: Улитка Паскаля:  $x = a \cos 2t + b \cos t$ ;  $y = a \cos t \sin t + b \sin t$ ,  $t \in [0, 2\pi)$ . Рассмотрите случаи, когда  $b \geq 2a$ ,  $a < b < 2a$ ,  $a > b$ .
- 4.13.** Постройте кривую по заданному параметрическому представлению:  $x = (a - b)\cos t + d \cos \varphi$ ;  $y = (a - b)\sin t - d \sin \varphi$ , где  $\varphi = (a/b)t$ ,  $d < b < a$ ; угол  $t$  меняется от 0 до  $2\pi n$ ,  $n = b / \text{НОД}(a, b)$ .
- 4.14.** Дополните созданный в упр. 4.2.2 калькулятор следующими операциями:
- а) возведение в квадрат;
  - б) возведение в степень  $n$ ;
  - с) нахождение факториала числа;
  - д) логические операции `or`, `xor`, `and`, `not`;
  - е) длинная арифметика.

### Вопросы для повторения

1. Опишите этапы создания приложения в среде программирования Delphi.
2. Для чего предназначены компоненты `Edit` и `Label`? В чем их принципиальное отличие?
3. Перечислите некоторые свойства компонента `Edit`.
4. Перечислите некоторые свойства компонента `Label`.
5. Для чего предназначено свойство `KeyPreview`?
6. Поясните назначение оператора `Button1.OnClick`.

## Списки строк.

# Обработка исключительных ситуаций

### 5.1. Класс TStrings

Очень часто при создании приложений необходимо отображать на экране не одну строку, а несколько. В Delphi для этого используются такие компоненты, как TComboBox, TListBox, TRichEdit, TNotebook, TTabSet, THeader и другие. Каждый из этих компонентов содержит свойство, принадлежащее типу TStrings.

TStrings является абстрактным классом и предназначен для работы с наборами строк. Набор строк технически реализуется в виде массива указателей. С помощью свойств и методов данного класса можно добавлять и удалять строки, сортировать строки, считывать и записывать строки в файл или поток и т. д. Кроме того, элементами набора могут быть пары строка-объект, в которых строка является собственно строкой символов, а объектом может быть объект любого класса Delphi (в том числе и потомок от TStrings). Эта особенность позволяет сохранять в TStrings объекты с текстовыми примечаниями, сортировать объекты, отыскивать нужный объект по его описанию, создавать многомерные наборы строк и т. д.

#### Свойства класса TStrings

Свойство **Capacity** определяет количество строк, которое может храниться в наборе, а абстрактное свойство **Count** содержит текущее количество элементов.

Если при добавлении очередного элемента Capacity окажется меньше Count, происходит автоматическое расширение массива. При этом в динамической памяти резервируется место для размещения Capacity+16 указателей, в новый массив переписывается содержимое старого массива, после чего старый массив уничтожается. Если вам известно количество элементов в создаваемом наборе строк, имеет смысл заранее нужным образом установить свойство Capacity, чтобы сократить непроизводительные расходы на многократные расширения массива указателей.

Свойство **CommaText** служит для задания или получения всего набора строк в формате SDF (системный формат данных, который задает текст в виде единой строки с кавычками, запятыми и пробелами).

### Правила формата SDF:

- строки разделяются запятыми или пробелами (можно также заключить строки в двойные кавычки);
- если строка содержит двойные кавычки, то они удваиваются;
- две идущие подряд запятые обозначают пустую строку;
- все запятые и пробелы, не заключенные в двойные кавычки, считаются разделителями строк.

Например, свойство `CommaText` имеет значение

"Стро,ка1", "Стр"ока 2", , Строка 3, Строка4

то набор состоит из строк:

```
Стро,ка1
Стр"ока 2

Строка
3
Строка4
```

Следующие два свойства используются при представлении строк в формате `Name=Value` (имя = значение). Свойство **Names** содержит набор названий строк, а **Values** — набор соответствующих значений. Например, если *i*-я строка имеет вид

`Color=clGreen` (обратите внимание, что ни до, ни после символа «=» не должно быть пробелов),

то

```
Names[i] := 'Color';
Values[i] := 'ClGreen'.
```

Элементами набора строк могут быть пары строка–объект. Свойство **Objects** открывает доступ к объекту, связанному со строкой с номером `Index`. Нумерация строк начинается с нуля.

Абстрактное свойство **Strings** используется для того, чтобы получить доступ к строке с номером `Index`. Для объектов класса `TStrings` свойство `Strings` является свойством по умолчанию, т. е. для доступа к конкретной строке необязательно использовать свойство `Strings`. Например, следующие операторы выполняют одно и то же действие:

```
var MyStrings: TStrings;
MyStrings.Strings[0] := 'Это первая строка';
MyStrings[0] := 'Это первая строка';
```

Свойство **Text** содержит набор строк объекта в виде одной длинной строки с символами возврата каретки и перехода на новую строку (стандартный признак разделения EOLN = #13#10).

### Методы класса TStrings

<b>function</b> Add( <b>const</b> S: string): Integer; <b>virtual</b> ;	добавляет строку в набор данных и возвращает ее индекс;
<b>function</b> AddObject ( <b>const</b> S: string; AObject: TObject): Integer; <b>virtual</b> ;	добавляет пару строка-объект в набор данных, возвращает индекс новой пары строка-объект;
<b>procedure</b> AddStrings (Strings: TStrings); <b>virtual</b>	добавляет к текущему набору новый набор строк;
<b>procedure</b> Append( <b>const</b> S: string);	добавляет строку в набор данных, но не возвращает индекс новой строки;
<b>procedure</b> Assign (Source: TPersistent); <b>override</b> ;	уничтожает прежний набор строк и загружает из Source новый набор. Если строки в новом наборе связаны с объектами, то объекты также будут загружены;
<b>procedure</b> Clear; <b>virtual</b> ; <b>abstract</b> ;	очищает набор данных и освобождает связанную с ним память;
<b>procedure</b> Delete (Index: Integer); <b>virtual</b> ; <b>abstract</b> ;	уничтожает элемент набора с индексом Index и освобождает связанную с ним память. Нумерация строк начинается с нуля;
<b>destructor</b> Destroy; <b>override</b> ;	уничтожает объект класса TStrings. Рекомендуется не вызывать данный метод в приложении, а вместо него использовать метод Free;
<b>function</b> Equals (Strings: TStrings): Boolean;	сравнивает построчно текущий набор данных с набором Strings и возвращает True, если наборы идентичны;
<b>procedure</b> Exchange (Index1, Index2: Integer); <b>virtual</b> ;	меняет местами строки с индексами Index1 и Index2;
<b>function</b> GetText: PChar; <b>virtual</b> ;	помещает значение свойства <b>Text</b> в динамически создаваемый буфер;

<b>function</b> IndexOf( <b>const</b> S: string): <b>Integer</b> ; <b>virtual</b> ;	для строки S возвращает ее индекс или -1, если такой строки в наборе нет;
<b>function</b> IndexOfName( <b>const</b> Name: string): <b>Integer</b> ;	возвращает индекс строки с именем Name;
<b>procedure</b> Insert(Index: <b>Integer</b> ; <b>const</b> S: string); <b>virtual</b> ; <b>abstract</b> ;	вставляет строку в набор и присваивает ей индекс Index;
<b>procedure</b> InsertObject(Index: <b>Integer</b> ; <b>const</b> S: string; AObject: TObject);	вставляет строку и объект в набор и присваивает им индекс Index;
<b>procedure</b> LoadFromFile( <b>const</b> FileName: string); <b>virtual</b> ;	загружает набор строк из файла;
<b>procedure</b> LoadFromStream(Stream: TStream); <b>virtual</b> ;	загружает набор из потока;
<b>procedure</b> Move(CurIndex, NewIndex: <b>Integer</b> ); <b>virtual</b> ;	перемещает строку из положения CurIndex в положение NewIndex;
<b>Procedure</b> SaveToFile( <b>const</b> FileName: string); <b>virtual</b> ;	сохраняет набор строк в файле с именем FileName;
<b>procedure</b> SaveToStream(Stream: TStream); <b>virtual</b> ;	сохраняет набор строк в потоке;
<b>procedure</b> SetText(Text: PChar); <b>virtual</b> ;	считывает строки из буфера и записывает их в набор строк.

**Упражнение 5.1.1.** Разработайте приложение «Блокнот» для хранения информации по месяцам года.

### Решение

Создайте новый проект. Сохраните новое приложение в папке NoteBook — файл модуля под именем Main.pas, файл проекта — Notebook.dpr.

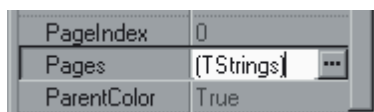
*1-й этап — визуальное проектирование*

Измените значение свойств формы следующим образом:

Caption	Блокнот
Name	NoteBookF

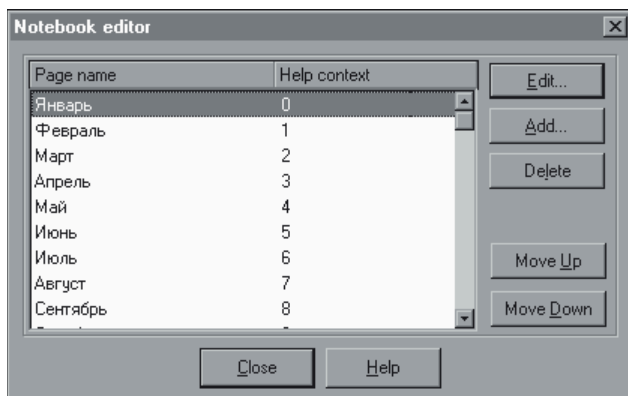
Поместите на форму компонент NoteBook (блокнот) со страницы Win3.1 палитры компонентов.

Создайте страницы компонента NoteBook1. Для этого в инспекторе объектов найдите свойство Pages (страницы); щелкните на кнопке с тремя точками (рис. 5.1.1) для открытия диалогового окна редактирования значений выбранного свойства.



**Рис. 5.1.1**

Используя диалоговое окно, создайте 12 страниц, каждая из которых называется именем, соответствующим месяцу года (рис. 5.1.2). Для редактирования названия первой страницы воспользуйтесь кнопкой Edit, для добавления очередной страницы — кнопкой Add. Значение Help context можно оставить без изменения (это свойство используется при вызове контекстно-зависимой справочной системы приложения), так как приложение не содержит справочной системы. Для завершения создания списка страниц нажмите на кнопку Close.



**Рис. 5.1.2**

Поместите компонент TabSet (список закладок) со страницы Win3.1 под компонентом NoteBook1. Сделайте эти компоненты одинаковыми по ширине (рис. 5.1.3).

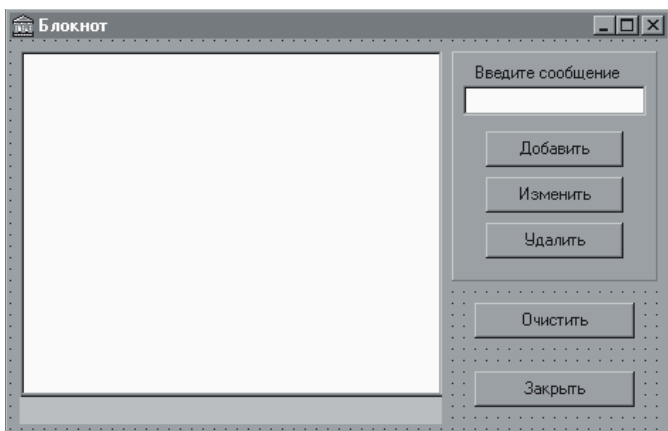


Рис. 5.1.3

Положите на каждую страницу компонента NoteBook1 компонент ListBox (окно списка) для размещения записей на каждый месяц. Для этого выполните такую последовательность действий.

Установите свойство ActivePage (активная страница) компонента NoteBook в значение, равное 'Январь'. Щелкните на выбранной странице объекта NoteBook1. В палитре компонентов на странице Standard выберите компонент ListBox и поместите (щелкните) в компонент NoteBook1. Свойство Align компонента ListBox1 установите в AlClient. Скопируйте компонент ListBox1 в буфер обмена (щелкните на объекте ListBox1 и выполните команду Edit/Сору).

На вторую страницу блокнота поместите компонент Listbox2. Установите свойство ActivePage (активная страница) компонента NoteBook1 в значение, равное 'Февраль'. Щелкните на компоненте Notebook1 (для активизации этой страницы блокнота). Поместите копию компонента ListBox1 на страницу блокнота, выполнив команду Edit/Paste.

Повторите эту операцию для оставшихся 10 страниц блокнота.

Итак, сейчас для каждого месяца года существует свой компонент `ListBox`: для января — `ListBox1`, для февраля — `ListBox2` и т. д.

В правую часть формы поместите компонент `Panel` (страница `Standard` палитры компонентов), на него положите один компонент `Label`, один компонент `Edit` и три компонента `Button` (рис. 5.1.3).

Установите свойства компонентов следующим образом:

Компонент	Свойство	Значение
<code>Panel1</code>	<code>Caption</code>	‘‘
<code>Label1</code>	<code>Caption</code>	‘Введите сообщение’
<code>Edit1</code>	<code>Name</code>	<code>RecordEdt</code>
<code>Edit1</code>	<code>Text</code>	‘‘
<code>Button1</code>	<code>Name</code>	<code>AddBtn</code>
	<code>Caption</code>	Добавить
<code>Button2</code>	<code>Name</code>	<code>ChangeBtn</code>
	<code>Caption</code>	Изменить
<code>Button3</code>	<code>Name</code>	<code>DeleteBtn</code>
	<code>Caption</code>	Удалить

Поместите еще 2 компонента `Button` в правой нижней части формы и измените свойства следующим образом:

Компонент	Свойство	Значение
<code>Button1</code>	<code>Name</code>	<code>ClearBtn</code>
	<code>Caption</code>	Очистить
<code>Button2</code>	<code>Name</code>	<code>CloseBtn</code>
	<code>Caption</code>	Закреть

Итак, визуальное проектирование главной формы приложения завершено.

**Эксперимент.** Сохраните проект. Убедитесь, что после запуска приложения форма ничем не отличается от режима проектирования. ◆

## 2-й этап — создание программного кода

Во-первых, необходимо создать закладки в `TabSet1` и синхронизировать их со страницами блокнота. Для этого используется свойство `Tabs` компонента `TabSet`, предоставляющее список

имен закладок. В нашем примере список закладок соответствует списку страниц блокнота (Pages). Создайте следующий обработчик события:

```
Procedure TNoteBookF.FormCreate(Sender: TObject);
begin
    TabSet1.Tabs:=NoteBook1.Pages;
        {копируем содержимое свойства Pages компонента NoteBook
        в свойство Tabs компонента TabSet, т. е. создаем закладки в TabSet}
    TabSet1.TabIndex:=NoteBook1.PageIndex;
        {В соответствии с тем, какая страница блокнота
        активна во время проектирования (свойство ActivePage),
        активной будет закладка с тем же именем (номером)}
end;
```

**Эксперимент.** Сохраните и запустите проект. Убедитесь в том, что закладки на форме получили названия, соответствующие страницам блокнота.

Используя свойства UnSelectedColor, SelectedColor, BackgroundColor, DitherBackground, StartMargin, EndMargin компонента TabSet1, измените внешний вид закладок. ♦

Создайте обработчик события OnClick для кнопки 'Добавить'. После щелчка на кнопке содержимое окна редактирования (RecordEdt) будет добавляться в список на странице блокнота. Для операций со строками у компонента ListBox имеется свойство Items, принадлежащее классу TStrings. Чтобы добавить строку в список, используется метод Add этого класса (см. таблицу методов класса TStrings п. 5.1.1). Обработчик события будет выглядеть следующим образом:

### *1-й способ*

```
procedure TNoteBookF.AddBtnClick(Sender: TObject);
begin
    if RecordEdt.Text <>'' then
begin
        case Notebook1.PageIndex of
            {Для того чтобы поместить запись в список на страницу блокнота,
            выбираем компонент ListBox по номеру страницы блокнота}
            0: ListBox1.Items.Add(RecordEdt.Text);
            1: ListBox2.Items.Add(RecordEdt.Text);
            2: ListBox3.Items.Add(RecordEdt.Text);
            3: ListBox4.Items.Add(RecordEdt.Text);
            4: ListBox5.Items.Add(RecordEdt.Text);
            5: ListBox6.Items.Add(RecordEdt.Text);
            6: ListBox7.Items.Add(RecordEdt.Text);
            7: ListBox8.Items.Add(RecordEdt.Text);
```

```
8: ListBox9.Items.Add(RecordEdt.Text);
9: ListBox10.Items.Add(RecordEdt.Text);
10: ListBox11.Items.Add(RecordEdt.Text);
11: ListBox12.Items.Add(RecordEdt.Text);
end;
RecordEdt.Text:=''; {введенный текст затираем}
end;
end;
```

Процедура получилась довольно громоздкой, ее можно описать по-другому.

### *2-й способ*

В Object Pascal определена функция:

```
function FindComponent(const AName: string): TComponent;
```

которая возвращает ссылку на компонент, имя которого задано параметром функции AName.

Воспользуемся этой функцией в коде обработчика события OnClick кнопки 'Добавить':

```
procedure TNoteBookF.AddBtnClick(Sender: TObject);
var TempComponent: TListBox;
    {введем вспомогательную переменную}
begin
    TempComponent:=TListBox(FindComponent('ListBox'+
    IntToStr(Notebook1.PageIndex+1)));
    {TListBox(<компонент>) используется для преобразования
    к типу TListBox}
    if RecordEdt.Text<>' ' then
        TempComponent.Items.Add(RecordEdt.Text);
    RecordEdt.Text:='';
end;
```

**Эксперимент.** Сохраните и запустите проект. Попробуйте добавить строки на разные страницы блокнота. Объясните, почему все строки у вас записываются только на страницу, которая была открыта при запуске приложения. ♦

Строки записываются на одну страницу блокнота потому, что закладки Tabset1 не синхронизированы со страницами Notebook1, т. е. при выборе закладки соответствующая страница блокнота не активизируется.

Чтобы устранить этот недостаток, создайте обработчик события OnClick закладок:

```
procedure TNoteBookF1.TabSet1Click(Sender: TObject);
begin
```

```
NoteBook1.PageIndex:=TabSet1.TabIndex;  
{страница блокнота и страница набора закладок должны совпадать}  
end;
```

**Эксперимент.** Сохраните проект. Проверьте правильность работы приложения, т. е. внесение записей на страницы блокнота. ♦

## Задания для самостоятельного выполнения

- 5.1. Напишите обработчик события нажатия кнопки 'Изменить': выделенная в списке страницы блокнота строка должна быть заменена содержимым окна редактирования.
- 5.2. Напишите обработчик события нажатия кнопки 'Удалить': выделенная в списке страницы блокнота строка должна быть удалена.
- 5.3. Напишите обработчик события нажатия кнопки 'Очистить': содержимое всего блокнота удаляется.
- 5.4. Модифицируйте программу следующим образом:
  - a) при выборе записи на странице блокнота соответствующая запись отображается в окне редактирования;
  - b) строка следующая (предыдущая) за удаленной становится выделенной и отображается в окне редактирования;
  - c) добавление записи из окна редактирования происходит при нажатии клавиши ввода.

## 5.2. Исключительные ситуации

Исключительная (или особая) ситуация представляет собой сигнал о произошедшей в приложении ошибке. Для обработки исключительных ситуаций используются две конструкции:

```
try  
    защищаемый код  
finally  
    код завершения  
end;
```

```
try  
    защищаемый блок  
except  
    обработчик исключения  
end;
```

В первом варианте блок `finally` выполняется независимо от того, возникла ли ошибка в процессе выполнения защищаемых операций.

Блок `try except` используется для обработки исключительных ситуаций. В Object Pascal исключительные ситуации представляют собой объекты, содержащие информацию, идентифицирующую ошибку и место ее возникновения. Внутри части

ехсерт создаются обработчики особых ситуаций для классов исключительных ситуаций. Обработчик особой ситуации имеет следующий формат:

```
try
                                                    защищаемый блок
except
  on E: ESomeException do <обработчик исключения>;
end;
```

Для обработки особой ситуации Delphi предоставляет возможность создания временных объектов особой ситуации (E), которые могут использоваться в обработчике исключения. Временный объект особой ситуации имеет тот же тип, что и объект исключения, который он обозначает (ESomeException).

Существуют предопределенные классы исключительных ситуаций для обработки стандартных ошибок, таких, как нехватка памяти, деление на ноль, числовое переполнение, ошибки ввода-вывода и другие. Перечислим некоторые из них:

<b>EMathError</b>	класс-предок исключений, случающихся при выполнении операций с плавающей точкой;
EInvalidArgument	значение параметра выходит за диапазон значений
EInvalidOp	передача математическому сопроцессору ошибочной конструкции;
EOverflow	переполнение разрядов при работе со слишком большими величинами;
EUnderflow	потеря разрядов при работе со слишком малыми величинами;
EZeroDivide	деление на ноль;
<b>EIntError</b>	класс-предок исключений, случающихся при выполнении целочисленных операций;
EDivByZero	деление на ноль;
EIntOverflow	выполнение операций, приводящих к переполнению целых переменных;
ERangeError	значение целочисленного выражения выходит за пределы установленного целочисленного типа. Попытка обращения к элементу массива по индексу, выходящему за пределы массива;
<b>EListError</b>	обращение к элементу списка (String, List, TStringList) по индексу, выходящему за пределы допустимых значений;

**EInOutError** ошибки при операциях с файловой системой. Код ошибки возвращается в локальной переменной **ErrorCode**, которая может принимать следующие значения:

- 2 файл не найден
- 3 неверное имя файла
- 4 слишком много открыто файлов
- 5 отказ в доступе
- 100 конец файла
- 101 диск полон
- 106 неверная операция ввода;

**EConvertError** ошибки преобразования типов (простых, объектных).

Например, для создания устойчивого к ошибкам приложения, записывающего данные в файл, необходимо написать следующий код:

```
begin
  assignfile(f, 'data.dan');
  try
    append(f);
    try
      writeln(f,s);
    finally
      closefile(f);
    end;
  except
    on E: EInOutError do
      if E.ErrorCode = 2 then
        if MessageDlg('Файл не найден. Создать его?', mtError,
[mbYes, mbNo],0)=mrYes then FileCreate('data.dan');
      end;
    end;
  end;
```

В данном листинге внутренний блок **try finally** используется для того, чтобы файл был закрыт в любом случае, т. е. независимо от того, была ошибка или нет.

Внешний блок **try except** используется для обработки исключительной ситуации, которая может произойти в программе. После закрытия файла во внутреннем блоке **finally** блок **except** использует временный объект **E** для определения типа произошедшей ошибки ввода/вывода.

**Упражнение 5.2.1.** Модифицируйте приложение, созданное в упр. 5.1.1, так, чтобы при повторном запуске программы данные, введенные в предыдущем сеансе, отображались на страницах блокнота.

### Решение

Напишем метод, который все странички блокнота будет сохранять в отдельных файлах в каталоге, указанном в переменной `dir`. В разделе `public` описания класса `TNoteBookF` опишите метод `SaveAll` и переменную `dir`:

```
...
public
    { Public declarations }
    dir: string;
    procedure SaveAll;
```

...

В разделе `Implementation` модуля опишите программный код метода:

```
procedure TNoteBookF.SaveAll;
var i: integer;
    TempComponent: TListBox;
begin
    InputQuery('Введите полное имя каталога', '', dir);
    for i:=0 to 11 do
begin
    Notebook1.PageIndex:=i;
    TempComponent:=
        TListBox(FindComponent('ListBox'+IntToStr(i+1)));
    TempComponent.Items.SaveToFile(dir+Month'+intToStr(i+1));
        {Строки компонента ListBox сохраняются
         в соответствующем файле}
end;
end;
```

**Эксперимент.** Сохраните и запустите приложение. Убедитесь в работоспособности приложения для существующих каталогов. Попробуйте сохранить файлы в несуществующий каталог, например `c:\NoteBook\Month\`. ♦

Для несуществующего каталога операционная система возбуждает исключительную ситуацию, результатом которой отображается сообщение об ошибке, изображенное на рис. 5.2.1: «Приложение `Project1.exe` возбудило исключительную ситуацию типа `EFCREATEERROR` с сообщением 'Невозможно создать

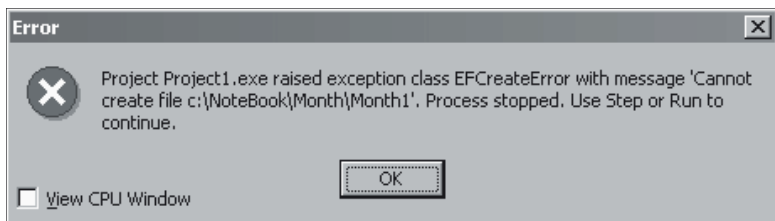


Рис. 5.2.1

файл C:\NoteBook\Month\Month1'. Процесс остановлен. Воспользуйтесь командами Step или Run для продолжения.»

Чтобы обезопасить приложение от подобного вида ошибок, добавим блок обработки исключительной ситуации EFCREATEError (рис. 5.2.1), возникшей при сохранении файла:

```
begin
  InputQuery('Введите полное имя каталога', '', dir);
  if dir[length(dir)] <> '\' then dir:=dir+'\'
    {добавляем разделитель подкаталогов в конец строки}
  for i:=0 to 11 do
  begin
    Notebook1.PageIndex:=i;
    TempComponent:=
      TListBox(FindComponent('ListBox'+IntToStr(i+1)));
    try
      TempComponent.Items.SaveToFile(dir+'Month'+intToStr(i+1));
      {Строки компонента ListBox сохраняются в соответствующем файле}
    except
      on EFCREATEError do
      begin
        {класс исключений, возникающих при неудачных попытках
          создания файла}
        mkdir(dir);      {процедура создания каталога (смотри ниже)}
        TempComponent.Items.SaveToFile(dir+'Month'+intToStr(i+1));
      end;
    end;
  end;
end;
```

Рассмотрим процедуру создания подкаталогов:

```
procedure MakeDir(dir: string);
var d: string;
    k: byte;
begin
  k:=pos('\', dir); d:=copy(dir, 1, k); delete(dir, 1, k);
```

{выделяем название корневого каталога из dir, запоминаем  
в переменной d}

```
while DirectoryExists(d) do
begin
    {пока каталог с именем, указанным в переменной d, существует}
    k:=pos('\',dir);
    d:=d+copy(dir,1,k);
    delete(dir,1,k);
    {выделяем имя очередного подкаталога из переменной dir}
end;
{в переменной d хранится имя несуществующего каталога}
CreateDir(d);
    {стандартная функция создания подкаталога с именем d}
while (pos('\',dir)<>0) do
begin
    k:=pos('\',dir); d:=d+copy(dir,1,k); delete(dir,1,k);
    CreateDir(d);
end;
end;
```

**Эксперимент.** Запустите приложение. Убедитесь, что при вводе неправильного имени каталога происходит обработка исключительной ситуации и создается каталог с заданным именем. ◆

## Задания для самостоятельного выполнения

### 5.5. Сохранение данных блокнота.

- При нажатии кнопки 'Закрыть', если были внесены изменения в блокноте, появляется диалоговое окно с текстом 'Сохранить изменения?' и двумя кнопками 'Да' и 'Нет'. При нажатии на 'Да' данные сохраняются в файлах, при нажатии на 'Нет' — приложение закрывается без сохранения изменений.
- Модифицируйте код приложения так, чтобы вопрос «Сохранить изменения?» появлялся только, если данные блокнота были изменены.
- После щелчка на системной кнопке закрытия формы (крестик в правом верхнем углу формы), форма закрывается без предупреждения о необходимости сохранить данные блокнота. Исправьте этот недостаток.

### 5.6. Считывание данных в блокнот.

- При запуске приложения страницы заполняются данными из файлов.

- b) Создайте модальное окно выбора папки для сохранения (и считывания) данных блокнота. Модифицируйте код приложения так, чтобы при запуске появлялось модальное окно для выбора папки, в которой хранятся данные.

**Примечание.** Используйте компоненты DriveComboBox, DirectoryListBox, FileListBox, FilterComboBox со страницы Win 3.1 палитры компонентов.

- 5.7. Данные блокнота хранятся в двенадцати файлах — это не является рациональным с точки зрения компактности данных. Продумайте структуру файла для хранения записей блокнота и внесите необходимые изменения в программный код.

**Примечание.** Для работы с файловым типом данных в языке Object Pascal используются следующие процедуры:

AssignFile(<файловая переменная>, <имя файла>) — устанавливает соответствие между внешним файлом и файловой переменной.

Reset(<файловая переменная>) — открывает на чтение существующий внешний файл, ассоциированный с файловой переменной

Append(<файловая переменная>) — подготавливает существующий файл, ассоциированный с файловой переменной, для добавления записей в конец.

Rewrite(<файловая переменная>) — создает новый файл с именем, соответствующим файловой переменной, и устанавливает его в режим записи элементов.

CloseFile(<файловая переменная>) — разрывает установленное соответствие между файловой переменной и внешним файлом.

- 5.8. Игра «Виселица». В файле содержатся слова русского алфавита. Приложение случайным образом выбирает одно из слов и предлагает его отгадать. Каждая буква загаданного слова отображается звездочкой. Игрок пытается отгадать слово, выбирая буквы из алфавита. Если выбранная буква в слове есть, она (или они, если таких букв несколько) отображается вместо звездочки в слове и удаляется из алфавита. Если выбранной буквы в слове нет, появляется очередной элемент виселицы (горизонтальная балка, перекладина, веревка, голова, туловище, правая рука, левая рука, правая нога, левая нога) и буква удаляется из алфавита. Игра заканчивается победой, если слово отгадано, проигрышем, если появились все элементы виселицы.

- 5.9. Используя компонент RadioGroup, напишите приложение-тест. Первое окно: приложение предлагает ввести имя тестируемого. Далее даются вопросы теста по одному:

1. В компании кто-то рассказывает историю, которую вы уже слышали. Вы

- а) улыбаясь, спокойно выслушиваете до конца;
- б) отворачиваетесь или выходите из комнаты;
- в) слушаете, но всем своим видом выражаете недовольство;
- г) перебиваете рассказчика.

2. Кто-то высказывает мнение в корне противоречащее вашему. Вы:

- а) выслушиваете и продолжаете заниматься своим делом;
- б) стараетесь выйти;
- в) говорите, что не желаете обсуждать с ним это;
- г) защищаете свои убеждения.

3. Вы считаете, что

- а) люди как правило на добро отвечают добром;
- б) многие быстро забывают доброе к ним отношение;
- в) люди корыстны;
- г) многие неблагодарны и пользуются вашим добрым отношением к ним.

4. Настоящий друг

- а) помогает тому, кого считает своим другом;
- б) старается помочь;
- в) в любых ситуациях поддержит;
- г) всегда разделяет точку зрения того, кого считает другом.

5. В отношениях лучше, когда

- а) никто из партнеров ни от кого не зависит;
- б) оба партнера друг от друга во многом зависят;
- в) кто-то зависит от вас;
- г) вы зависите от партнера, зато он многое берет на себя.

6. Когда кто-то часто допускает речевые ошибки, вы

- а) не обращаете внимания;
- б) стараетесь лишь изредка поправлять его;
- в) предлагаете ему взять уроки русского языка;
- г) не можете удержаться от того, чтобы постоянно не поправлять его.

7. Встречаясь с новыми людьми, вы

- а) ведете себя естественно;
- б) бываете остроумны, чтобы произвести впечатление;
- в) стремитесь быть настороже, чтобы ничто не прошло мимо вашего внимания;
- г) постоянно помните, что подвергаетесь оценке окружающих.

8. Вы

- а) любите обсуждать комические ситуации, в которых сами побывали;
- б) любите когда подшучивают над вами;

- в) очень любите подшучивать над другими;
- г) не позволяете, чтобы другие подшучивали над вами.

9. Когда вас приглашает в гости кто-то из близких друзей, а вы уже собрались в другое место, то вы

- а) говорите об этом;
- б) не называете причины отказа;
- в) прикидываетесь, что больны;
- г) говорите, что у вас работа.

10. Вы считаете совершенно необходимым

- а) быть с собой в ладу в любых ситуациях;
- б) всегда соответствовать общему духу компании;
- в) чтобы окружающие считались с вашим настроением;
- г) держаться с достоинством и не показывать своих чувств.

Для перехода на следующий вопрос теста необходимо обязательно выбрать один из предлагаемых ответов. После ответа на последний вопрос теста предлагается интерпретация полученного результата:

Если общая сумма ответов «а» составляет 7 и более, то большинство людей относятся к вам с симпатией. Умение быть искренним и в то же время не обижать позволяет вам сохранять мир с самыми разными людьми. Вы умеете, оставаясь собой, позволять и другим быть собою. Вы не подстраиваетесь под чужое мнение, но и не требуете к себе особого отношения, проявляя терпимость и способность понять другого.

Если общая сумма ответов «б» составляет 7 и более, то у вас, в общем, хорошие отношения с людьми. Вы достаточно терпимы. Иногда, правда, вы слишком «подыгрываете» окружающим, вплоть до неискренности. От этого сами, бывает, чувствуете дискомфорт. Тем не менее вам не чужды прагматизм и некоторое недоверие к окружающим. Поверьте, они это чувствуют.

Если общая сумма ответов «в» составляет 7 и более, то лишь немногие относятся к вам с симпатией. Со многими у вас возникают проблемы. Вы недостаточно терпимы к окружающим, вас многое раздражает. Вы предпочитаете иногда слишком дистанцироваться или, наоборот, чересчур давить на людей, если что-то вам не по вкусу. Ваше недоверие и подозрительность не могут оставаться незамеченными и отталкивают людей.

Если общая сумма ответов «г» составляет 7 и более, то вам никто не симпатизирует.

Если в ваших ответах самые разные варианты, то люди к вам относятся по-разному. Одни — с симпатией, а кому-то вы антипатичны. Но в любом случае, чем больше в ваших ответах вариантов «а», тем больше у вас шансов на взаимопонимание и взаимную симпатию в общении.

## 5.3. Класс TList

Класс TList используется в качестве основы для создания списков общего назначения. Кроме того, TList можно использовать как класс-контейнер для хранения в списках объектов любого типа. В большинстве случаев, чтобы использовать класс TList, необходимо создать два собственных класса: один — для элементов, предназначенных для хранения в списке, а другой — для самого списка.

**Упражнение 5.3.1.** Напишите приложение, позволяющее создавать список абонентов телефонного узла.

### Решение

Создайте новый проект. Сохраните новое приложение в папке PhoneBook — файл модуля под именем Main.pas, файл проекта — PhoneBook.dpr.

*1-й этап. Создадим визуальный интерфейс приложения (рис. 5.3.1)*

Поместите на форму семь компонентов Label, четыре Edit, два SpinEdit, один Button и один MaskEdit (Addition).

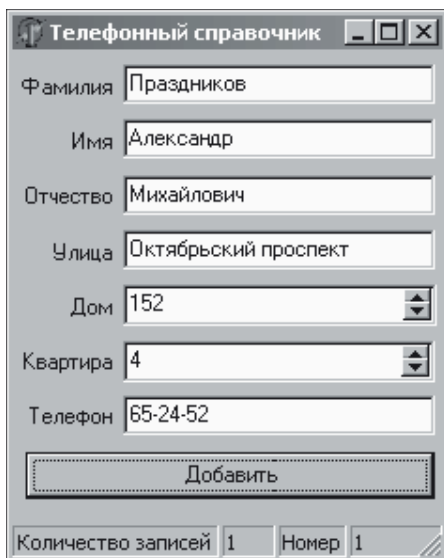


Рис. 5.3.1

Измените значения свойства `Caption` компонентов `Label`, `Form1` и `Button1` в соответствии с рис. 5.3.1, очистите свойство `Text` компонентов `Edit`, свойства `Name` компонентов установите следующим образом:

Компонент	Свойство	Значение
<code>Form1</code>	<code>Name</code>	<code>MainF</code>
<code>Edit1</code>	<code>Name</code>	<code>LastNameEdt</code>
<code>Edit2</code>	<code>Name</code>	<code>NameEdt</code>
<code>Edit3</code>	<code>Name</code>	<code>PatronymicEdt</code>
<code>Edit4</code>	<code>Name</code>	<code>StreetEdt</code>
<code>SpinEdit1</code>	<code>Name</code>	<code>HouseSE</code>
<code>SpinEdit2</code>	<code>Name</code>	<code>FlatSE</code>
<code>Button1</code>	<code>Name</code>	<code>AddBtn</code>
<code>MaskEdit1</code>	<code>Name</code>	<code>PhoneME</code>
	<code>AutoSelect</code>	<code>False</code>
	<code>EditMask</code>	<code>!00-00-00;1;_</code>
<code>StatusBar1</code>	<code>Panels[0].Text</code>	Количество записей
	<code>Panels[0].Width</code>	110
	<code>Panels[1].Text</code>	
	<code>Panels[1].Width</code>	30
	<code>Panels[2].Text</code>	Номер
	<code>Panels[2].Width</code>	38
	<code>Panels[3].Text</code>	

Свойство `EditMask` используют, чтобы обеспечить ввод данных в соответствии с заданной маской. Если пользователь вводит несовместимые с заданной маской символы, `MaskEdit` игнорирует их.

Для ввода номера телефона используется маска: `!00-00-00;1;_`

Маска состоит из трех полей, разделенных символом точка с запятой. Первая часть определяет собственно маску. Для построения маски используются специальные символы:

!	Поле заполняется слева направо
0	Цифра (0-9) или знак (+/-) обязательны
-	Автоматически вставляет пробел в указанной позиции
>	Все символы справа от указанной позиции переводятся в верхний регистр
<	Все символы слева от указанной позиции переводятся в нижний регистр
, :: / - ( )	Разделители
L	Буква обязательна

**Примечание.** Другие символы, используемые для создания маски, можно посмотреть в справочной системе Delphi.

Таким образом, поле Телефон определен как шестизначный номер, цифры которого разбиваются по парам и будут заполняться слева направо.

Вторая часть маски определяет, сохранять ли невводимые символы как данные. Значение параметра, равное нулю, показывает, что свойство Text для примера содержит 6 символов (невводимые символы (в примере — «-») не сохраняются). Любое другое значение параметра заставляет сохранять невводимые символы в свойстве Text (для примера свойство Text будет содержать 8 символов). В примере параметр равен единице, невводимые символы будут сохраняться в свойстве Text.

Третья часть определяет символ, определяющий поля маски, которые необходимо заполнить. В примере это символ «\_».

### *2-й этап. Создание программного кода*

Предположим, что длина полей LastName (фамилия), Name (имя), Patronymic (отчество) не превосходит 25 символов, а Street (улица) — 40 символов. Измените свойства MaxLength компонент Edit в соответствии с введенными ограничениями.

Опишем два новых типа:

**type**

```
PMYList = ^AList;
AList = record
  LastName: string[25];
  Name: string[25];
  Patronymic: string[25];
  Street: string[40];
  House: integer;
  Flat: integer;
  Phone: string[8];
end;
```

В разделе `private` описания класса `TMainF` опишите переменные:

```
MyList: TList;           {для хранения введенных записей}
ARecord: PMyList;       {для формирования полей одной записи}
Yk: Integer;            {указатель перемещения по записям MyList}
```

При запуске приложения создадим объект `MyList` типа `TList`:

```
procedure TMainF.FormCreate(Sender: TObject);
begin
    MyList := TList.Create;    {выделение памяти под объект MyList}
    Yk:=-1;
end;
```

При завершении работы с приложением освободим память, выделенную под объект `MyList`:

```
procedure TMainF.FormDestroy(Sender: TObject);
begin
    MyList.Free;
end;
```

Для добавления записи опишем обработчик события `OnClick` кнопки `Добавить`:

```
procedure TMainF.AddBtnClick(Sender: TObject);
begin
    New(ARecord);           {выделяем место под запись}
    with ARecord do
    begin                     {заполняем поля записи}
        LastName:=LastNameEdt.Text;
        Name:=NameEdt.Text;
        Patronymic:= PatronymicEdt.Text;
        Street:=StreetEdt.Text;
        House:=HouseSE.Value;
        Flat:=FlatSE.Value;
        Phone:= PhoneME.Text;
    end;
    MyList.Add(ARecord);     {добавляем запись в объект MyList}
    inc(yk);                 {увеличиваем значение указателя}
    StatusBar1.Panels[1].Text:=IntToStr(MyList.Count);
                               {отображаем количество записей в панели статуса}
    StatusBar1.Panels[3].Text:=IntToStr(yk+1);
                               {отображаем номер отображаемой записи в панели статуса}
end;
```

**Эксперимент.** Сохраните проект. Запустите его и попробуйте добавить в телефонную книгу информацию о себе и своем друге. Убедитесь, что в строке состояния отображается число два (количество введенных записей). ♦

## Задание для самостоятельного выполнения

### 5.10. Дополните приложение PhoneBook

- а) кнопками для перемещения по записям списка;

**Примечание.** Для обращения к полям записи воспользуйтесь операторами:

```
ARecord:=MyList[yk];  
with ARecord^ do ...
```

- б) сохраните список записей телефонной книги в файле;  
в) при запуске приложения считайте данные из файла;  
г) процедурой поиска записей в телефонной книге по фамилии и вывода найденных записей в диалоговое окно;  
д) функцией добавления абонента в список, сохраните внесенные изменения в файле;  
е) процедурой удаления абонента из списка, сохраните изменения в файле.

**Примечание.** Для удаления записи из списка необходимо выполнить следующую последовательность операторов:

```
ARecord := MyList.Items[B];  
                                     {B — это номер удаляемой записи}  
Dispose(ARecord);
```

**5.11.** Данные о фондах библиотеки хранятся в файле. Формат данных: автор, название книги, шифр издания, год издания, количество книг данного автора в фонде библиотеки. Все сведения в файле неупорядочены. Требуется прочитать сведения из файла в динамическую последовательность. Упорядочить сведения по фамилиям авторов методом простого выбора. Обеспечить выборку книг одного автора по запросу пользователя. Реализовать возможность внесения дополнений в список при поступлении новых книг в библиотеку. Все пункты обработки сведений должны выводиться в окно этого пункта работы.

Дополните приложения следующими функциями:

- а) удалить данные из файла по запросу;  
б) исправить шифр издания;  
в) исправить количество книг автора.

## 5.4. Классы TStringList и TIniFile.

### Динамическое помещение компонентов на форму

Класс TStringList является абстрактным классом, определяющим все поведение, ассоциирующееся со списком строк. Однако TStringList не обеспечивает никаких механизмов для реального хранения списка строк. За это несут ответственность потомки TStringList, подобные TStringList. Класс TStringList использует объект TList для создания списков строк. Чаще всего объекты TStringList применяются для чтения и записи текстовых файлов.

Приведем пример использования объекта TStringList:

```
var SL: TStringList;  
begin  
  SL:=TStringList.Create; {создаем экземпляра класса TStringList}  
  try  
    SL.LoadFromFile('data.txt'); {загружаем данные из файла}  
    listbox1.items.AddStrings(SL); {добавляем данные в listbox1}  
  finally  
    SL.free; {освобождаем память,  
             выделенную под экземпляра типа TStringList}  
  end;  
End;
```

В Delphi существует класс TIniFile, который облегчает чтение и запись инициализационных файлов Windows (файлы с расширением .ini).

**Примечание.** Вместо Ini-файлов в win32 используется системный реестр, в котором приложения сохраняют свои установки. Для работы с реестром в Delphi используется класс TRegistryIniFile. Работа с реестром аналогична работе с классом TIniFile.

Класс TIniFile не является компонентом, поэтому создание и освобождение объектов типа TIniFile, а также вызов их методов необходимо осуществлять только программным образом.

Для создания или обновления файла необходимо сначала создать объект типа TIniFile, например, в обработчике события OnCreate формы:

```
IniFile := TIniFile.Create(<имя_файла>);
```

После окончания работы с объектом IniFile его необходимо освободить с помощью оператора:

```
IniFile.Free;
```

Ini-файл имеет следующую структуру:

```
[название_раздела1]
имя_ключа_10=значение_ключа_10
имя_ключа_11=значение_ключа_11
...
имя_ключа_1n=значение_ключа_1n
[название_раздела2]
имя_ключа_20=значение_ключа_20
имя_ключа_21=значение_ключа_21
...
имя_ключа_2n=значение_ключа_2n
...
```

Для работы с Ini-файлами используются следующие методы:

```
function SectionExists
(const Section: string):
boolean;
```

определяет существование раздела, заданного параметром Section, в INI-файле;

```
function ValueExists (const
Section, Ident: string):
boolean;
```

определяет, существует ли заданное значение ключа Ident в разделе Section Ini-файла;

```
procedure ReadSection (const
Section: string; Strings:
TStrings);
```

читает все имена ключей из раздела Section Ini-файла в переменную Strings;

```
procedure ReadSectionValues
(const Section: string;
Strings: TStrings);
```

читает значения всех ключей из раздела Section Ini-файла в переменную Strings;

```
function ReadString (const
Section, Ident, Default:
string): string;
```

извлекает значение типа **string** из раздела Section, значение ключа Ident. Параметр Default определяет значение по умолчанию в том случае, если

Подобные функции:

ReadBool

ReadDate, ReadDateTime,

ReadTime

ReadFloat

ReadInteger

- заданный в Section раздел не существует,
- заданный в Ident ключ не существует, значение ключа не задано;

```
procedure WriteString(const
Section, Ident, Value:
string);
```

используется для записи значения типа **string** в INI-файл. Параметр Section определяет секцию, а Ident — имя ключа, для которого устанавливается значение. Параметра Value задает значение строки для записи.

Подобные процедуры:

WriteBool

WriteDate, WriteDateTime,

WriteTime

WriteFloat

WriteInteger

**Упражнение 5.4.1.** Напишите приложение, которое позволяет настраивать цвет компонентов ListBox, TabSet, Edit.

### Решение

Создайте новый проект. Сохраните новое приложение в папке ColorIni — файл модуля под именем Main.pas, файл проекта — ColorIni.dpr.

*1-й этап. Создадим визуальный интерфейс приложения (рис. 5.4.1)*

Для выбора компонента воспользуемся компонентом ComboBox, RadioGroup — для выбора изменяемого свойства, Shape — для отображения значения выбранного свойства.

Для подбора цветовых значений используются полосы прокрутки ScrollBar) RedSB, GreenSB, BlueSB, или строки ввода (SpinEdit), в которые можно ввести целые значения от 0 до 255.

Для передачи выбранного цвета компоненту служит кнопка Установить (Set), для возвращения первоначального значения — кнопка Отменить (Reset). Для сохранения текущих цветовых установок выбранного компонента в файле Colors.ini используется кнопка Сохранить (Save). Для завершения работы приложения — кнопка Закреть (Close).

Поместите на форму 4 компонента Label, ComboBox, RadioGroup, Shape, по 3 компонента ScrollBar и SpinEdit, 4 компонента Button.

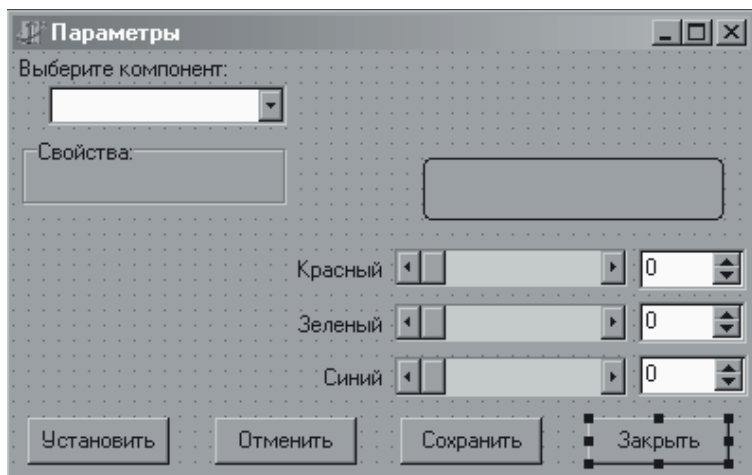


Рис. 5.4.1

Установите значения Caption компонента, в соответствии с рис. 5.4.1, измените значения других свойств следующим образом:

Компонент	Свойство	Значение
Form1	Name	MainF
Shape1	Name	CurrentShape
	Shape	stRoundRect
ScrollBar1	Name	RedSB
	Tag	0
ScrollBar2	Name	GreenSB
	Tag	1
ScrollBar3	Name	BlueSB
	Tag	2
RedSB, GreenSB, BlueSB	Max	255
Button1	Name	SetBtn
Button2	Name	ResetBtn
Button3	Name	SaveBtn
Button4	Name	Close
ComboBox1	Items.Strings	TabSet ListBox Edit
RadioGroup1	Name	ColorPropertyRG
SpinEdit1	Name	RedSE
	Tag	0
SpinEdit2	Name	GreenSE
	Tag	1
SpinEdit3	Name	BlueSE
	Tag	2
RedSE, GreenSE, BlueSE	AutoSelect	False
	MaxValue	255
	Value	0

Расположите компоненты на форме в соответствии с рис. 5.4.1, сохраните проект.

### *2-й этап. Разработка программного кода*

Выбранные цветовые параметры компонента TabSet будем хранить в файле Colors.ini. Для работы с Ini-файлом создадим экземпляр класса TIniFile, в разделе Private описания класса формы опишите переменную

```
IniFile: TIniFile;
```

Кроме этого необходимо задать имя Ini-файла. В разделе Interface модуля опишите константу — полное имя Ini-файла:

```
IniFileName='c:\Colors.ini';
```

Для синхронизации выбранных цветовых значений в компонентах ScrollBar и SpinEdit введем вспомогательные массивы:

```
EditControls: array[0..2] of TSpinEdit;  
ScrollBars: array[0..2] of TScrollBar;
```

Опишите эти переменные в разделе Private описания класса формы.

При запуске приложения необходимо инициализировать значения элементов массивов и создать экземпляр класса TIniFile. Создайте обработчик события OnCreate формы:

```
procedure TMainF.FormCreate(Sender: TObject);  
begin  
  editcontrols[0]:=redSE;  
  editcontrols[1]:=greenSE;  
  editcontrols[2]:=blueSE;  
  scrollbars[0]:=redSb;  
  scrollbars[1]:=greenSb;  
  scrollbars[2]:=blueSb;  
  iniFile:=TIniFile.Create(IniFileName);  
end;
```

После завершения работы с приложением освободим память, выделенную под объект IniFile:

```
procedure TMainF.FormDestroy(Sender: TObject);  
begin  
  IniFile.Free;  
end;
```

Компонент ComboBox1 служит для выбора компонента, цветовые настройки которого нужно изменить. Для обработки события выбора элемента из списка Items используется обработчик OnChange.

В компонентах типа TTabSet для изменения цветового отображения служат свойства BackGroundColor (фон), SelectedCo-

lor (выбранная закладка), UnSelectedColor (невыведенные закладки), Font.Color (текст), в компонентах типа TEdit и TListBox — Color (фон), Font.Color (текст).

При событии OnChange компонента Combobox1 в ColorPropertyRG (типа TRadioGroup) нужно добавить соответствующие выбранному компоненту свойства:

```
procedure TMainF.ComboBox1Change(Sender: TObject);
begin
    ColorPropertyRG.Items.Clear;
                                {удаляем свойства предыдущего выбора}
    case combobox1.itemindex of 0:
begin                                {выбран TabSet}
    with ColorPropertyRG.Items do
begin
        Add('BackgroundColor');
        Add('SelectedColor');
        Add('UnSelectedColor');
        Add('FontColor');
    end;
    NewTabset;
end;
1: with ColorPropertyRG.Items do
begin                                {выбран Edit}
    Add('Color');
    Add('FontColor');
end;
2: with ColorPropertyRG.Items do
begin                                {выбран ListBox}
    Add('Color');
    Add('FontColor');
end;
end;
ColorPropertyRG.Height:=35*(ColorPropertyRG.items.Count)
                                {изменение высоты компонента ColorPropertyRG
                                в зависимости от количества свойств}
end;
```

Динамическое создание компонента типа TTabSet. Для визуального отображения цветовых установок выбранного в Combobox1 элемента нужно создать объект соответствующего типа и отобразить на форме. При выборе значения «TabSet» эти действия выполняет процедура NewTabset:

```
procedure TMainF.NewTabset;
var i: 0..3;
begin
    Tabset:=TTabSet.Create(MainF);
    with Tabset do
```

```

begin
    Parent:=MainF;
    Left:=200;
    Top:=5;                                {свойства left и top используются
для позиционирования компонента внутри родительского компонента}
    DitherBackground:=true;                {значение свойства, равное true,
устанавливает цвет фона компонента Tabset светлее на 50 %}
    Tabs.Add('Tab_1');
    Tabs.Add('Tab_2');
    Tabs.Add('Tab_3');                      {добавление трех закладок в Tabset}
    TabIndex:=0;                           {выделение первой закладки}

    BackGroundColor:= getsyscolor(COLOR_BTNFACE);      {●}
    SelectedColor:= clWhite;                            {●}
    UnSelectedColor:= getsyscolor
        (TColor(clBtnFace xor $80000000));              {●}
    Font.Color:= getsyscolor
        (TColor(clWindowtext xor $80000000));           {●}

    ColorArray[0].OriginalColor:=BackGroundColor;
    ColorArray[1].OriginalColor:=SelectedColor;
    ColorArray[2].OriginalColor:=UnSelectedColor;
    ColorArray[3].OriginalColor:=Font.Color;
    for i:=0 to 3 do

ColorArray[i].CurrentColor:=ColorArray[i].Originalcolor;
end;
// LoadTabsetSetting;                        { *}
end;

```

Поясним операторы процедуры.

Переменная `Tabset` содержит адрес компонента. Опишите переменную `Tabset` типа `TTabset` в разделе `private` описания класса `TMainF`.

Создание экземпляра любого класса осуществляет конструктор. Оператор

```
Tabset:=TTabSet.Create(MainF);
```

вызывает конструктор класса `TTabset`. Параметр конструктора определяет владельца компонента. Владелец созданного компонента `Tabset` будет форма — `MainF`.

Самый важный этап при динамическом создании компонента — это установка свойства `Parent`. Это свойство помещает текущий компонент в список `Controls` родителя. Компонент-предок использует список `Controls` для выдачи команды нарисовать себя всем принадлежащим ему компонентам. Поэтому если

компонента нет в родительском списке Controls, то он не появится при выполнении приложения. Примерами компонентов, которые могут владеть другими компонентами, являются Panel и GroupBox. Оператор

```
Tabset.parent:=MainF
```

определяет форму родительским компонентом.

Операторы {●} задают цветовые настройки компонента Tabset.

В операционной системе Windows для всех элементов (рабочий стол, заголовок активного окна, полоса прокрутки и др.) определены цветовые константы, каждая из которых имеет свое имя (COLOR\_WINDOW, COLOR\_ACTIVECAPTION, COLOR\_SCROLLBAR и др.). Значения этих констант изменяются в зависимости от выбранной схемы оформления (Пуск\Программы\Настройка\Панель управления\Экран\Оформление).

В Object Pascal соответствующие цветовые константы определяются в модуле C:\Program Files\Borland\Delphi5\Source\Vcl\graphics.pas следующим образом:

```
clScrollBar = TColor(COLOR_SCROLLBAR or $80000000);  
clActiveCaption = TColor(COLOR_ACTIVECAPTION or $80000000);  
clWindow = TColor(COLOR_WINDOW or $80000000); и т. д.
```

Функция getsyscolor определяет текущее значение заданной в параметре цветовой константы. Выражения getsyscolor(COLOR\_BTNFACE) и getsyscolor(TColor(clBtnFace xor \$80000000)) определяют одно и то же цветовое значение. В первом выражении параметр задан цветовой константой Windows, во втором эта константа получена из константы, определенной в Object Pascal обратным преобразованием.

Для сохранения текущих цветов определим новый тип ColorRec, который является записью, состоящей из двух значений типа TColor. OriginalColor представляет собой значение цвета, определенное для элемента операционной системой или Ini-файлом. CurrentColor — это значение цвета, задаваемое элементу после нажатия кнопки Set. При нажатии кнопки Save значения из поля CurrentColor переписываются в поле OriginalColor и сохраняются в Ini-файле.

Этот тип является вспомогательным и используется только подпрограммами модуля. В разделе implementation опишите тип и введите переменную этого типа:

**type**

```
ColorRec = record
  OriginalColor: TColor;
  CurrentColor: TColor;
end;
```

**var**

```
ColorArray: array[0..3] of ColorRec;
```

Начальные значения массива определяются операторами:

```
ColorArray[0].OriginalColor:=BackColor;
ColorArray[1].OriginalColor:=SelectedColor;
ColorArray[2].OriginalColor:=UnSelectedColor;
ColorArray[3].OriginalColor:=Font.Color;
for i:=0 to 3 do
  ColorArray[i].CurrentColor:=ColorArray[i].Original.color;
```

Оператор {\*} будет рассмотрен позднее, он предназначен для считывания цветовых значений из Ini-файла.

*Эксперимент.* Сохраните и запустите проект. Убедитесь, что при выборе элемента Tabset из списка Combobox1 на форме появляется объект Tabset, а в ColorPropertyRG появляются его свойства. ♦

Выбор элемента в ColorPropertyRG. В прямоугольнике CurrentShape нужно установить цвет, соответствующий выбранному свойству, а также установить значения в полосах прокрутки и окнах редактирования.

```
procedure TMainF.ColorPropertyRGClick(Sender: TObject);
begin
  case Combobox1.ItemIndex of 0: {количество и названия свойств
                                зависят от выбранного компонента}
    case ColorPropertyRG.ItemIndex of
      0: CurrentShape.Brush.Color:=Tabset.BackColor;
      1: CurrentShape.Brush.Color:=Tabset.SelectedColor;
      2: CurrentShape.Brush.Color:=Tabset.UnSelectedColor;
      3: CurrentShape.Brush.Color:=Tabset.Font.Color;
    end;
  end;
  SetScrollbars(CurrentShape.Brush.Color); {изменить значения
                                             в полосах прокрутки и окнах редактирования}
end;
```

Опишем процедуру SetScrollbars, которая устанавливает значения в полосах прокрутки. В разделе implementation модуля опишем 32-битовые беззнаковые шестнадцатеричные целые кон-

станты, которые будут использоваться для выделения отдельных составляющих цвета из объектов TColor:

**Const**

```
RedMask = $800000FF;  
GreenMask = $8000FF00;  
BlueMask = $80FF0000;
```

**procedure** TMainF.SetScrollbars(c: TColor);

**begin**

```
RedSB.Position:=c and redMask;  
RedSE.Value:=RedSb.Position;  
                {выделение красной составляющей цвета c}
```

```
GreenSB.Position:=(c and GreenMask) shr 8;  
GreenSE.Value:=GreenSb.Position;  
                {выделение зеленой составляющей цвета c}
```

```
BlueSB.Position:=(c and BlueMask) shr 16;  
BlueSE.Value:=BlueSb.Position;  
                {выделение синей составляющей цвета c}
```

**end;**

*Эксперимент.* Запустите приложение. Убедитесь в том, что цвет компонента CurrentShape изменяется в соответствии с выбранным в компоненте ColorPropertyRG свойством Tabset. ♦

Изменение значения в полосе прокрутки. Выделите компоненты RedSb, GreenSB, BlueSb и создайте для них обработчик события OnChange:

**procedure** TMainF.ColorSbChange(Sender: TObject);

**begin**

```
with Sender as TScrollbar do  
    EditControls[Tag].Text:=IntToStr(Position);  
    CurrentShape.Brush.Color:=RGB ( RedSB.Position,  
    GreenSB.Position, BlueSB.Position);
```

**end;**

Первый оператор процедуры синхронизирует значение полосы прокрутки с соответствующим ей окном редактирования. Для определения необходимого окна редактирования используется свойство Tag. Проверьте, чтобы значения свойства Tag соответствующих компонентов совпадали. Напомним, что для удобства программирования окна редактирования были объединены в массив EditControls.

Второй оператор процедуры, используя функцию RGB, вычисляет значения цвета по значениям, установленным в полосах прокрутки, и перекрашивает прямоугольник CurrentShape.

**Эксперимент.** Убедитесь, что при изменении положения «бегунка» в полосах прокрутки изменяется цвет компонента CurrentShape. ♦

При активизации кнопки Set необходимо сохранить новое значение цвета в поле CurrentColor массива цветов и отобразить этот цвет в компоненте:

```
procedure TMainF.SetBtnClick(Sender: TObject);
begin
  If ColorPropertyRG.ItemIndex<>-1 then {если компонент выбран}
    ColorArray[ColorPropertyRG.ItemIndex].CurrentColor:=
    CurrentShape.Brush.Color;      {в массиве ColorArray запоминаем
    новое текущее значение для свойства, выбранного в ColorPropertyRGB}

  case ComboBox1.ItemIndex of
    0: ChangeTabsetColor;          {отображаем новое цветовое значение
                                   выбранного свойства в компоненте}

  end;
end;
```

Для отображения новых значений свойств компонента Tabset используется массив ColorArray:

```
procedure TMainF.ChangeTabsetColor;
begin
  with Tabset do
  begin
    BackGroundColor:=ColorArray[0].currentColor;
    SelectedColor:=ColorArray[1].currentColor;
    UnSelectedColor:=ColorArray[2].currentColor;
    Font.Color:=ColorArray[3].currentColor;
  end;
end;
```

Опишите метод ChangeTabsetColor в разделе private описания класса TMainF.

При активизации кнопки Отменить, нужно восстановить исходные значения свойств из массива ColorArray:

```
procedure TMainF.ResetBthClick(Sender: TObject);
var i: 0..3;
begin
```

```
if Combobox1.itemindex<>-1 then
begin
    {если компонент выбран}
    for i:=0 to ColorPropertyRG.Items.Count-1 do
        with ColorArray[i] do CurrentColor:=OriginalColor;
            {восстанавливаем значения свойств в массиве ColorArray}
    case Combobox1.itemindex of
        0: ChangeTabSetColor;
            {восстанавливаем значения свойств компонента Tabset}
    end;
    ColorPropertyRG.OnClick(ColorPropertyRG);
        {отображаем восстановленный цвет выбранного
        в ColorPropertyRG свойства в прямоугольнике CurrentShape}
end;
end;
```

**Эксперимент.** Сохраните проект. Запустите, попробуйте изменить цветовые настройки компонента Tabset. Щелчок на кнопке Установить влияет на изменение внешнего вида компонента Tabset. Происходит ли восстановление цветовых значений после щелчка на кнопке Отменить? ♦

Кнопка Сохранить предназначена для сохранения текущих цветовых значений в Ini-файле:

```
procedure TMainF.SaveBtnClick(Sender: TObject);
var i: 0..3;
    Section: string;
        {вспомогательная переменная,
        определяющая раздел Ini-файла,
        в котором будет сохранены данные }
begin
    if Combobox1.itemindex<>-1 then
begin
    {если компонент выбран}
    case Combobox1.itemindex of
        0: Section:='Tabset'; {определяем название секции Ini-файла}
    end;
    for i:=0 to ColorPropertyRG.items.Count-1 do
        IniFile.WriteString(Section,
            string(ColorPropertyRG.Items[i]),
            IntToStr(ColorArray[i].CurrentColor));
        {запись данных в Ini-файл}
    end;
end;
```

**Эксперимент.** Убедитесь, что после щелчка на кнопке Сохранить был создан файл с именем 'с:\Colors.ini'. Посмотрите структуру созданного файла. Измените значение одного из свойств. ♦

Опишем процедуру чтения из Ini-файла цветовых параметров компонента Tabset:

```

procedure TMainF.LoadTabsetSetting;
var IniValueList: TStringList;
    i: Integer;
begin
    IniValueList:=TStringList.Create;
                                {создаем объект типа TStringList
                                для считывания цветовых значений из Ini-файла}
try
    IniFile.ReadSectionValues('Tabset',IniValueList);
                                {чтение цветовых значений свойств}
for i:=0 to IniValueList.Count-1 do      {заполняем массив
                                цветовыми значениями}

with ColorArray[i] do begin
    CurrentColor:=StrToInt
        (IniValueList.Values[ColorPropertyRG.items[i]]);
    OriginalColor:=CurrentColor;
end;
finally
    IniValueList.Free;
                                {освобождаем память,
                                выделенную под объект типа TStringList}
end;
    ChangeTabsetColor;
end;
                                {}

```

*Эксперимент.* Удалите комментарии в методе NewTabset, запустите проект. Что происходит при выборе компонента Tabset? ♦

### Задания для самостоятельного выполнения

**5.12.** Допишите приложение ColorIni (упр. 5.4.1), дополнив его следующими функциями:

- добавьте возможность изменения свойств компонентов ListBox и Edit;
- синхронизируйте изменение значений в компонентах SpinEdit с соответствующими им компонентами Scrollbar.

**5.13.** Модифицируйте программу «Блокнот»: добавьте возможность настройки компонентов Tabset, ListBox, Edit по запросу пользователя.

**5.14.** Напишите приложение «Кулинарная книга». Четыре страницы содержат информацию:

- закуски и холодные блюда;

- ☐ первые блюда;
- ☐ вторые блюда;
- ☐ десерты.

На каждой закладке представлен список названий блюд. При выборе какого-либо названия из списка отображается рецепт приготовления выбранного блюда. Рецепты всех блюд содержатся в файле (файлах).

Предусмотрите возможность добавления рецептов в Кулинарную книгу, а также изменения существующих рецептов и удаления ненужных.

**5.15.** Напишите приложение-тест «Проверка знания английских слов». При запуске программы тестирующему предлагается ввести фамилию и имя. Далее появляется окно теста, содержащее два столбика слов: первый — английские слова, второй — русские, причем столбик с русскими словами несколько длиннее по количеству слов. Тестируемый должен перетаскивать английское слово из первого столбика на соответствующее ему русское слово во втором (после перетаскивания английское слово исчезает). Слова во второй столбик вводятся в произвольном порядке (т.е. порядок русских и английских слов произвольный).

Приложение подсчитывает количество правильных ответов и после окончания теста выдает результат. Приложение генерирует файл отчета, содержащий фамилию и имя тестируемого, количество правильных ответов и общее число слов. Файл должен содержать информацию о каждом, кто проходил тест.

### Вопросы для повторения

1. Для чего предназначены экземпляры классов TStrings, TList, TStringList?
2. Каково значение свойства parent компонента RecordEdit, AddBtn, ListBox1 в приложении Блокнот?
3. Что такое Ini-файл? Какова структура Ini-файлов? Экземпляры какого класса рациональнее использовать для считывания данных из Ini-файлов? Объясните.
4. Что такое исключительная ситуация в Delphi? Какие классы исключительных ситуаций Вам известны? Какие операторы используются для обработки исключительных ситуаций?

# 6

## Сетки строк

### 6.1. Класс TDrawGrid

Компонент DrawGrid предназначен для создания таблицы, в ячейках которой расположены данные. Компонент обеспечивает двумерное представление данных, упорядоченных по строкам и столбцам.

Таблица делится на две части — фиксированную и рабочую. Фиксированная часть служит для показа заголовков столбцов/строк и для ручного управления их размерами. Обычно фиксированная часть занимает крайний левый столбец и самый верхний ряд таблицы. Она может содержать произвольное количество столбцов и рядов, причем эти величины можно изменять как в процессе разработки, так и программно. Рабочая часть состоит из ячеек, в которых находятся данные. Если рабочая часть не помещается целиком в пределах окна компонента, то у компонента автоматически появляются полосы прокрутки. При прокрутке рабочей области фиксированная часть не исчезает, но меняется ее содержимое — заголовки строк и рядов.

Заносить данные в ячейки таблицы можно только в ходе работы программы.

#### Свойства компонента DrawGrid

У компонента есть множество свойств, некоторые из них доступны уже в процессе разработки программы, остальные — только в ходе ее выполнения.

Рассмотрим основные свойства, доступные во время разработки.

BorderStyle	определяет наличие или отсутствие внешней рамки таблицы;
ColCount	устанавливает количество столбцов таблицы, включая столбцы фиксированной части;
DefaultColWidth	определяет ширину столбца по умолчанию;

DefaultDrawing	при значении, равном True, происходит автоматическая прорисовка служебных элементов таблицы (фиксированной зоны, фона и прямоугольника сфокусированной ячейки и т. д.). Если свойство установлено в False, то прорисовки этих элементов необходимо определять в обработчике события OnDrawCell;
DefaultRowHeight	содержит значение высоты строки по умолчанию;
FixedColor	устанавливает цвет фиксированной зоны;
FixedCols	определяет количество столбцов фиксированной зоны;
FixedRows	определяет количество строк фиксированной зоны;
RowCount	устанавливает количество строк таблицы.

Дополнительно к перечисленным в таблице свойствам необходимо обратить особое внимание на свойство **Options**, определяющее некоторые особенности поведения компонента DrawGrid. Свойство Options определяется следующим образом:

#### type

```
TGridOption = (goFixedVertLine, goFixedHorzLine,  
               goVertLine, goHorzLine, goRangeSelect,  
               goDrawFocusSelected, goRowSizing, goColSizing,  
               goRowMoving, goColMoving, goEditing, goTabs,  
               goRowSelect, goAlwaysShowEditor, goThumbTracking);  
TGridOptions = set of TGridOption;  
property Options: TGridOptions;
```

Каждое значение характеризует особенности поведения таблицы в процессе работы приложения:

goAlwaysShowEditor	значение, равное True, позволяет редактировать сфокусированную (выделенную) ячейку. Редактирование возможно после выбора ячейки клавишей Tab (Tab+Shift). Подсвойство игнорируется, если goEditing установлено в False;
--------------------	---

<code>goColMoving</code>	значение, равное <code>True</code> , позволяет перемещать столбцы (для этого нужно нажать левую клавишу мыши на фиксированной ячейке перемещаемого столбца и, удерживая клавишу нажатой, переместить столбец на новое место);
<code>goColSizing</code>	контролирует изменение ширины
<code>goDrawFocusSelected</code>	включение этого свойства приводит к выделению ячейки, в которой находится фокус. Если же свойство равно <code>False</code> , то ячейка, имеющая фокус, не выделяется никаким цветом;
<code>goEditing</code>	значение <code>True</code> свойства позволяет редактировать содержимое ячейки (свойство игнорируется, если значение <code>goRowSelect</code> равно <code>True</code> ). Редактирование начинается после щелчка на ячейке клавишей мыши или нажатия клавиши <code>F2</code> и завершается при щелчке на другой ячейке или нажатии <code>Enter</code> ;
<code>goFixedHorzLine</code>	включение свойства заставляет прорисовывать горизонтальные полосы для разделения строк в фиксированной области;
<code>goFixedVertLine</code>	установление значения в <code>True</code> заставляет отображать вертикальные полосы для разделения столбцов в фиксированной области;
<code>goHorzLine</code>	при значении <code>False</code> будут отсутствовать горизонтальные линии в рабочей области;
<code>goRangeSelect</code>	для того чтобы пользователь мог выбирать насколько ячеек одновременно, данное свойство следует установить в <code>True</code> (значение свойства будет игнорироваться, если свойство <code>goEditing</code> равно <code>True</code> );

<code>goRowMoving</code>	свойство аналогично <code>goColMoving</code> , разрешает перемещение строки;
<code>goRowSelect</code>	значение <code>True</code> этого свойства позволяет выделять все (а не отдельные) ячейки строки, в этом случае будет игнорироваться свойство <code>goAlways ShowEditor</code> ;
<code>goRowSizing</code>	включение свойства позволяет вручную (мышью) изменять высоту строк;
<code>goTabs</code>	если свойство установлено в <code>True</code> , то можно выбирать ячейки клавишей <code>Tab</code> ( <code>Shift+Tab</code> );
<code>goThumbTracking</code>	ячейки таблицы будут обновляться в процессе использования полосы прокрутки. Если значение равно <code>False</code> , то обновление ячеек произойдет только после окончания прокрутки;
<code>goVertLine</code>	при значении свойства, равном <code>False</code> , в рабочей области отсутствуют вертикальные линии.

Кроме перечисленных свойств, во время выполнения программы становятся доступными еще некоторые свойства.

Свойство **Col/Row** определяет номер столбца/строки сфокусированной (выделенной) ячейки. Нумерация и строк и столбцов начинается с нуля, включая строки и столбцы фиксированной зоны.

Номер самого левого столбца, видимого в прокручиваемой зоне ячеек, содержится в **LeftCol**, а номер самого верхнего ряда — в свойстве **TopRow**.

Свойство **EditorMode** отвечает за возможность редактирования ячеек (свойство будет игнорироваться, если `goAlwaysShowEditor` равно `True` или `goEditing` равно `False`). Когда во время работы программы пользователь нажимает клавишу `F2`, **EditorMode** устанавливается в `True` автоматически. После того как пользователь нажимает клавишу ввода, свойство принимает значение `False`.

Свойство **Selection** позволяет определить координаты текущего выделения. Описывается свойство следующим образом:

**type**

TGridCoord = **record**

X: Longint;

Y: Longint;

**end;**

TGridRect = **record**

**case** Integer **of**

0: (Left, Top, Right, Bottom: Longint);

1: (TopLeft, BottomRight: TGridCoord);

**end;**

**property** Selection: TGridRect;

Свойство Selection определяет группу выделенных ячеек в координатах левая верхняя и правая нижняя ячейки. После выделения сфокусированной окажется правая нижняя ячейка.

### Методы компонента DrawGrid

Экранные координаты прямоугольника ячейки можно получить по номерам столбца ACol и ряда ARow с помощью метода CellRect:

**function** CellRect (ACol, ARow: Longint): TRect;

где тип TRect — это

**type**

TRect = **record**

**case** Integer **of**

0: (Left, Top, Right, Bottom: Integer);

1: (TopLeft, BottomRight: TPoint);

**end;**

TPoint = **record**

X: Longint;

Y: Longint;

**end;**

Получить номер столбца ACol и номер строки ARow по экраным координатам (X,Y) точки можно с помощью метода MouseToCell:

**procedure** MouseToCell (X, Y: Integer; **var** ACol, ARow: Longint);

Например, если необходимо определить, по какой ячейке был произведен щелчок мышью, то можно воспользоваться обработчиком события OnMouseDown:

**procedure** TForm1.DrawGrid1MouseDown (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

**var** ACol, Arow: Integer;

{переменные для хранения номера столбца/строки}

**begin**

```
DrawGrid1.MouseToCell(x, y, Acol, Arow);  
    {используя параметры события OnMouseDown,  
    определяем номера строки и столбца}
```

**end;**

### События компонента DrawGrid

Событие **OnColumnMoved** возникает при перемещении столбца. Оно происходит только тогда, когда подсвойство **goColMoving** равно **True**. Заголовок обработчика этого события имеет вид:

```
procedure TForm1.DrawGrid1ColumnMoved(Sender: TObject;  
FromIndex, ToIndex: Integer);
```

Параметр **FromIndex** содержит «старый» индекс столбца, а **ToIndex** — «новый» индекс перемещаемого столбца.

Событие **OnRowMoved** возникает при перемещении строки. Оно происходит только тогда, когда **goRowMoving** включено в свойство **Options**. Заголовок обработчика этого события имеет вид:

```
procedure TForm1.StringGrid1RowMoved(Sender: TObject;  
FromIndex, ToIndex: Integer);
```

Событие **OnTopLeftChanged** происходит при изменении значения **TopRow** или **LeftCol** в результате прокрутки рабочей зоны:

```
procedure TForm1.DrawGrid1TopLeftChanged(Sender: TObject);
```

Событие **OnSelectCell** возникает при попытке выделить ячейку с табличными координатами (**ACol**, **ARow**). В параметре **CanSelect** обработчик сообщает о возможности выделения ячейки. Установите его значение равным **False**, чтобы пользователь не мог выделять ячейку. Событие описывается следующим образом:

```
procedure TForm1.DrawGrid1SelectCell(Sender: TObject;  
ACol, ARow: Integer; var CanSelect: Boolean);
```

Событие **OnSetEditText** возникает по завершении редактирования ячейки с координатами (**ACol**, **ARow**). В параметре **Value** обработчик получает результат ввода или редактирования текста. Событие произойдет только в том случае, когда свойство **Options** содержит значение **goEditing**. Описывается событие так:

```
procedure TForm1.DrawGrid1SetEditText(Sender: TObject;  
ACol, ARow: Integer; const Value: String);
```

События **OnGetMaskEdit** и **OnGetEditText** возникают при редактировании текста в ячейке с табличными координатами (ACol, ARow). В параметре Value первого события обработчик должен вернуть шаблон для редактора TEditMask. Параметр Value для события OnGetEditText должен содержать текстовую информацию для редактора TEditMask. Описание событий выглядит следующим образом:

```
procedure TForm1.DrawGrid1GetEditMask(Sender: TObject;  
    ACol, ARow: Integer; var Value: String);
```

```
procedure TForm1.DrawGrid1GetEditText(Sender: TObject;  
    ACol, ARow: Integer; var Value: String);
```

Событие **OnDrawCell** происходит всякий раз, когда необходимо прорисовать ячейку таблицы. Обработчик данного события полностью берет на себя ответственность за размещение в каждой ячейке нужных данных. Описывается следующим образом:

```
procedure TForm1.DrawGrid1DrawCell(Sender: TObject;  
    ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
```

Событие происходит в случае необходимости перерисовки ячейки с номером столбца ACol и номером строки ARow. Параметр Rect определяет прямоугольник прорисовки, а State — состояние ячейки (gdSelected — ячейка выделена, gdFocused — ячейка сфокусирована, gdFixed — ячейка принадлежит фиксированной зоне таблицы). Для прорисовки используется свойство Canvas.

**Упражнение 6.1.1.** Создайте приложение, которое позволяет просматривать символы системных шрифтов.

### Решение

Создайте новый проект. Сохраните новое приложение в папке Fonts, файл модуля — под именем Main.pas, файл проекта — fonts.dpr.

*1-й этап. Визуальное проектирование*

Измените значения свойств формы следующим образом:

Name	Fonts
Caption	Символы

Положите на форму компонент TPanel:

Align	alTop
Caption	Отрезки

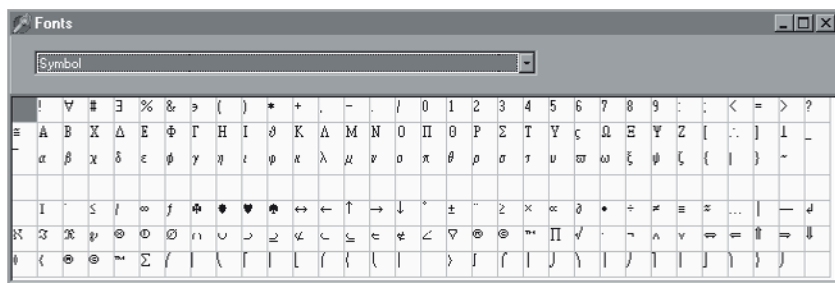


Рис. 6.1.1

Разместите на компоненте Panel1 компонент ComboBox (рис. 6.1.1.). Пусть имя этого компонента будет FontListCB.

Далее расположите на форме компонент DrawGrid:

Name	FontDG
Align	alClient
RowCount	7
ColCount	32
FixedCols	0
FixedRows	0
DefaultColWidth	20
DefaultRowHeight	20

Измените размеры формы так, чтобы сетка не имела полос прокрутки, а вокруг ячеек не было пустого пространства.

### 2-й этап. Разработка программного кода

Для того чтобы содержимое каждой ячейки перерисовывалось, создадим обработчик события OnDrawCell для компонента FontDG. Для изображения символов шрифта воспользуемся свойством Canvas компонента FontDG. Непосредственно нам понадобится метод TextRect свойства Canvas. Этот метод используется для вывода текстовой информации в определенной ячейке. Обработчик события будет выглядеть так:

```
procedure TFonts.FontDGDrawCell(Sender: TObject; ACol,
ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
  with FontDG.Canvas do
    TextRect (Rect, Rect.Left, Rect.Top,
```

```
Char ((ARow+1)*32+ACol));
```

{параметры ячейки для вывода символов шрифта берутся из параметров обработчика события, а символ шрифта для отображения в ячейке определяется в зависимости от строки и столбца}

```
end;
```

**Эксперимент.** Сохраните проект. Убедитесь, что в ячейках таблицы отображаются символы системного шрифта, установленного по умолчанию. ♦

Для выбора шрифта воспользуемся компонентом FontListCB. Для того чтобы данный компонент содержал все экранные шрифты, надо при создании формы занести их в список.

Названия всех экранных шрифтов можно узнать с помощью глобальной переменной Screen типа TScreen. Данная переменная автоматически добавляется во все приложения Delphi. Переменная Screen содержит информацию о текущем состоянии экрана приложения: названия форм и модулей данных, которые используются приложением; данные об активной форме и компонентах, используемых этой формой; размер и разрешение используемого экрана; информацию о доступных приложению курсорах и шрифтах.

Информация о доступных приложению шрифтах содержится в свойстве Font, принадлежащем переменной Screen.

Создадим следующий обработчик:

```
procedure TFonts.FormCreate(Sender: TObject);
```

```
begin
```

```
  with FontListCB do
```

```
  begin
```

```
    Items := Screen.Fonts;
```

{в свойстве Fonts переменной Screen содержатся названия всех экранных шрифтов}

```
    ItemIndex := Items.IndexOf(Font.Name);
```

{свойства IndexOf содержит номер строки в списке FontListCB, которая выбрана, и, соответственно, содержит имя текущего шрифта}

```
  end;
```

```
end;
```

**Эксперимент.** Сохраните и запустите проект. Компонент FontDG содержит символы шрифта, установленного в FontListCB. Сколько шрифтов установлено на компьютере? Что происходит при выборе другого шрифта?

Для того чтобы связать значение имени шрифта у FontDG и FontListCB, создадим еще один обработчик события:

```
procedure TFonts.FontListCBClick(Sender: TObject);  
begin  
  FontDG.Font.Name := FontListCB.Text;  
end;
```

*Эксперимент.* Сохраните и запустите проект. Что происходит при изменении шрифта? ♦

## 6.2. Класс TStringGrid

Компонент StringGrid предназначен для создания таблиц, в ячейках которых располагаются произвольные текстовые строки. Класс TStringGrid является прямым потомком класса TDrawGrid, от которого им унаследовано большинство свойств и методов. У данного класса появляется лишь несколько новых свойств.

Cells[ACol, ARow: Integer]	определяет содержимое ячейки с координатами (ACol, ARow);
Cols[Index: Integer]	содержит все строки колонки с номером Index;
Objects [ACol, ARow: Integer]	обеспечивает доступ к объекту, связанному с ячейкой (ACol, ARow);
Rows[Index: Integer]	содержит все строки столбца с номером Index.

Хорошей иллюстрацией работы компонента типа TStringGrid служит программная реализация игры «Жизнь».

**Упражнение 6.2.1.** Игра «Жизнь». Идея игры состоит в том, чтобы, начав с какого-нибудь простого расположения фишек (организмов), расставленных по различным клеткам доски, проследить за эволюцией исходной позиции под действием «генетических законов» Конуэя, которые управляют рождением, гибелью и выживанием фишек.

Генетические законы игры сводятся к следующему.

- **Выживание.** Каждая фишка, у которой имеются две или три соседние фишки, выживает и переходит в следующее поколение.
- **Гибель.** Каждая фишка, у которой оказывается больше трех соседей, погибает, т. е. снимается с доски, из-за перенаселенности. Каждая фишка, вокруг которой свободны все соседние клетки или же занята только одна клетка, погибает от одиночества.

- Рождение. Если число фишек, с которыми граничит какая-нибудь пустая клетка, в точности равно трем (не больше и не меньше), то на этой клетке происходит рождение нового «организма», т. е. следующим ходом на нее ставится одна фишка.

Таким образом, гибель и рождение всех «организмов» происходят одновременно. Вместе взятые, они образуют одно поколение или один «ход» в эволюции начальной конфигурации. Ходы Конуэй рекомендует делать следующим образом:

- 1) начать с конфигурации, целиком состоящей из черных фишек;
- 2) определить, какие фишки должны погибнуть, и положить на каждую из обреченных фишек по одной черной фишке;
- 3) найти все свободные клетки, на которых должны произойти акты рождения, и на каждую из них поставить по одной фишке белого цвета;
- 4) выполнив все эти указания, еще раз внимательно проверить, не сделано ли каких-либо ошибок, затем снять с доски все погибшие фишки (т. е. столбики из двух фишек), а всех новорожденных (белые фишки) заменить черными фишками.

Пусть в нашей игре фишки  $n$ -го поколения будут умирать не сразу, а лишь после появления  $(n+2)$ -го поколения. Фишки разных поколений будут отличаться друг от друга цветом.

### Решение

Создайте новый проект Delphi. Сохраните новое приложение в папке GameLife — файл модуля под именем Main.pas, файл проекта — GameLife.dpr.

*1-й этап. Визуальное проектирование (рис. 6.2.1)*

Измените значения свойств формы следующим образом:

Name	LifeFrm
Caption	Жизнь

Положите на форму компонент ToolBar (панель инструментов) со страницы Win32. Измените его свойство следующим образом:

EdgeBorders | [ebTop,ebBottom]

Сейчас панель инструментов будет находиться в самом верху формы, а ее верхняя и нижняя стороны будут выделены вдавленной линией.

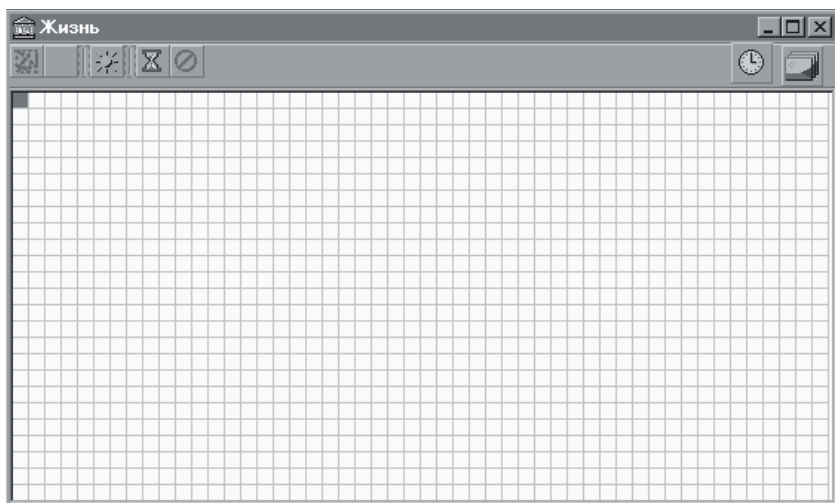


Рис. 6.2.1

Разместите на панели инструментов 5 кнопок. Для этого щелкните правой кнопкой мыши на панели инструментов и в появившемся контекстном меню выберите пункт `NewButton`. На панели инструментов появится кнопка типа `TToolButton`. С помощью этой кнопки мы будем задавать произвольную первоначальную конфигурацию расположения фишек, поэтому назовите ее `RandomTBt`.

Добавьте еще одну кнопку и назовите ее `NewTBt`. Эта кнопка понадобится для очистки игрового поля.

Поместите разделитель на панель инструментов. Для этого опять вызовите контекстное меню панели инструментов и выберите в нем пункт `NewSeparator`.

Поместите еще одну кнопку на `ToolBar1` (ее свойство `Name` сделайте равным `FadeTBt`), разделитель и еще две кнопки типа `TToolButton` (с именами соответственно `RunTBt` и `StopTBt`). С помощью этих кнопок мы будем управлять ходом игры. Кнопка `FadeTBt` будет регулировать отображение трех или одного поколения фишек. Кнопки `RunTBt` и `StopTBt` предназначены для начала и остановки игры.

Для того чтобы поместить изображения на кнопки, нам понадобится компонент `ImageList` со страницы `Win32`. Этот компонент может содержать серию изображений.

Создадим изображение для кнопки RandomTBt. Воспользуемся программой ImageEditor, входящей в состав Delphi.

- Запустите ImageEditor, выполнив команду Пуск/Программы/Borland Delphi 5/Image Editor.
- Для создания пиктограммы выберите в пункте меню File команду New Bitmap File (.bmp). В диалоговом окне Bitmap Properties задайте размеры изображения (Height и Width) равными 20.
- С помощью инструментов программы Image Editor создайте изображение для кнопки RandomTBt.
- Сохраните в папке Life под названием Bitmap1.bmp.

**Задание для самостоятельного выполнения.** Создайте изображение для кнопки NewTBt и сохраните его под именем Bitmap2.bmp.

Для остальных кнопок воспользуемся стандартным набором изображений Delphi.

Поместите изображения, предназначенные для кнопок приложения, в компонент ImageList1.

- Щелкните правой кнопкой мыши на этом компоненте и в контекстном меню выберите пункт ImageList Editor...
- В диалоговом окне редактора изображений LifeFrm. Image1List ImageList щелкните на кнопке Add.
- В диалоговом окне открытия файла выберите файл Bitmap1.bmp с изображением для кнопки RandomTBt. В редакторе изображений появится выбранное изображение под индексом 0. Аналогичным образом добавьте изображение для кнопки NewTBt (оно должно иметь индекс 1).

Изображения для трех остальных кнопок выберите в папке C:/Program Files/Common Files/Borland Shared/Images/Buttons.

- В диалоговом окне редактора изображений LifeFrm. Image1List ImageList щелкните на кнопке Add, выберите файл C:/ Program Files/ Common Files/ Borland Shared/ Images/ Buttons/ Day.bmp. После щелчка на кнопке Open появится окно (рис. 6.2.2) с сообщением о том, что размер изображения в day.bmp больше, чем размер, определенный в ImageList. Нужно разделить на 2 обособленных изображения? Щелкните на кнопке No.
- Установите Options диалогового окна редактора изображений в значение Crop — срезать.

Добавьте изображения для кнопок FadeTBt (индекс равен двум), RunTBt (индекс равен трем) и StopTBt (индекс равен четырем).

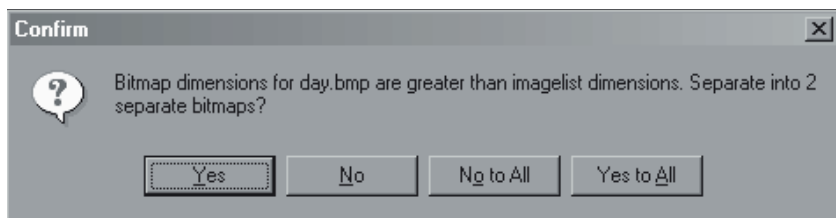


Рис. 6.2.2

Чтобы показать изображения на кнопках, установите свойство Images компонента ToolBar1 в ImageList1, а у каждой кнопки свойству ImageIndex присвойте индекс соответствующего изображения в редакторе изображений (для кнопки RandomTBt — 0, для NewTBt — 1 и т. д.).

**Эксперимент.** Сохраните проект. Запустите, на панели инструментов должны появиться кнопки с созданными вами изображениями. ◆

Для изображения жизни организмов нам понадобится компонент StringGrid. Установите его свойства следующим образом:

Align	alClient
ColCount	50
DefaultColWidth	10
DefaultRowHeight	10
FixedCols	0
FixedRows	0
Name	LifeSGd
RowCount	25

Измените размеры формы так, чтобы у компонента LifeSGd не было полос прокрутки.

Смену поколений организмов будем отображать через каждую секунду. Воспользуемся компонентом Timer (страница System).

Класс `TTimer` поддерживает единственный обработчик события `OnTimer`. Это событие появляется через интервал времени, заданный свойством `Interval`. Таким образом, если свойство `Interval` установлено в 1000 (миллисекунд), то обработчик события `OnTimer` будет вызываться каждую секунду.

В классе `TTimer` определено свойство `Enabled`, значение `False` которого приводит к игнорированию событий `OnTimer`, значение `True` позволяет обрабатывать события `OnTimer`.

Компонент `Timer` — это невизуальный компонент `Delphi`, т. е. отображается он только на этапе проектирования приложения. Положите компонент `Timer` на форму, установите значения свойств следующим образом:

<code>Enabled</code>	<code>True</code>
<code>Interval</code>	<code>1000</code>

Итак, визуальное проектирование приложения завершено. Сохраните проект.

### *2-й этап. Написание программного кода*

Опишем глобальные переменные.

В разделе `Const` раздела `Interface` опишите две глобальные константы

```
MX = 50;                                {количество столбцов в LifeSGd}
MY = 25;                                {количество строк в LifeSGd}
```

В разделе `Private` класса `LifeSGd` опишите глобальную переменную

```
Fade: Boolean;
```

Эта переменная будет отвечать за количество поколений, отображаемых приложением: если значение `Fade` равно `True` — будут отображаться три поколения организмов (фишек), иначе — только живые организмы (фишки), т. е. одно поколение.

Воспользуемся свойством `Cells` компонента `LifeSGd` для хранения состояния организмов колонии. Введем следующие обозначения:

- 0 — фишка мертва;
- 1 — фишка жива;
- 2 — фишка мертва одно поколение (на предыдущем ходе она была жива);
- 3 — фишка мертва два поколения.

Как уже говорилось, генерация каждого нового поколения будет происходить по таймеру, поэтому при создании формы необходимо таймер выключить. Кроме того, пусть в самом начале переменная `Fade` будет равна `False`, а все организмы будут помечены мертвыми. В соответствии с этими утверждениями обработчик события создания формы будет выглядеть так:

```
procedure TLifeFrm.FormCreate(Sender: TObject);  
  var i,j:Integer;  
begin  
  Timer1.Enabled := False;  
  Fade := False;  
  for i :=0 to MX-1 do  
    for j :=0 to MY-1 do LifeSGd.Cells[i,j] := '0';  
end;
```

Рассмотрим процедуру задания произвольной расстановки фишек — событие `OnClick` кнопки `RandomTBt`.

```
procedure TLifeFrm.RandomTBtClick(Sender: TObject);  
  var x,y: Integer;  
begin  
  Randomize; {включаем генератор случайных чисел}  
  for y :=0 to MY-1 do  
    for x :=0 to MX-1 do  
      LifeSGd.Cells[x,y]:=IntToStr(Random(2));  
      {если в ячейку записывается единица, то фишка жива,  
      если в ячейку записывается ноль, то фишка считается мертвой}  
end;
```

Кнопка `NewTBt` используется для очистки содержимого ячеек компонента `LifeSGd`. Поэтому в обработчике события `OnClick` данной кнопки нужно просто значению каждой ячейки присвоить ноль. Кроме того, надо выключить таймер. Обработчик события будет выглядеть так:

```
procedure TLifeFrm.NewTBtClick(Sender: TObject);  
  var y,x: Integer;  
begin  
  for y := 0 to MY-1 do  
    for x := 0 to MX-1 do LifeSGd.Cells[x,y] := '0';  
  Timer1.Enabled := False;  
end;
```

При нажатии кнопки `FadeTBt` будем изменять значение переменной `Fade`, которая показывает, изображать или нет несколько поколений. Если `Fade` равно `True`, то кнопка будет вдавlenной. Создайте следующий обработчик события:

```
procedure TLifeFrm.FadeTBtClick(Sender: TObject);  
begin  
    Fade := not(Fade);  
    FadeTBt.Down := Fade;  
end;
```

Для запуска игры служит кнопка RunTBt. При ее нажатии таймер должен начинать работать, кнопка RunTBt — становиться вдавленной, а кнопка StopTBt — принимать обычный вид. Все эти действия выполняются в данном обработчике:

```
procedure TLifeFrm.RunTBtClick(Sender: TObject);  
begin  
    Timer1.Enabled := True;  
    RunTBt.Down := True;  
    StopTBt.Down := False;  
end;
```

Кнопка StopTBt используется для остановки работы программы. Обработчик события OnClick этой кнопки должен выглядеть так:

```
procedure TLifeFrm.StopTBtClick(Sender: TObject);  
begin  
    Timer1.Enabled := False;  
    StopTBt.Down := True;  
    RunTBt.Down := False;  
end;
```

*Эксперимент.* Сохраните проект. Убедитесь в правильности функционирования написанного кода. ◆

Задавать расположение фишек можно и вручную. Для этого достаточно щелкнуть левой кнопкой мыши по нужной вам ячейке. Запрограммируем такой вариант расстановки фишек. Удобнее всего пометать фишки как живые в событии OnSelectCell компонента LifeSGd:

```
procedure TLifeFrm.LifeSGdSelectCell(Sender: TObject;  
ACol, ARow: Integer; var CanSelect: Boolean);  
begin  
    if LifeSGd.Cells[ACol, ARow]='1'  
    then LifeSGd.Cells[ACol, ARow]:= '0'  
    else LifeSGd.Cells[ACol, ARow]:= '1';  
end;
```

Остановимся подробнее на том, как изменяется цвет каждой ячейки. Известно, что прорисовка ячеек происходит в событии OnDrawCell. Это событие происходит столько раз, сколько яче-

ек содержит компонент `StringGrid` (в задаче — `LifeSGd`). Кроме того, событие вызывается каждый раз при изменении значения свойства `Cells`. Именно поэтому не приходилось вручную перерисовывать `LifeSGd`.

Схема изменения цвета ячейки достаточно проста: обработчик данного события проверяет значение свойства `Cells` ячейки (`ACol`, `ARow`) и в зависимости от него присваивает нужный цвет кисти свойства `Canvas` компонента `LifeSGd`. Живые фишки изображаются синим цветом; предыдущее поколение (фишки, мертвые в течение одного хода) — светло-голубым цветом; фишки, мертвые в течение двух ходов, — светло-фиолетовым цветом; мертвые фишки — белым цветом. Однако если значение переменной `Fade` равно `False`, то отображаются только живые и мертвые клетки (без промежуточных состояний). После того как цвет установлен, остается только заполнить этим цветом нужную ячейку с помощью метода `FillRect` свойства `Canvas`. В результате процедура будет выглядеть следующим образом:

```
procedure TLifeFrm.LifeSGdDrawCell(Sender: TObject; ACol,
ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
  LifeSGd.Canvas.Brush.Color := clWhite;
  case StrToInt(LifeSGD.Cells[ACol, ARow]) of
    1: LifeSGd.Canvas.Brush.Color := clBlue;
    2: if Fade then LifeSGd.Canvas.Brush.Color := clBlue - 13000;
    3: if Fade then LifeSGd.Canvas.Brush.Color := clBlue-17000;
  end;
  LifeSGd.Canvas.FillRect(Rect);
end;
```

***Эксперимент.*** Запустите проект, обратите внимание, что после щелчков мышью по полю таблицы клетки перекрашиваются. ♦

Итак, осталось написать последнюю процедуру, в которой будем определять, какие фишки выживут, а какие умрут. Именно здесь нам и понадобится таймер. На каждый «тик» таймера будет вызываться процедура `OneStep`, в которой оценивается текущее состояние фишек, и в зависимости от него строится новое расположение фишек. Добавьте название процедуры `OneStep` в раздел `public` описания класса формы `TLifeFrm`.

```
...
public
  procedure OneStep;
...
```

Принцип выбора живых фишек следующий: просматриваем поочередно все ячейки; для каждой ячейки считаем количество окружающих ее живых фишек; в зависимости от количества этих фишек заполняем элементы двумерного массива **A** либо нулем, либо единицей; далее просматриваем элементы этого массива **A** и в зависимости от их значений изменяем свойство **Cells** компонента **LifeSGd**.

Для реализации этого принципа нам понадобится двумерный массив **A**, элементы которого соответствуют ячейкам компонента **LifeSGd**. Размерность массива **A** будет аналогична размерности **LifeSGd**. Кроме того, нам будут нужны два массива — константы **DX** и **DY**, содержащие приращение по вертикали и горизонтали соответственно. Эти массивы понадобятся для просмотра соседних с клеткой ячеек.

```

procedure TLifeFrm.OneStep;
const
  DX : array [1..8] of Integer = (-1, 0, 1, 1, 1, 0, -1, -1);
  DY : array [1..8] of Integer = (-1, -1, -1, 0, 1, 1, 1, 0);
var
  i, j, k, Count: Integer;
  A: array [0..MX, 0..MY] of Byte;
begin
  FillChar(A, SizeOf(A), 0);           {обнуляем значения массива A}
  for i:=1 to MX-1 do                   {просматриваем все ячейки}
    for j:=1 to MY-1 do
      begin
        Count:=0;
        {переменная Count используется для подсчета количества
          живых фишек}
        for k:=1 to 8 do
          {просматриваем все ячейки, которые являются соседними
            для ячейки (i, j)}
          if LifeSGd.Cells[i+DX[k], j+DY[k]]='1' then Inc(Count);
        case Count of
          {анализируем полученное значение переменной Count
            и в зависимости от него заполняем массив A}
          0..1, 4..8: A[i, j]:=0;
          2: if LifeSGd.Cells[i, j]='1' then A[i, j]:=1
             else A[i, j]:=0;
          3: A[i, j]:=1;
        end;
      end;
  for i:=0 to MX-1 do   {меняем значения ячеек компонента LifeSGd}
    for j:=0 to MY-1 do

```

```
if A[i,j]=1 then LifeSGd.Cells[i,j]:='1'  
                                     {помечаем фишку как живую}  
else if (LifeSGd.Cells[i,j]='1')  
then LifeSGd.Cells[i,j]:='2'  
    {помечаем фишку как мертвую в течение одного поколения}  
else if (LifeSGd.Cells[i,j]='2')  
then LifeSGd.Cells[i,j]:='3'  
    {помечаем фишку как мертвую в течение двух поколений}  
else LifeSGd.Cells[i,j]:='0';  
    {помечаем фишку как окончательно мертвую}  
end;
```

**Эксперимент.** Мы завершили создание приложения. Сохраните проект.

Запустите приложение. Проследите за изменением популяции клеток.

Популяция непрестанно претерпевает необычные, нередко очень красивые и всегда неожиданные изменения. Иногда первоначальная колония организмов постепенно вымирает, т. е. все фишки исчезают, однако произойти это может не сразу, а лишь после того, как сменится очень много поколений. Однако в большинстве своем исходные конфигурации либо переходят в устойчивые (последние Конуэй называет «любителями спокойной жизни») и перестают изменяться, либо навсегда переходят в колебательный режим.

Придумайте колебательные и устойчивые конфигурации популяции. ♦

## Задания для самостоятельного выполнения

- 6.1. Приложение «Секундомер» (рис. 6.2.3). При нажатии на кнопку «Пуск» запускается секундомер (минуты : секунды : десятые доли секунды), а название кнопки изменяется на «Пауза». При нажатии на кнопку «Пауза» отсчет времени прекращается, а название кнопки изменяется на «Пуск». При нажатии на кнопку «Стоп» секундомер останавливается и происходит сброс времени.

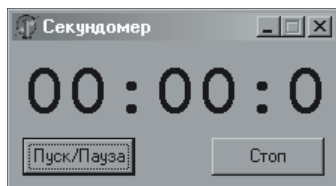


рис. 6.2.3

**6.2.** Напишите программу, которая с помощью компонентов Gauge (страница Samples) отображает количество часов, минут, секунд, прошедших с начала суток.

**Примечание.** Воспользуйтесь функциями Time (возвращает текущее время) и TimeToStr(Дата) (преобразует полученную дату в строковый формат).

**6.3.** «Часы шахматиста». Принцип работы: в начале шахматной партии часы установлены в нулевое положение. В начале игры часы первого шахматиста стартуют, по окончании хода он нажимает на кнопку, в результате чего его часы останавливаются и стартуют часы второго, после хода второго игрока его часы останавливаются, часы первого игрока продолжают отсчет от установленного ранее времени и т. д.

**6.4.** Приложение «Крестики-нолики». Дано: доска (поле) размером 3×3. Напоминаем суть игры. Партнеры по очереди ставят на поля квадрата (доски) крестики и нолики, выигрывает тот, кто первым выстроит три своих знака в ряд. Игра длится не более девяти ходов. Если никому из игроков не удастся добиться цели, партия заканчивается вничью. Напишите приложение, в которой партнерами выступают компьютер и человек.

**Примечание.** Для изображения крестиков и ноликов воспользуйтесь компонентом ImageList. Поместите в этот компонент изображения. Обработчик события OnDrawCell компонента StringGrid будет содержать следующий код:

```
Number := StrToInt(StringGrid1[ACol, ARow]);
      {В ячейках StringGrid помещены символы 0 (клетка свободна),
      1 (крестик), 2 (нолик)}
ImageList1.Draw(StringGrid1.Canvas, Rect.Left-2, Rect.Top-2,
Number);      {в ячейке StringGrid1 отображается изображение
      под номером Number}
```

**6.5.** Создайте приложение «Пятнадцать». Приложение произвольным образом расставляет фишки (1 ... 14) в коробке, оставляя свободной клетку в правом нижнем углу. В строке состояния должно отображаться количество сделанных ходов.

После завершения игры должно выводиться сообщение, поздравляющее с выигрышем.

Создайте форму, запрашивающую имя победителя. Сохраните имя победителя и его результат в файле. Предусмотрите просмотр победителей игры.

На форме должно отображаться количество минут и секунд, прошедших с начала игры.

**6.6.** Приложение «Перевертыши». Дано поле размером  $4 \times 4$  ( $6 \times 6$ ,  $8 \times 8$ ). В каждой клетке расположены фишки — синие и белые. Начальное расположение фишек генерируется случайным образом. За один ход можно перевернуть фишку в какой-либо произвольно выбранной ячейке, одновременно с ней переворачиваются фишки в соответствующих ячейках по вертикали и горизонтали. Цель игры: получить во всех ячейках фишки одного и того же цвета.

**6.7.** Приложение «Волки и овцы». В данную игру можно играть с кем-нибудь вдвоем или с самим собой. Один играет за овец, другой — за волков. У вас есть квадратная «поляна» размером  $4 \times 4$  (рис. 6.2.4). В ее углах стоят 2 овцы («О») и 2 волка («В»).

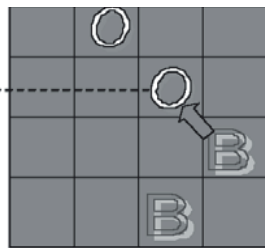
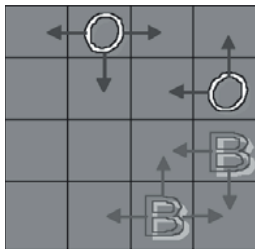
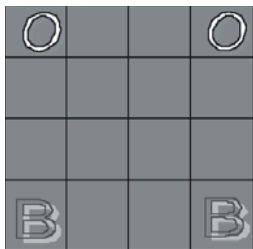


Рис. 6.2.4

Рис. 6.2.5

Рис. 6.2.6

Ходы осуществляются по очереди: волк, овца, волк и т. д. При этом каждый может передвигаться только на соседнюю клетку вперед, назад, влево и вправо (рис. 6.2.5).

Но если овца окажется на какой-нибудь соседней клетке по диагонали, то волк очередным ходом может съесть ее. В этом случае он становится на клетку, где была овца, а та уходит с поляны. Если зазевается волк, овца делает то же самое.

Выигрывает тот, кто останется на поляне.

**6.8.** Приложение «Ку-ну» (корейская игра). Играют два человека. На поле размером  $5 \times 5$ , клетки которого являются камушками, стоят два отряда воинов по 7 человек в каждой (рис. 6.2.7). Одни воины одеты в синее кимоно, другие — в зеленое. После того как брошен жребий, какому отряду начать игру, любой из воинов прыгает на соседний камушек, но только по диагонали. Затем точно так же прыгает воин из другого отряда. Остаться всему отряду на месте никак

нельзя, так как на скользких камнях трудно держать равновесие, поэтому если камушки впереди уже заняты, тогда придется прыгать назад — опять же по диагонали — на свободное место. Побеждает тот, кто быстрее успеет перейти на другой берег.

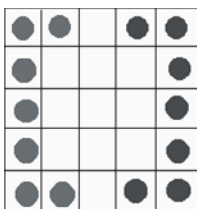


Рис. 6.2.7

**6.9. Приложение «Вечный Календарь».** Напишите приложение, визуальный интерфейс которого показан на рис. 6.2.8.

**Примечание.** Воспользуйтесь компонентом Calendar (страница Samples).

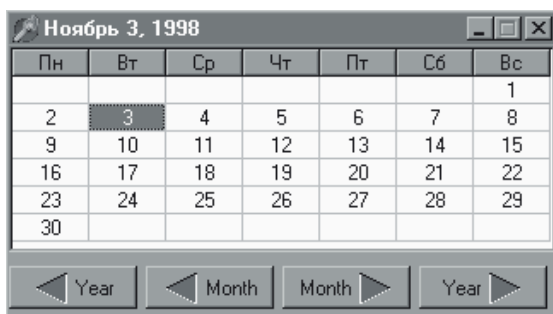


Рис. 6.2.8

При изменении выбранной даты отобразите соответствующие данные в заголовке формы.

**Примечание.** Для получения информации о компоненте воспользуйтесь файлом `c:\Program files\ Borland\ Delphi5\ Source\ Samples\ Calendar.pas`

**6.10. Модифицируйте приложение, описанное в задании 6.9.**

а) Известно, что астрологи делят год на 12 периодов и каждому из них ставят в соответствие один из знаков Зодиака:

20.01–18.02 Водолей	21.05–21.06 Близнецы	24.09–22.10 Весы
19.02–20.03 Рыбы	22.06–22.07 Рак	23.10–22.11 Скорпион
21.03–19.04 Овен	23.07–22.08 Лев	23.11–21.12 Стрелец
20.04–20.05 Телец	23.08–23.09 Дева	22.12–19.01 Козерог

Модифицируйте программу так, чтобы свойство Hint компонента Calendar отображало знак Зодиака для выбранной даты.

- b) В старояпонском календаре был принят 60-летний цикл, состоявший из пяти 12-летних подциклов. Подциклы обозначались названиями цвета: зеленый, красный, желтый, белый и черный. Внутри каждого подцикла годы носили названия животных: крысы, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки и свиньи (1984 год — год зеленой крысы — был началом очередного цикла). Модифицируйте программу так, чтобы свойство Hint формы отображало название выбранного года по старояпонскому календарю.

**6.11.** Приложение «Блоки». На поле размером  $6 \times 6$  находятся картинки (по две одинаковые), невидимые для игрока. Необходимо открыть все картинки. Картинки открываются парно, при этом, если открыты одинаковые картинки, они исчезают. Во время игры открытыми могут быть только две картинки: при открытии третьей картинки предыдущие две закрываются. Цель игры: открыть все картинки за наименьшее число попыток.

**6.12.** «Биологические ритмы». Биологические ритмы вычисляют, основываясь на гипотезе, что существует три цикла: физический (его период равен 23 дням), эмоциональный (период — 28 дней) и интеллектуальный (период — 33 дня). Кривые биологических ритмов могут быть представлены в виде синусоид. Начало всех трех кривых — день рождения. В первой половине каждого периода значения синусоиды положительны — это дни рабочего, приподнятого настроения, в дни второй части периода (когда значения синусоиды отрицательны) человек находится в пассивном, плохом настроении. В самом начале (после дня рождения) все биологические ритмы попадают в отрицательную часть периода. Составьте программу, которая запрашивает день рождения; число, на которое следует определить значения синусоид биологического ритма, и по этим данным строит синусоиды на 1 месяц.

**6.5. Приложение «Японский кроссворд».** В клетках японского кроссворда скрываются не слова, а картинки. Задача — нарисовать картинку по числам, которые проставлены слева от строк и над колонками. Числа разделены на группы, количество которых показывает, сколько групп закрашенных клеток находится в соответствующей линии а сами числа показывают, сколько слитных закрашенных клеток содержит каждая группа. Например, числа 2, 5 и 4 означают, что в этом ряду есть 3 группы, состоящие: первая — из 2, вторая — из 5, третья — из 4 закрашенных клеток. Группы разделены как минимум одной пустой клеткой. Пустые клетки могут быть и по краям рядов. Самое трудное — определить, сколько же пустых клеток находится между закрашенными группами.

Решение японского кроссворда разберем на простом примере. Если возле ряда стоит одно число, которое больше, чем половина длины ряда, то несколько клеток в середине ряда будут закрашены. Поэтому их можно смело пометать. В примере можно закрасить некоторые клетки в 1-й, 4-й и 5-й строках (рис. 6.2.9). Шестая строка содержит одну группу из 10 клеток, поэтому всю эту строку можно сразу закрасить.

			10				2	1		1	1		
8				1	3	5	7	3	3	6	5	5	6
		6											
2	1	1											
1	1	1											
		9											
		9											
		10											
	2	2											
	2	2											

Рис. 6.2.9

Теперь посмотрим на колонки. В первой всего одна клетка, ее положение уже известно, так что остальные клетки этой колонки явно пустые — отметим их серым цветом и не будем обращать на них внимание. Со второй колонкой поступим аналогично. Можно еще закрасить несколько клеток в четвертой колонке, так как группа содержит 7 клеток, а высота всей колонки 8, поэтому можем закрасить средние 6 клеток (рис. 6.2.10).

10							2	1		1	1	
8			1	3	5	7	3	3	6	5	5	6
		6										
2	1	1										
1	1	1										
		9										
		9										
		10										
	2	2										
	2	2										

Рис. 6.2.10

Продвигаясь шаг за шагом, получим вот такой автомобиль (рис. 6.2.11).

10							2	1		1	1	
8			1	3	5	7	3	3	6	5	5	6
		6										
2	1	1										
1	1	1										
		9										
		9										
		10										
	2	2										
	2	2										

Рис. 6.2.11

Напишите приложение, которое помогает разгадывать японский кроссворд, т. е. получает на входе файл с заданием японского кроссворда, а затем при выборе (отмене выбора) той или иной клетки отображает это в клетках справа и снизу.

### Вопросы для повторения

1. Для чего предназначен компонент StringGrid? В чем его отличие от компонента DrawGrid?
2. Назовите два класса-предка класса TStringGrid.
3. Компонент Timer. Расскажите о его свойствах и событиях.
4. Для чего используется компонент ImageList?

## Интерфейс Drag&Drop

Операционная система Windows широко использует специальный прием связывания программ с данными, который называется Drag&Drop (перетащи и отпусти). Такой прием в Проводнике Windows используется для копирования или перемещения файлов, а также для запуска обрабатывающей программы. Если, например, файл с расширением DOC перетащить на пиктограмму WinWord, автоматически запустится текстовый редактор Word for Windows и в окне появится текст из этого файла.

В Delphi реализован собственный интерфейс Drag&Drop, позволяющий компонентам обмениваться данными путем «перетаскивания» их мышью. Объекты можно перемещать в пределах формы и даже в другую прикладную программу.

В операции Drag&Drop участвуют 2 элемента:

- ☐ источник (или перемещаемый объект). Источником может быть элемент управления (кнопка, изображение, метка и т. д.) или выбранная часть какого-либо объекта (например, строка из TListBox);
- ☐ приемник (объект, на который будет опущен источник). Приемником может быть любой элемент управления.

В операции перетаскивания можно выделить четыре основных этапа.

1. Начало перетаскивания.
2. Проверка готовности приемника принять перетаскиваемый объект.
3. Сбрасывание перетаскиваемого объекта (источника).
4. Окончание процесса перетаскивания.

Для реализации данных этапов в Delphi существуют несколько свойств, событий и методов. Рассмотрим их по порядку.

### Свойства компонентов, участвующих в операции Drag&Drop

Свойство

**DragMode:** TDragMode;

где

`TDragMode = (dmManual, dmAutomatic);`

определяет, как будет выполняться весь комплекс действий, связанных с Drag&Drop.

Если DragMode принимает значение dmManual, то все события перетаскивания должны определяться вручную (т. е. программистом по ходу выполнения программы). Перетаскивание начинается только после вызова специальных методов.

Если значение DragMode равно dmAutomatic, то все события перетаскивания определяются автоматически, перетаскивание начинается сразу после нажатия кнопки мыши пользователем.

Свойство **DragCursor** определяет вид курсора в момент, когда над компонентом «перетаскиваются данные». Если компонент готов принять данные, то он присваивает этому свойству значение crDrag (курсор принимает вид прямоугольника со стрелкой), в противном случае — crNoDrag (курсор — перечеркнутый круг).

Изменение внешнего вида курсора осуществляется автоматически, если свойство DragMode имеет значение dmAutomatic.

### События компонентов, участвующих в операции Drag&Drop

В начале операции перетаскивания происходит событие OnStarDrag. Событие возникает у перетаскиваемого объекта. Событие не является обязательным для выполнения. Операция перетаскивания может быть произведена и без обработки этого события. Не все компоненты генерируют данное событие. Заголовок обработчика события имеет вид:

```
procedure TForm1.<имя_компонента>StartDrag(Sender: TObject;  
    var DragObject: TDragObject);
```

Параметр Sender содержит информацию о перетаскиваемом объекте. В параметре DragObject процедура получает информацию об объекте, создаваемом данным событием. Этот параметр используется для того, чтобы определить вид курсора или вид рисунка при перетаскивании объекта.

Проверка готовности приемника принять перетаскиваемый объект происходит при обработке события **OnDragOver**. Событие возникает в момент перемещения указателя мыши «с грузом» над компонентом. Заголовок обработчика этого события имеет вид:

```
procedure TForm1.<имя_компонента>DragOver  
    (Sender, Source: TObject; X, Y: Integer;  
    State: TDragState; var Accept: Boolean);
```

Параметр `Sender` указывает на компонент, над которым перемещается объект.

Параметр `Source` содержит информацию о компоненте — отправителе груза. В параметрах `X` и `Y` процедура получает координаты указателя мыши, выраженные в пикселях относительно компонента `Sender`.

Параметр `State` указывает состояние перемещаемого объекта относительно `Sender`. Тип `TDragState` описан так:

```
TDragState = (dsDragEnter, dsDragLeave, dsDragMove);
```

Значение `dsDragEnter` показывает, что `Source` только что появился над `Sender`. Если значение `State` стало равно `dsDragLeave`, то `Source` только что покинул `Sender` либо была отпущена кнопка мыши. Если `Source` перемещается над `Sender`, то значение `State` становится равным `dsDragMove`.

Параметр `Accept` сообщает, готов ли `Sender` принять перетаскиваемые данные. Если параметр имеет значение `True`, то `Sender` готов принять перетаскиваемый объект (если пользователь «сбросил» перетаскиваемый объект (отпустил кнопку мыши в данной точке), то приложение вызовет событие обработки операции по сбрасыванию объектов). Значение `False` этого параметра сообщает, что `Sender` не может принять перетаскиваемый объект (если пользователь отпустит кнопку мыши, то ничего не произойдет).

В обработчике этого события главное — определить значение параметра `Accept`.

Если в обработчике события `OnDragOver` значение параметра `Accept` было равно `True` и Вы попытались сбросить объект, то возникает событие **`OnDragDrop`** у компонента, на который объект был сброшен. Заголовок обработчика этого события имеет вид:

```
procedure TForm1.<имя_компонента>DragDrop  
(Sender, Source: TObject; X, Y: Integer);
```

Значения параметров этого обработчика события совпадают со значениями одноименных параметров обработчика события `OnDragOver`.

В обработчике события `OnDragDrop` необходимо выполнить все действия над перетаскиваемым объектом по «сбрасыванию».

При завершении перетаскивания (вне зависимости от того, приняты данные или нет), для перетаскиваемого объекта воз-

никает событие **OnDragEnd**. Оно происходит также при отмене перетаскивания. Событие не является обязательным для выполнения. Операция перетаскивания может быть произведена и без обработки этого события. Не все компоненты генерируют данное событие.

Заголовок обработчика события имеет вид:

```
procedure TForm1.<имя_компонента>EndDrag  
(Sender, Target: TObject; X, Y: Integer);
```

Параметр Sender получает информацию о перетаскиваемом объекте. Параметр Target содержит информацию об объекте, который получил данные. Если перетаскиваемый объект не был принят, то Target= Nil объект. X, Y — координаты указателя мыши в момент отпускания левой кнопки.

**Упражнение 7.1.** Пусть имеется 8 коробочек и 8 фигур. Фигуры отличаются друг от друга формой (круг или квадрат), цветом (белый или черный) и размером (большой или маленький). Каждая коробочка может содержать фигуру только определенного типа (форма, размер, цвет). Напишите приложения для размещения фигур в коробочки.

### Решение

Создайте новый проект. Сохраните новое приложение в папке Drag&Drop\_1 — файл модуля под именем Main.pas, файл проекта — DragDrop\_1.dpr.

#### *1-й этап. Визуальное проектирование*

Для реализации данной игры воспользуемся 8 компонентами типа TShape (это будут фигуры) и 8 компонентами типа TLabel (это коробочки).

Установите значения свойств формы следующим образом:

Name	Drag_Dropfrm
Caption	Drag&Drop

Для именования компонентов введем следующие обозначения: по размеру — B (big, большой) или S (small, маленький), по цвету — B (black, черный) или W (white, белый), по форме — S (square, квадрат) или C (circle, круг). У компонентов типа TShape добавим окончание shp. Таким образом, имя компонента, изображающего большой черный квадрат, будет BBSshp. А у белого маленького круга — SWCshp. Используя эти обозначения, измените свойства Name компонентов Shape.

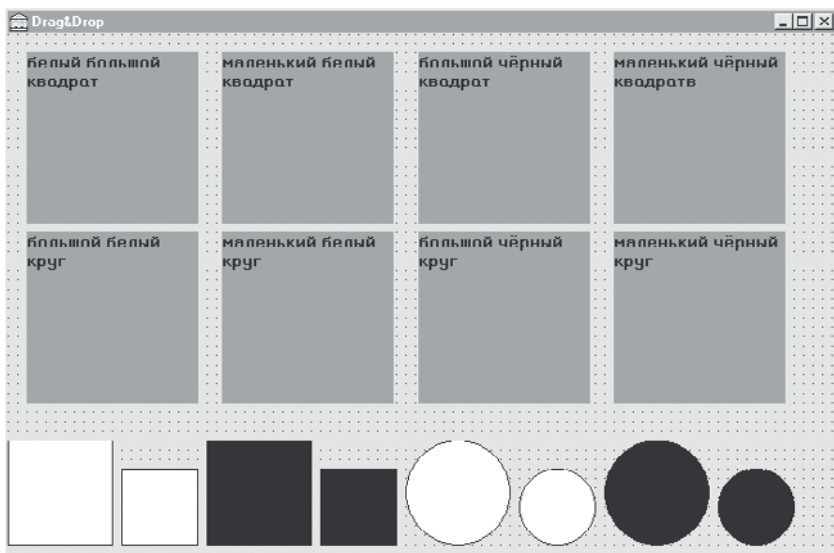


Рис. 7.1

Компоненты класса `TLabel` назовем по такой же схеме, изменим только окончание на `lbl`. То есть, если коробочка предназначена для большого черного круга, название будет `BBClbl`. Измените свойства `Name` компонентов `Label` в соответствии с введенной схемой обозначения.

У всех компонентов `Label` установите свойство `AutoSize` в значение `False`, свойство `Transparent` — свойство `True` (чтобы компоненты, помещенные на них, были видны), значения свойств `Caption` в соответствии с рис. 7.1.

Свойство `DragMode` компонентов `Shape` установите в значение `dmAutomatic`.

Для проверки правильности выбора коробочки (компонента типа `TLabel`) для компонента типа `TShape` воспользуемся свойством `Hint`. Значения свойства `Hint` задайте в соответствии с названием компонента, но без окончания (`shp` или `lbl`). Например, у компонента `SWSshp` оно будет равно `SWS`, причем оно будет совпадать со значением свойства для компонента `SWSlbl`.

### *2-й этап. Создание программного кода*

Создайте обработчик события `OnDragOver` для компонента `BWSlbl`. Чтобы определить, можно ли положить фигуру в коро-

бочку, необходимо сравнить значения свойства Hint перетаскиваемого объекта и объекта-приёмника. Если эти значения равны, то коробочка подходит, и соответственно Ассерт должно быть равно True, иначе объект сбросить нельзя и Ассерт принимает значение False:

```
procedure TDrag_Dropfrm.BWSlblDragOver(Sender, Source: TObject;  
    X, Y: Integer; State: TDragState; var Accept: Boolean);  
begin  
    if (Sender is TLabel) and (Source is TShape) then  
        Accept := ((Sender as TLabel).Hint = (Source as TShape).Hint);  
end;
```

Для остальных компонентов Label необходимо создать аналогичные обработчики события OnDragOver: в инспекторе объектов для всех компонентов Label в событии OnDragOver создайте ссылку на обработчик события TDrag\_Dropfrm.BWSlblDragOver.

**Примечание.** Если необходимо, чтобы при обработке события OnDragOver происходили какие-либо изменения с перетаскиваемым объектом, то желательно проверять его принадлежность к соответствующему классу (операция *is*), иначе могут возникнуть ошибки из-за несоответствия типов.

**Эксперимент.** Сохраните проект. Запустите проект и убедитесь, что при перетаскивании фигур только над коробочкой, соответствующей фигуре, курсор имеет форму стрелки с прямоугольником, а во всех остальных случаях — стрелки с перечеркнутым кругом. ♦

Если в обработчике события OnDragOver значение параметра Ассерт было равно True и Вы попытались сбросить объект, то возникает событие OnDragDrop у компонента, на который объект был сброшен.

Таким образом, если фигуру можно положить в коробочку и вы отпускаете клавишу мыши, то произойдет событие OnDragDrop. При этом необходимо изменить местоположение фигуры — она должна находиться в коробочке. Для этого нужно задать новые значения свойства Left и Top у перетаскиваемого компонента. Они будут зависеть от значений соответствующих свойств компонента-приемника.

```
procedure TDrag_Dropfrm.BWSlblDragDrop(Sender, Source: TObject;  
    X, Y: Integer);  
begin  
    if (Source is TShape) then
```

```

with (Source as TShape ) do
begin
  {определяем абсолютные координаты перетаскиваемого компонента}
  Left := (Sender as TLabel).Left + X;
  Top := (Sender as TLabel).Top + Y;
end;
end;

```

Для остальных компонентов, принадлежащих классу TLabel, установите ссылки на обработчик данного события.

*Эксперимент.* Сохраните проект. Запустите проект и убедитесь, что в результате операции Drag&Drop фигура «сбрасывается» внутрь корбочки (т. е. соответствующего компонента Label). ♦

После завершения перетаскивания (вне зависимости от того, приняты данные или нет) для перетаскиваемого объекта возникает событие OnDragEnd. При удачном перемещении фигуры будем очищать заголовок компонента-приемника:

```

procedure TDrag_Dropfrm.BWSshpEndDrag(Sender, Target: TObject;
  X, Y: Integer);
begin
  if Target <> nil then (Target as TLabel).Caption := ''
end;

```

Для всех остальных компонентов TShape сделайте ссылки на обработчик данного события.

*Эксперимент.* Сохраните проект. Убедитесь, что после «сбрасывания» фигуры надпись в соответствующем компоненте Label исчезает. ♦

### Методы компонентов, участвующих в операции Drag&Drop

Метод **BeginDrag** применяется для того, чтобы начать операцию перетаскивания. Метод понадобится в случае, когда свойство DragMode перетаскиваемого объекта установлено в значение dmManual. Чтобы перетаскивание началось, необходимо инициализировать метод BeginDrag у объекта, который надо перетащить. Удобнее всего это делать при обработке событий мыши данного объекта. Метод описан следующим образом:

```

procedure BeginDrag(Immediate: Boolean; Threshold: Integer=-1);

```

Параметр Immediate может принимать два значения. Если его значение равно True, то перетаскивание начинается немедленно. При значении, равном False, перетаскивание начинается при смещении курсора мыши в любом направлении на коли-

чество пикселей, определенное параметром `Threshold`. Как правило, удобнее присваивать `Immediate` значение `False`, так как в этом случае можно обрабатывать нажатие кнопки мыши, не начиная операцию перетаскивания.

После применения метода с объектами будут происходить все те же события, рассмотренные выше для значения свойства `DragMode`, равного `dmAutomatic`.

Обычно вызов метода `BeginDrag` осуществляется в обработчике события `OnMouseDown` перетаскиваемого объекта. Следует отметить, что при использовании данного метода пользователь сам должен позаботиться о проверке корректности начала операции перетаскивания, а именно, перетаскивание должно начинаться только при нажатии левой кнопки мыши (значение `Button` должно быть равно `mbLeft`). Следует также отфильтровать двойные щелчки, поскольку их использование приводит к странным побочным эффектам (значение `Shift` не должно содержать значения `ssDouble`, что показывает, что не было двойного щелчка).

**Примечание.** Для того чтобы начать процесс перетаскивания, можно просто в ходе программы в нужном месте присвоить свойству `DragMode` перетаскиваемого объекта значение `dmAutomatic`.

Метод **`EndDrag`** используется для того, чтобы остановить операцию перетаскивания, начатую вызовом метода `BeginDrag`.

**procedure** `EndDrag` (`Drop`: `Boolean`);

Параметр `Drop`, равный значению `True`, приводит к завершению операции перетаскивания и сбрасыванию объекта. Значение `False` отменяет процесс перетаскивания.

**Упражнение 7.2.** Решите задачу, сформулированную в упр. 7.1, установив значение свойства `DragMode` в `dmManual`.

### Решение

Откройте проект `DragDrop_1.dpr`.

У всех компонентов типа `TShape` установите свойство `DragMode` в значение `dmManual`.

**Эксперимент.** Запустите приложение. Осуществляется ли процесс перетаскивания компонентов? Объясните, почему это происходит. ♦

Создайте обработчик события `OnMouseDown` для компонента `BWSShp`, в котором при нажатии левой кнопкой мыши по выбранному компоненту будет начинаться процесс перетаскивания:

```

procedure TDrag_Dropfrm.BigSquareWShpMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  If (Button=mbLeft) and {нажата левая кнопка мыши}
    not(ssDouble in Shift) then {это не двойной щелчок мышью}
    (Sender as TShape).BeginDrag(False);
end;

```

**Эксперимент.** Сохраните проект. Запустите и убедитесь в том, что после смещения курсора мыши на несколько пикселей при нажатой левой кнопке мыши выполняется процесс Drag&Drop. ♦

Итак, программирование операции Drag&Drop заключается в выполнении следующих действий:

- 1) инициализация метода BeginDrag перетаскиваемого объекта (источника), если значение его свойства DragMode равно dmManual;
- 2) создание обработчика события OnDragOver компонента-приемника, чтобы определить, где можно «сбрасывать» перетаскиваемый объект;
- 3) создание обработчика события OnDragDrop компонента-приемника, чтобы определить, какие действия должны выполняться при «сбрасывании» перетаскиваемого объекта;
- 4) создание обработчика события OnDragEnd компонента-источника. Если два предыдущих шага необходимы для любой операции перетаскивания, то последний шаг выполняется лишь тогда, когда надо выполнить некоторые действия в исходном компоненте при завершении процесса перетаскивания.

## Задания для самостоятельного выполнения

- 7.1.** Имеется восемь кружков. Их начальное положение показано на рис. 7.2 (посередине — пустое поле). Цель игрока: поменять местами белые и черные фишки за наименьшее число ходов. Любую фишку можно перемещать только на пустое соседнее поле или же через одно занятое поле, опять-таки в пустое. Переводить фишки в другое место нельзя. На любом поле может находиться только одна фишка. Напишите приложение, реализующее эту игру.



Рис. 7.2

**7.2.** Дано квадратное поле размером  $N \times N$  ( $6 \leq N \leq 10$ ) клеток, на котором находятся 3 шара произвольного цвета. Количество цветов не превосходит  $N$ . Игрок может за один ход переносить один шар в любое свободное место поля. Если в результате хода появятся 5 шаров, выставленных в ряд по горизонтали, вертикали или одной из диагоналей, то они исчезают, и новых шаров не появляется. Игрок в этом случае получает 5 очков. В том случае, когда ряд не образуется, в свободных местах поля появляются три шара любого цвета. Игра заканчивается в том случае, когда на поле нет шаров или останется менее трех свободных клеток. Напишите программную реализацию игры.

**7.3.** Прямоугольное поле для игры полностью заполнено разноцветными шариками пяти цветов. За один ход игрок может менять местами любые два шарика. Если в результате хода образуется ряд по горизонтали, вертикали или одной из диагоналей, состоящий не менее чем из 3 шариков одного цвета, то шарики ряда исчезают и за каждый исчезнувший шарик игроку начисляется 5 баллов. Шарики, находящиеся сверху над исчезнувшими, сдвигаются вниз, заменяя исчезнувшие. Образовавшиеся пустые поля заполняются шариками произвольного цвета.

Если в результате хода не происходит построения ряда, то игра заканчивается. Цель игры набрать максимальное количество баллов. Напишите реализацию игры.

**7.4.** Напишите программу, реализующую процесс перетаскивания и удаления файлов, с изображением процесса перетаскивания и удаления.

**7.5.** Напишите программу, реализующую разложение карточного пасьянса (любого) с использованием механизма Drag&Drop.

**7.6.** «Ханойские башни». Имеются три колышка A, B, C и  $n$  дисков разного размера. Сначала все диски надеты на колышек A так, как показано на рис. 7.3. Цель игрока — перенести все диски с колышка A на колышек C, соблюдая при этом следующие условия: диски можно переносить только по одному, больший диск нельзя ставить на меньший. Напишите приложение, реализующее эту игру. Приложение должно подсчитывать количество перемещений, а также число минимально возможных перемещений, и показывать эти значения по окончании игры.

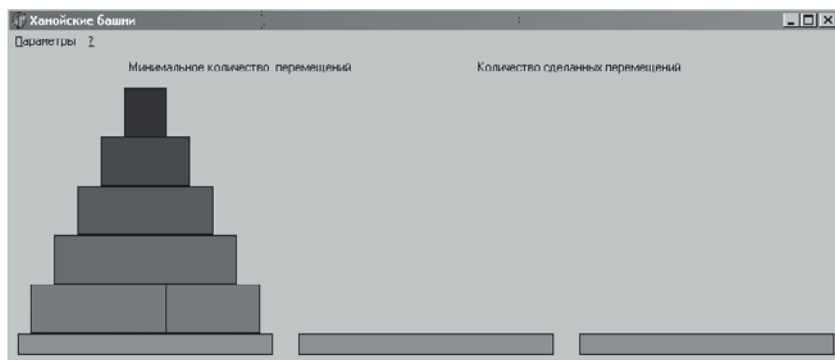


Рис. 7.3

### Вопросы для повторения

1. Что такое Drag&Drop?
2. Перечислите основные этапы процесса перетаскивания и события, им соответствующие.
3. Для чего нужно свойство DragMode?
4. Какой параметр события OnDragOver показывает, приняты перетаскиваемые данные или нет?
5. Когда применяется метод BeginDrag?

## Невизуальные компоненты Delphi

Невизуальные компоненты представляют собой, как правило, компоненты, с помощью которых осуществляется доступ к системным ресурсам. Они отображаются только во время конструирования интерфейса, но не видны во время работы приложения. Примером таких компонентов служит компонент `Timer`, который был использован при разработке игры «Жизнь». Кроме него к невизуальным компонентам относят диалоговые компоненты и компоненты-меню.

### 8.1. Диалоговые компоненты

В Win32 диалоговым окном называется окно стандартного размера без кнопок максимизации и минимизации. Диалоговые окна позволяют управлять различными режимами работы программы и сообщать ей необходимую информацию. Диалоговое окно может пересылать сообщения, задавать вопросы и принимать ответы.

Выделяют два вида диалоговых окон — модальные и немодальные. Модальные диалоговые окна сохраняют фокус ввода до закрытия окна. Немодальные диалоговые окна позволяют переключаться в другие окна без закрытия данного окна.

Компонент	Диалоговая панель
<code>OpenDialog</code>	выбор открываемого файла по шаблону
<code>SaveDialog</code>	создание файла
<code>FontDialog</code>	выбор шрифта и его характеристик
<code>ColorDialog</code>	выбор цвета
<code>PrintDialog</code>	вывод на устройство печати
<code>PrinterSetupDialog</code>	панель настройки устройства печати
<code>FindDialog</code>	панель поиска
<code>ReplaceDialog</code>	панель замены
<code>OpenPictureDialog</code>	выбор графического изображения с просмотром
<code>SavePictureDialog</code>	сохранение графического изображения с просмотром

Библиотека VCL Delphi содержит 10 компонентов, реализующих стандартные диалоговые панели, используемые многими Windows-приложениями. Эти компоненты расположены в панели компонентов на странице Dialogs. Компоненты и реализуемые ими стандартные диалоговые панели перечислены в таблице на предыдущей странице.

### 8.1.1. Основные правила использования диалоговых панелей

Работа со стандартными диалоговыми окнами осуществляется в три этапа.

Вначале на форму помещается соответствующий компонент и осуществляется настройка его свойств. Настройка свойств может проходить как на этапе конструирования, так и в ходе выполнения программы. Как и для любых других компонентов, программист не должен заботиться о вызове конструктора и деструктора диалогового окна — эти вызовы реализуются автоматически в момент старта и завершения программы.

На втором этапе осуществляется вызов стандартного для диалоговых компонентов метода `Execute`, который отображает соответствующее диалоговое окно. Вызов этого метода обычно располагается внутри обработчика какого-либо события. Например, для отображения стандартного диалогового окна сохранения файла в ответ на выбор пункта меню Файл/Сохранить может быть выполнен оператор:

```
If SaveDialog1.Execute Then ...
```

а обработчик нажатия кнопки Save может вызвать такой же метод у компонента `TSaveDialog` и т. д. Только после обращения к методу `Execute` на экране появляется соответствующее диалоговое окно. Стандартное окно диалога является модальным окном, поэтому сразу после обращения к `Execute` дальнейшее выполнение программы приостанавливается до тех пор, пока пользователь не закроет окно.

Метод `Execute` является логической функцией, которая возвращает значение `True`, если результат диалога с пользователем был успешным. Проанализировав результат выполнения метода `Execute`, программа может выполнить третий этап — использование введенных с помощью диалогового окна данных — имени файла, настроек принтера, выбранного шрифта и т. д.

**Упражнение 8.1.** Создайте приложение, которое использует компонент `ColorDialog` для изменения цвета формы.

**Решение**

Создайте каталог `Dialog_1`, файл модуля сохраните в файле `Main.pas`, файл проекта — `ChangeColor.dpr`.

Поместите на форму компоненты `ColorDialog` и `Button` (измените значения свойства `Caption` на «Изменить цвет», `Name` — «`ChangeBtn`»).

Создайте обработчик события `OnClick` кнопки:

```
procedure TForm1.ChangeBtnClick(Sender: TObject);
begin
  If ColorDialog1.Execute then
    {отображение стандартного диалогового окна выбора цвета}
    Color:=ColorDialog1.Color;
    {значение, возвращенное диалоговым окном, присваивается
                                         свойству Color формы}
end;
```

**Эксперимент.** Сохраните файлы проекта. Запустите приложение. Убедитесь, что компонент `ColorDialog` работает как стандартное диалоговое окно выбора цвета. Определите значение, возвращаемое методом `Execute` при выборе кнопок `OK` и `Cancel`. ♦

Аналогичным образом используются другие стандартные диалоговые компоненты. Заметим, что для всех компонентов, кроме `PrinterSetupDialog`, метод `Execute` является функцией.

## 8.1.2. Компоненты `OpenDialog` и `SaveDialog`

Компонент `OpenDialog` позволяет выбрать открываемый файл по заданному шаблону.

Компонент `SaveDialog` используется для выбора имени файла, в котором будет сохраняться информация.

<code>FileName</code>	содержит маршрут поиска и выбранный файл при успешном завершении диалога;
<code>DefaultExt</code>	определяет расширение файла по умолчанию: если при задании имени файла пользователь не указал расширение, то к имени файла будет добавлена разделительная точка и значение этого свойства;

Filter	используется для выбора файлов, отображаемых в диалоговом окне. Для определения фильтра можно воспользоваться редактором свойства. Свойство можно устанавливать с помощью специального редактора или программно: <pre>OpenDialog1.Filter:='Текстовые файлы  *.txt  Файлы Паскаля *.pas;*.dpr';</pre> Символы «   » служат для разделения фильтров друг от друга, а также для разделения описания фильтруемых файлов от соответствующей маски выбора;
FilterIndex	указывает, какой из заданных шаблонов отображается в списке. По умолчанию значение этого свойства равно 1;
InitialDir	задает название каталога, содержимое которого будет отображаться при вызове диалогового окна. Если значение этого свойства не задано, то отображается содержимое текущего каталога;
Title	задает текст заголовка диалогового окна;
Options	позволяет определить настройки диалогового окна

Значение	Описание (при установленном значении True)
ofAllowMultiSelect	позволяет одновременно выбрать из списка более одного файла
ofCreatePrompt	при вводе имени несуществующего файла отображает окно, которое запрашивает подтверждение на создание этого файла
ofExtensionDifferent	расширение имени выбранного файла отлично от расширения, заданного свойством DefaultExt
ofFileMustExist	при вводе имени несуществующего файла выдается предупреждающее сообщение
ofHideReadOnly	переключатель Read Only (только для чтения) не отображается

Значение	Описание (при установленном значении True)
ofNoChangeDir	после закрытия диалогового окна изменения текущего каталога не происходит (выбранный пользователем каталог не сохраняется)
ofNoReadOnlyReturn	файлы с атрибутом «только для чтения» не отображаются
ofNoTestFileCreate	при создании файла приложение должно само отслеживать правильность создания файла
ofNoValidate	пользователь не информируется о вводе недопустимых в именах файлов символов
ofOverWritePrompt	попытка сохранения файла поверх уже существующего приводит к отображению предупреждающего сообщения
ofReadOnly	при начальном отображении диалогового окна установлен флажок Read Only (при отключенном значении ofHideReadOnly)
ofPathMustExist	пользователь может указывать только существующие каталоги
ofShareAware	ошибки доступа к файлу игнорируются
ofShowHelp	диалоговое окно содержит кнопку Help
ofOldStyleDialog	создает диалог в стиле Windows 3.x
ofNoNetWorkButton	запрещает вставку кнопки для создания сетевого диска (при включенном ofOldStyleDialog)
ofNoLongNames	запрещает использование длинных имен файлов

### 8.1.3. Компонент FontDialog

Компонент FontDialog используется для вызова стандартного диалогового окна выбора шрифтов и их характеристик.

Device позволяет указать тип устройства, для которого выбираются шрифты:  
 fdScreen — экран;  
 fdPrinter — принтер;  
 fdBoth — шрифты, поддерживаемые и экраном, и принтером;

MinFontSize, этими свойствами определяется диапазон  
 MaxFontSize возможных значений размеров шрифтов.  
 Значения этих свойств задаются в пунктах (1 пункт равен 1/72 дюйма, что приблизительно равно 0.36 мм). Если свойства содержат 0, то ограничения на размер шрифта отсутствуют;

Font содержит результат выбора шрифта;

Options задает ряд опций диалоговой панели выбора шрифтов:

Значение	Описание (при установленном значении True)
fdAnsiOnly	пользователь может выбирать только шрифты, в которых находится набор символов, поддерживаемых Windows
fdEffects	в диалоговом окне будут отображены группа кнопок Effects и список Color
fdFixedPitchOnly	в списке шрифтов приводятся только моноширинные шрифты
fdForceFontExist	при вводе имени несуществующего шрифта выдается предупреждающее сообщение
fdNoFaceSel	при начальном отображении диалогового окна ни один шрифт не выбран
fdNoOEMFonts	запрещает выбор шрифтов MS-DOS
fdNoVectorFonts	исключает векторные шрифты (шрифты для Windows 1.0; используются в плоттерах)
fdNoSimulations	отображаются только реальные шрифты, а не синтезированные графическим интерфейсом Windows
fdNoSizeSel	при начальном отображении диалогового окна ни один размер не выбран

Значение	Описание (при установленном значении True)
fdNoStyleSel	при начальном отображении диалогового окна ни один стиль шрифта не выбран
fdShowHelp	в диалоговом окне находится кнопка Help
fdTrueTypeOnly	в списке шрифтов отображаются только TrueType-шрифты
fdWysiwyg	в списке шрифтов отображаются только шрифты, доступные и для экрана, и для принтера
fdLimitSize	включает ограничения на размер шрифта, заданные свойствами MinFontSize и MaxFontSize
fdScalableOnly	включает только масштабируемые шрифты (векторные и TrueType)
fdApplyButton	диалоговое окно содержит кнопку Apply

Для того чтобы изменить шрифт компонента на значение, установленное в диалоговом окне, необходимо определить обработчик сообщения OnApply компонента FontDialog.

8.1.4. Компоненты PrintDialog и PrinterSetupDialog

Компоненты PrintDialog и PrinterSetupDialog предназначены для управления параметрами принтера и процессом печати.

8.1.5. Компонент ColorDialog

Компонент ColorDialog используется для вызова диалогового окна настройки цветов.

Color	содержит выбранный цвет;
CustomColors	содержит до 16 цветов, определенных пользователем. Каждая строка имеет такой формат: ColorX=НННННН, где X — буква от А до Р, определяющая номер цвета, НННННН — шестнадцатеричное представление цвета в формате RGB;
Options	задает значения опций, определяющих настройку окна:

Значение	Описание (при установленном значении True)
cdFullOpen	показывать с развернутым окном выбора цвета пользователя
cdPreventFullOpen	запретить показ окна выбора цвета пользователем
cdShowHelp	включить в окно кнопку Help
cdSolidColor	выбирать ближайший сплошной цвет
cdAnyColor	разрешить выбор несплошных цветов

### 8.1.6. Компоненты FindDialog и ReplaceDialog

Компонент FindDialog используется для отображения стандартного диалогового окна, предназначенного для ввода искомой информации.

FindText устанавливает образец для поиска;

Options задает значения опций, определяющих настройку окна:

Значение	Описание (при установленном значении True)
frDown	устанавливает направление поиска вперед по тексту. Это значение установлено по умолчанию
frFindNext	устанавливается в True, когда пользователь нажимает кнопку Найти далее, в False — при закрытии окна
frHideMatchCase	удаляет переключатель «С учетом регистра»
frHideWholeWord	удаляет переключатель «Только слово целиком»
frHideUpDown	удаляет кнопки выбора направления поиска
frMatchCase	устанавливает флажок в переключателе «С учетом регистра»
frDisableMatchCase	запрещает выбор переключателя «С учетом регистра»
frDisableUpDown	запрещает использование кнопок выбора направления поиска
frDisableWholeWord	Блокирует переключатель «Только слово целиком»

Значение	Описание (при установленном значении True)
frReplace	используется в компоненте TReplaceDialog, чтобы сообщить системе о необходимости замены текущего выбора
frReplaceAll	используется в компоненте TReplaceDialog, чтобы сообщить системе о необходимости замены всех вхождений образца поиска
frWholeWord	устанавливает флажок в переключателе «Только слово целиком»
frShowHelp	включает в окно кнопку Help

Диалоговое окно поиска является немодальным, т. е. реализована возможность просмотра найденного фрагмента и, при необходимости, продолжения поиска. С этой целью для компонента определено событие OnFind, которое возникает всякий раз, когда пользователь нажимает кнопку FindNext. Обработчик события должен содержать алгоритм поиска заданного образца в тексте и отображения его пользователю.

Компонент ReplaceDialog предназначен для ввода текста, который необходимо найти и заменить на указанный текст. Класс TReplaceDialog является прямым потомком класса TFindDialog и наследует от него большинство свойств.

Новое свойство ReplaceText определяет текст замены. Обработчик события OnReplace, которое происходит после нажатия кнопок Replace или ReplaceAll, должен содержать алгоритм замены найденного текста на текст, заданный значением свойства ReplaceText.

### 8.1.7. Компонент OpenPictureDialog

Компонент OpenPictureDialog является потомком класса TOpenDialog и предназначен для выбора графических изображений с возможностью их просмотра. Компонент OpenPictureDialog поддерживает графические изображения в следующих форматах:

BMP	Windows Bitmap
ICO	Icon
WMF	Windows Metafile
EMF	Enhanced Windows Metafile

**Примечание.** Форматы графических изображений определяются классом `TPicture`, используемым для просмотра изображений.

### 8.1.8. Компонент `SavePictureDialog`

Компонент `SavePictureDialog` является «потомком» класса `TSaveDialog` и предназначен для сохранения графических изображений с возможностью их предварительного просмотра в одном из форматов, поддерживаемых классом `TPicture`.

## 8.2. Компоненты-меню

Компоненты-меню `MainMenu` (главное меню) и `PopupMenu` (всплывающее меню) располагаются на странице `Standartd` палитры компонентов. Редакторы меню (рис. 8.2.1) позволяют создавать меню во время проектирования приложения и сразу же испытывать их по мере готовности. Для вызова редактора меню вызовите контекстное меню компонента.



Рис. 8.2.1

Для определения свойств пунктов меню используется Инспектор объектов. Кроме того, редактор меню обладает всплывающим меню, которое обеспечивает быстрый доступ к наиболее часто используемым командам и доступ к шаблонам меню (`Menu Template`). Для вызова контекстного меню нажмите на правую кнопку мыши в окне редактора меню или комбинацию клавиш `Alt+F10`, когда курсор расположен в этом окне.

Всякий раз, когда пользователь нажимает кнопку мыши на элементе меню либо использует короткие клавиши или клавиши быстрого доступа, связанные с этим элементом, происходит событие `OnClick`.

### Примечания

1. Разделительные линии используются для наглядного группирования родственных элементов меню, например элементы меню `Файл`, связанные с печатью (`Параметры страницы...`, `Предварительный просмотр`, `Печать`). Для добавления в меню разделительной линии используйте символ «-» (минус) в качестве значения свойства `Caption`.

2. Клавиша быстрого доступа — это подчеркнутая буква в элементе меню. Она может использоваться совместно с клавишей Alt для доступа прямо к этому элементу. Чтобы назначить клавишу быстрого доступа в свойстве Caption соответствующего пункта меню, поставьте символ «&» непосредственно перед буквой, которую хотите сделать клавишей быстрого доступа.

3. Горячая клавиша — это комбинация клавиш Ctrl+символ, которая дает непосредственный доступ к какой-либо функции меню, минуя само меню. Горячие клавиши обычно назначают часто используемым элементам меню. Горячая клавиша для элемента меню назначается через свойство ShortCut.

4. Ветви меню могут быть вложенными — прицеплены к элементам меню более высокого уровня в качестве подменю. Для того чтобы создать подменю, растущее из какого-то верхнего элемента, выделите этот элемент и нажмите комбинацию клавиш Ctrl+стрелка вправо.

**Упражнение 8.2.** Разработайте приложение «Текстовый редактор».

### Решение

Создайте новый проект. Создайте каталог TextEdit, сохраните файл модуля под именем Main.pas, файл проекта — TextEdit.dpr.

В Windows-приложениях встречаются два основных стиля интерфейса пользователя. Все разработанные ранее приложения относились к SDI (Single Document Interface — однодокументный интерфейс) приложениям. SDI-приложение состоит из одного главного окна и может использовать несколько дополнительных вторичных окон. SDI-приложения позволяют работать одновременно только с одним документом. Чтобы открыть другой документ, нужно закрыть текущий.

В MDI (Multiply Document Interface — многодокументный интерфейс) приложениях возможна работа одновременно с несколькими документами, при этом каждый документ отображается в собственном окне. Каждое дочернее окно по сути своей является главным, но оно содержится внутри родительского окна.

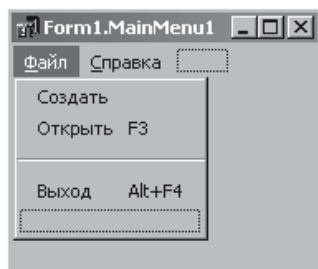
При разработке приложения «Текстовый редактор» используем многодокументный интерфейс.

#### *1-й этап. Визуальное проектирование*

Измените свойства главной формы следующим образом:

Свойство	Значение
Caption	Текстовый редактор
Name	MDIFile
FormStyle	fsMDIForm
Position	poDefault

Положите на форму компонент MainMenu. Используя редактор меню, разработайте меню, показанное на рис. 8.2.2.



**Рис. 8.2.2**

Измените значение свойства Name каждого элемента меню следующим образом:

Файл	MenuFileMIt
Создать	FileNewMIt
Открыть	FileOpenMIt
Выход	FileExitMIt
Справка	HelpMIt

В Delphi меню определяется минимальным числом элементов меню. При добавлении дочернего окна меню главного и дочернего окон объединяются, порядок расположения пунктов меню определяется значением свойства GroupIndex.

Установите значение свойства GroupIndex элемента меню Файл равным 1, а Справка — 10.

Сохраните файлы проекта.

Для работы с текстовыми документами добавьте в проект новую форму. Сохраните модуль формы под именем Child.pas.

Установите свойства дочерней формы следующим образом:

Caption	Новый файл
Ctrl3D	False
FormStyle	fsMDIChild
Name	MDIChild

Опустите на форму компонент RichEdit (страница Win32) и установите свойства следующим образом:

Align	alClient
BorderStyle	bsNone
ScrollBars	ssBoth

Сохраните файлы проекта.

**Эксперимент.** Запустите проект. Как отображаются формы приложения? Объясните, почему так происходит. ♦

После запуска приложения появляются одновременно оба окна — главное и дочернее. Причина, по которой отображается окно MDIChild, связана с тем, как Delphi генерирует код. Когда в проект добавляется новая форма, Delphi автоматически создаст глобальную переменную для новой формы в файле модуля, а к файлу проекта прибавляет строку, инициализирующую экземпляр соответствующего класса формы. Чтобы обойти это, удалите из файла проекта строку кода:

```
Application.CreateForm(TMDIChild, MDIChild);
```

**Эксперимент.** Запустите проект. Убедитесь в том, что при запуске приложения отображается только главная форма приложения. ♦

### *2-й этап. Создание программного кода*

Создадим обработчик события выбора элемента меню главной формы Файл/Выход:

```
procedure TMDIFile.FileExitMITClick(Sender: TObject);  
begin  
  Close;  
end;
```

**Эксперимент.** Запустите проект. Что происходит при выборе пункта меню Файл/Выход? Объясните, почему так происходит. ♦

Добавьте обработчик события выбора меню Файл/Создать:

```
procedure TMDIFile.FileNewMITClick(Sender: TObject);  
var MDIChild: TMDIChild;  
begin  
  MDIChild := TMDIChild.Create(Self);  
end;
```

**Эксперимент.** Сохраните и запустите проект. Создайте 5 файлов. Попробуйте выполнить различные манипуляции с окнами: переместить, изменить размеры, минимизировать и максимизировать, закрыть некоторые окна. Что происходит при выборе пункта меню Файл/Выход? Объясните, почему так происходит. ♦

При закрытии дочернего окна оно не закрывается, а минимизируется. Чтобы устранить этот недостаток, создайте обработчик события OnClose дочерней формы:

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

Вернемся к разработке визуального интерфейса дочернего окна: необходимо включить в меню главной формы опции меню дочернего окна. Положите на форму MDIChild компонент Main-Menu, создайте пункты меню дочернего окна и измените значения свойств следующим образом:

Элемент меню	Name	Group Index	Shortcut
&Файл	FileMenuMIIt	1	
Созд&ать	FileNewMIIt		
&Открыть	FileOpenMIIt		F3
&Закрыть	FileCloseMIIt		Alt+F3
(разделитель)			
&Сохранить	FileSaveMIIt		Ctrl+S
Со&хранить как...	FileSaveAsMIIt		
(разделитель)			
&Выход	FileExitMIIt		Alt+F4
&Формат	FormatMIIt	2	
&Окно	WindowMIIt	5	
&Упорядочить	WindowTileMIIt		
&Закрыть все	WindowCloseAllMIIt		

Элементы меню Формат и Окно дочерней формы, имеющие GroupIndex, равные 2 и 5 соответственно, помещаются между элементами Файл и Справка меню главного окна.

**Эксперимент.** Сохраните и запустите проект. Попробуйте создать несколько новых текстовых файлов.

Поскольку после создания нового файла меню главного окна приложения заменяется на меню дочерней формы, и, соответственно, события выбора пунктов меню адресуются не к окну главной формы, а к дочернему окну, то создание еще одного

файла становится невозможным. Чтобы исправить эту ошибку, создайте обработчик события `OnClick` пункта меню `Файл/Создать` дочернего окна:

```
procedure TMDIChild.FileNewMIClick(Sender: TObject);  
begin  
  MDIFile.FileNewClick(Sender);  
end;
```

Напишите аналогичный код для пункта меню `Файл/Выход`. Объясните различие между процедурами `MDIChild.FileCloseMIClick` и `MDIChild.FileExitMIClick`.

Запустите проект. Убедитесь в возможности создания более одного файла. ♦

Создадим обработчики выбора пунктов меню `Окно`.

```
procedure TMDIChild.WindowTileMIClick(Sender: TObject);  
begin  
  MDIFile.Tile;                                {упорядочить открытые окна}  
end;
```

```
procedure TMDIChild.WindowCloseAllMIClick(Sender: TObject);  
  var i: Integer;  
begin  
  with MDIFile do  
    for I:= 0 to MDIChildCount-1 do  
      {свойство MDIChildCount хранит значение количества дочерних окон}  
      MDIChildren[i].Close;  
      {массив MDIChildren содержит указатели на дочерние окна}  
end;
```

*Эксперимент.* Запустите проект. Убедитесь в работоспособности подпунктов меню `Окно`. ♦

Вернемся к разработке программного кода главной формы приложения. При обработке события выбора пункта меню `Файл/Открыть` необходимо знать имя открываемого файла — воспользуемся компонентом `OpenDialog`. Измените свойства компонента следующим образом:

Name	FileOpenDialog
Options	
OfFileMustExist	True
Filter	Определите самостоятельно, предоставив возможность открытия текстовых файлов (*.txt), файлов Паскаля (*.pas; *.dpr), всех файлов (*.*)

Создайте обработчик события выбора пункта меню Файл/Открыть:

```
procedure TMDIFile.FileOpenMitClick(Sender: TObject);
var MDIChild: TMDIChild;
begin
  if FileOpenDialog.Execute then
  begin
    MDIChild := TMDIChild.Create(Self);
    MDIChild.Open(FileOpenDialog.FileName);
    MDIChild.SetFocus;
    {устанавливаем фокус клавиатуры на это окно}
  end;
end;
```

Метод Open класса TMDIChild предназначен для открытия выбранного файла:

```
procedure TMDIChild.Open(const AFileName: string);
begin
  FileName := AFileName;
  RichEdit1.Lines.LoadFromFile(FileName);
  Caption := FileName;
end;
```

В описание класса TMDIChild добавьте переменную FileName (раздел private) и заголовок процедуры Open (раздел public).

**Эксперимент.** Запустите приложение. Откройте любой файл с расширением .txt. Попробуйте открыть еще один файл. Почему не выполняется команда меню Файл/Открыть? Объясните и устраните эту ошибку. ◆

## Задание для самостоятельного выполнения

8.1. Напишите программный код для пунктов меню Сохранить и Сохранить как...

### 8.2.1. Форматирование абзаца

Поместите на форму компонент Toolbar (страница Win32). Используя контекстное меню, поместите в него три новые кнопки (New Button) и разделитель (New Separator). Используя Image Editor, создайте для кнопок иконки «Выравнивание по правому краю», «Выравнивание по левому краю», «Выравнивание по центру» и отобразите их на кнопках панели инструментов.

Установите следующие значения свойств:

	ToolButton1	ToolButton2	ToolButton3
Name	LeftTlb	CenterTlb	RightTlb
Grouped	True	True	True
Tag	0	2	1
Style	TbsCheck	TbsCheck	TbsCheck

Свойство `Style` кнопок палитры инструментов определяет тип кнопки. Значение, равное `tbsButton`, определяет обычный вид кнопки, `tbsCheck` — в этом случае щелчок на кнопке приводит к изменению свойства `Down` (если кнопка выбрана, то она отображается вдавленной), `tbsDropDown` — кнопка отображается в виде раскрывающегося списка, `tbsSeparator` — создает расстояние между группами, `tbsDivider` — кнопка отображается как вертикальная разделительная линия.

Значение свойства `Grouped`, равное `true`, в последовательности расположенных рядом кнопок `TbsCheck` позволяет выделить не более одной кнопки одновременно.

Создайте обработчики событий нажатия на кнопки:

```
procedure TMDIChild.LeftTlbClick(Sender: TObject);
begin
  with Sender as TToolButton do
    RichEdit1.Paragraph.Alignment := TAlignment(Tag);
end;
```

Объектное свойство `Paragraph` класса `TRichEdit` содержит информацию о форматировании абзаца. `Alignment` (выравнивание по горизонтали) определяет внешний вид и ориентацию краев абзаца:

```
type TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

соответственно `TAlignment(0)` принимает значение `taLeftJustify` — выравнивание по левому краю.

**Эксперимент.** Запустите приложение. Убедитесь, что при выделении кнопок происходит соответствующее выравнивание границ абзаца. ◆

## Задания для самостоятельного выполнения

8.2. Дополните меню **Формат** командой **Абзац**. При выборе этой команды должно появляться диалоговое окно (рис. 8.2.3).

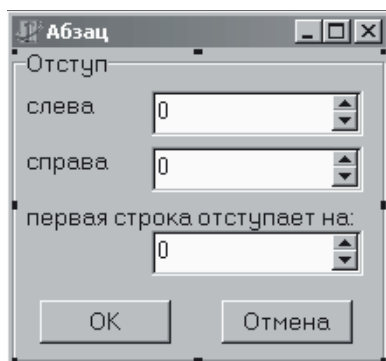


Рис. 8.2.3

При выборе кнопки ОК должны вноситься соответствующие изменения в форматирование текущего абзаца (изучите свойства, описанные в классе `TParaAttributes`).

- 8.3. Модифицируйте программу так, чтобы при перемещении от абзаца, выровненного, например, по центру, к абзацу, выровненному по левому краю, производились соответствующие изменения в панели инструментов (кнопка «Выравнивание по левому краю» становилась активной).

## 8.2.2. Форматирование текста

Поместите на панель инструментов еще три кнопки и разделитель для изменения начертания выделенных символов (курсив, полужирный, подчеркивание). Создайте иконки для этих кнопок.

В Object Pascal свойство `Font.Style` поддерживается следующими типами:

```
TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
TFontStyles = set of TFontStyle;
```

Установите значения свойств кнопок следующим образом:

	ToolButton1	ToolButton2	ToolButton3
Name	BoldTlb	ItalicTlb	UnderlineTlb
Tag	0	1	2
Style	TbsCheck	TbsCheck	TbsCheck

Напишите обработчики события нажатия на кнопки:

```
procedure TMDIChild.BoldTlbClick(Sender: TObject);
var Temp: TToolButton;
begin
  Temp := Sender as TToolButton;
                                {определяем, какая кнопка была нажата}
  with RichEdit1.SelAttributes do
    {с выделенным фрагментом текста выполняем следующие действия}
    if TFontStyle(Temp.Tag) in Style then
      begin
        {если стиль начертания был использован, то отменяем его}
        Style := Style-[TFontStyle(Temp.Tag)];
        Temp.Down := false
      end
    else
      begin
        {иначе применяем его}
        Style := Style+[TFontStyle(Temp.Tag)];
        Temp.Down := true
      end;
end;
```

*Эксперимент.* Запустите приложение. Убедитесь в правильности работы приложения. ♦

### Задания для самостоятельного выполнения

8.4. Модифицируйте программу так, чтобы при перемещении между словами, выделенными различными стилями начертания, соответствующие изменения происходили в панели инструментов.

8.5. Поместите на панель компоненты для отображения

- а) размера символов;
- б) шрифта символов (Times New Roman и др.);
- в) цвета символов.

Создайте обработчики событий выбора этих компонентов.

8.6. Добавьте в меню Формат команду Шрифт. Напишите обработчик события выбора этого пункта меню, используя компонент FontDialog.

### 8.2.3. Элементы меню Правка

Используя команду Insert From Template.../Edit Menu контекстного меню редактора меню, добавьте в меню дочернего окна

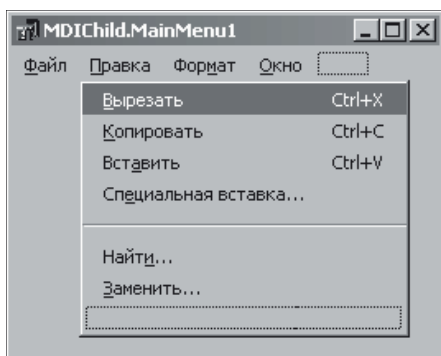


Рис. 8.2.4

элемент Правка. Измените свойство Caption подпунктов меню в соответствии с рис. 8.2.4 (лишние пункты меню удалите).

Создайте обработчик события подпункта меню Правка/Вырезать:

```
procedure TMDIChild.Copy1Click(Sender: TObject);
begin
    Richedit1.CutToClipboard;
    {заменяет содержимое буфера обмена на выделенный текст}
end;
```

**Эксперимент.** Запустите приложение. Убедитесь в правильности работы меню Правка/Вырезать. Создайте обработчики событий выбора пунктов меню Правка/Копировать и Правка/Вставить. ♦

Для реализации пункта меню Правка/Найти воспользуемся компонентом FindDialog:

```
procedure TMDIChild.Find1Click(Sender: TObject);
begin
    FindDialog1.Execute;
end;
```

После выбора кнопки Find Next у компонента FindDialog1 возникает событие OnFind:

```
procedure TMDIChild.FindDialog1Find(Sender: TObject);
var
    FoundAt, N_Line: LongInt;
    StartPos, ToEnd: integer;
begin
    with RichEdit1 do
```

```

begin
if SelLength <> 0 then
    {если искомый фрагмент найден, но поиск продолжается}
    StartPos := SelStart + SelLength
    {следующий фрагмент начинаем искать за найденным фрагментом}
else StartPos := 0;
                                {иначе поиск начинаем с начала текста}
ToEnd := Length(Text) - StartPos;
{определяем длину фрагмента, в котором будет осуществляться поиск}
FoundAt := FindText(FindDialog1.FindText, StartPos,
ToEnd,
    [stMatchCase]);
    {последний параметр метода TRichEdit.FindText определяет опции
        поиска, в данном случае при поиске будет учитываться регистр}
if FoundAt <> -1 then
begin
                                {если искомый фрагмент найден}
SetFocus;
                                {выделяем найденный фрагмент:}

SelStart := FoundAt;
SelLength := Length(FindDialog1.FindText);
    {определяем номер первой строки найденного фрагмента:}
N_line := Richedit1.Perform(EM_LINEFROMCHAR,
    RichEdit1.SelStart, 0);
                                {изменяем значение позиции полосы прокрутки,
                                    чтобы отобразить найденный текст:}
SendMessage(RichEdit1.Handle, EM_SCROLLCARET, N_Line, 0);
end;
end;
end;

```

**Эксперимент.** Запустите приложение. Убедитесь в правильности работы меню Правка/Найти.

Установите в диалоговом окне Find флажок Match whole word only (только слово целиком). Правильно ли выполняется поиск? Используя свойство Options компонента FindDialog1, внесите исправления в код метода TMDIChild.FindDialog1Find. ♦

## Задания для самостоятельного выполнения

8.7. Напишите обработчики событий выбора пунктов меню

- Правка/Заменить (воспользуйтесь компонентом ReplaceDialog);
- Правка/Специальная вставка... (воспользуйтесь проектом Fonts.dpr, разработанным в упр. 6.1.1).

- 8.8.** Реализуйте команды Найти и Заменить для объекта типа TStrinList.
- 8.9.** Создайте приложение «Музыкальная открытка». Приложение позволяет выбрать цвет формы, ввести текст поздравления, изменить шрифт текста, добавить рисунок и мелодию (воспользуйтесь компонентом MediaPlayer со страницы System). После щелчка на кнопке «Готово» остаются видимыми только текст поздравления и рисунок, звучит выбранная мелодия.
- 8.10.** Используя компонент MediaPlayer, создайте программу, которая с помощью компонентов DirectoryListBox, DriveComboBox, FileListBox, FilterComboBox позволяет составить список треков (они заносятся в компонент ListBox). Программа работает в фоновом режиме, проигрывая все файлы из списка по порядку. После того как проиграны все файлы, выводится сообщение и пользователю предлагается изменить список либо оставить его прежним. Для изменения списка добавьте на форму кнопки «Очистить все», «Удалить файл». Модифицируйте программу, сделав возможным сохранение списка. При запуске программы пользователь выбирает либо создание нового списка, либо использование сохраненного.
- 8.11.** Создайте приложение просмотра файлов, которое может работать с несколькими различными файлами.
- 8.12.** Создайте программу документирования проектов Delphi, которая печатала бы все файлы с расширением .pas из текущего каталога, а также файл проекта .dpr. Кроме того, следует предусмотреть возможность печати файлов пиктограмм и растровых изображений.

### Вопросы для повторения

1. Какие компоненты называются невизуальными?
2. В чем отличие модального диалогового окна от немодального? Приведите примеры.
3. Какие диалоговые компоненты входят в библиотеку визуальных компонентов? Что общего в их использовании?
4. Все приложения Windows подразделяются на обладающие однодокументным и многодокументным интерфейсом. В чем их отличие?

## Разработка компонентов в среде Delphi

Все компоненты Delphi являются частью иерархии, которая называется Visual Component Library (VCL). Общим предком всех компонентов является класс TComponent (рис. 9.1.1), в котором собран минимальный набор общих для всех компонентов Delphi свойств.

Свойство ComponentState содержит набор значений, указывающих на текущее состояние компонента. Приведем некоторые значения свойства:

table border="0">
csDesigning	компонент находится в режиме проектирования;
csDestroying	компонент сейчас будет разрушен;
csLoading	компонент загружается из файла формы;
csReading	компонент считывает значения из файла формы;
csWriting	компонент записывает значения своих свойств в поток;
csUpdating	компонент вносит изменения, чтобы отразить изменения в родительской форме.

Класс TComponent вводит концепцию принадлежности. Каждый компонент имеет свойство Owner (владелец), ссылающееся на другой компонент как на своего владельца. В свою очередь, компоненту могут принадлежать другие компоненты, ссылки на которые хранятся в свойстве Components. Конструктор компонента принимает один параметр, который используется для задания владельца компонента. Если передаваемый владелец существует, то новый компонент добавляется к списку Components владельца. Свойство Components обеспечивает автоматическое разрушение компонентов, принадлежащих владельцу. Свойство ComponentCount показывает количество принадлежащих компонентов, а ComponentIndex — номер компонента в массиве Components.

В классе TComponent определено большое количество методов. Наибольший интерес представляет метод Notification. Он вызывается всегда, когда компонент вставляется или удаляется из списка Components владельца. Владелец посылает уведомление каждому члену списка Components. Этот метод переопределяется в порожденных классах для того, чтобы обеспечить действительность ссылок компонента на другие компоненты. Например,

при удалении компонента Table1 с формы свойство DataSet компонента DataSource1, равное Table1, устанавливается в Nil.

Процесс разработки компонента включает пять этапов:

- выбор класса-предка;
- создание модуля компонента;
- добавление в новый компонент свойств, методов и событий;
- тестирование;
- регистрацию компонента в среде Delphi;

## 9.1. Выбор класса-предка

На рис. 9.1.1 изображены базовые классы, формирующие структуру VCL. В самом верху расположен TObject, который является предком для всех классов в Object Pascal. От него происходит TPersistent, обеспечивающий методы, необходимые для создания потоковых объектов. Потоковый объект — объект, который может запоминаться в потоке. Поток представляет собой объект, способный хранить двоичные данные (файлы). Поскольку Delphi реализует файлы форм, используя потоки, то TComponent порождается от TPersistent, предоставляя всем компонентам способность сохраняться в файле формы.

Класс TComponent представляет собой вершину иерархии компонентов и является первым из четырех базовых классов, используемых для создания новых компонентов. Прямые потомки TComponent — невизуальные компоненты.

### 9.1.1. Класс TControl

Вершину иерархии визуальных компонентов представляет класс TControl.

Класс TControl вводит понятие родительских элементов управления (parent control). Свойство Parent является окном, которое содержит элемент управления. Например, если компонент Panel1 содержит Button1, то свойство Parent компонента Button1 равно Panel1.

Свойство ControlStyle определяет различные стили, применимые только к визуальным компонентам, например:

csAcceptControls элемент управления становится родителем любых элементов управления, помещенных на него во время проектирования. Применим только к оконным элементам управления;

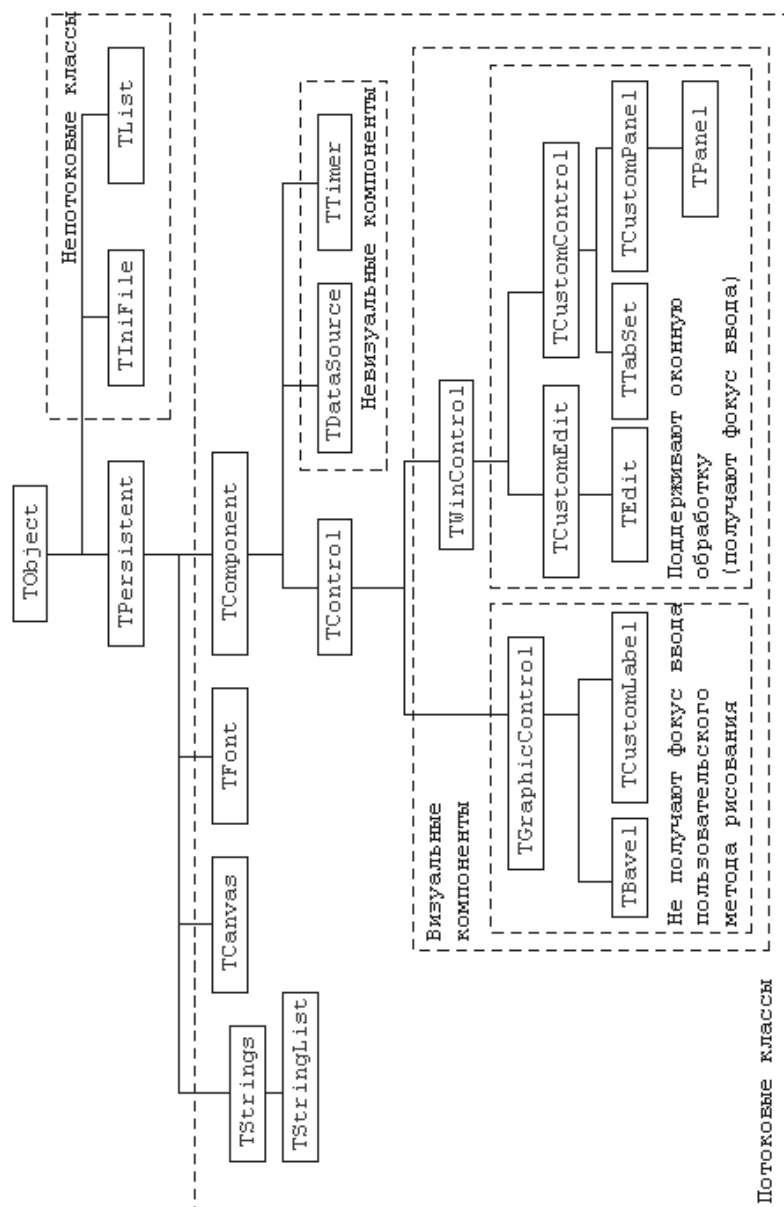


Рис. 9.1.1

<code>csCaptureMouse</code>	элемент управления перехватывает события мыши;
<code>csFrames</code>	элемент управления имеет рамку;
<code>csSetCaption</code>	свойства <code>Caption</code> и <code>Text</code> элемента управления (если не заданы явно) устанавливаются так, чтобы совпадать со свойством <code>Name</code> ;
<code>csOpaque</code>	элемент управления скрывает все элементы позади себя.

В классе `TControl` определено большинство свойств, используемых визуальными компонентами: свойства позиционирования (`Align`, `Left`, `Top`, `Height`, `Width`), свойства клиентской области (`ClientHeight`, `ClientWidth`), свойства внешнего вида (`Color`, `Enabled`, `Font`, `ShowHint`, `Visible`), строковые свойства (`Caption`, `Name`, `Text`, `Hint`), свойства мыши (`Cursor`, `DragCursor`, `DragKind`, `DragMode`).

Кроме того, класс `TControl` реализует методы диспетчеризации событий.

## Задание для самостоятельного выполнения

### 9.1. Используя справочную систему Delphi, познакомьтесь с методами класса `TControl`.

Все визуальные компоненты подразделяют на графические элементы управления и оконные элементы управления. Каждый тип представляет свою иерархию классов, происходящую соответственно от `TGraphicControl` и `TWinControl`. Главная разница между этими типами компонент состоит в том, что графические компоненты не поддерживают идентификатор окна, и, соответственно, не могут принять фокус ввода.

Оконные компоненты далее разбиваются на две категории. Прямые потомки `TWinControl` являются оболочками вокруг существующих элементов управления, реализованных в Windows (например, `TEdit`, `TButton`, и др.) и, следовательно, знают, как себя рисовать.

Для компонентов, которые требуют идентификатора окна, но не инкапсулируют базовых элементов Windows, которые бы обеспечивали возможность перерисовывать себя, имеется класс `TCustomControl`.

### 9.1.2. Класс `TGraphicControl`

Класс `TGraphicControl` является базовым для компонентов, которые не нуждаются в получении фокуса ввода и не служат в

качестве родительских для других элементов управления (эти функции требуют наличия идентификатора окна).

По умолчанию объекты TGraphicControl не имеют собственного визуального отображения, но для наследников обеспечиваются виртуальный метод Paint (вызывается всегда, когда элемент управления должен быть нарисован) и свойство Canvas (используется как «поверхность» для рисования).

### 9.1.3. Класс TWinControl

Класс TWinControl используется как базовый для создания компонентов, инкапсулирующих соответствующие оконные элементы управления Windows, которые сами себя рисуют.

Класс TWinControl обеспечивает свойство Handle, являющееся ссылкой на идентификатор окна базового элемента управления. Кроме этого свойства класс реализует свойства, методы и события, поддерживающие клавиатурные события и изменения фокуса:

свойства фокуса	TabStop, TabOrder;
свойства внешнего вида	Ctl3D, Showing;
методы фокуса	CanFocus, Focused;
методы выравнивания	AlignControl, EnableAlign, ReAlign;
оконные методы	CreateWnd, CreateParam, RecreateWnd, CreateWindowHandle, DestroyWnd;
события фокуса	OnEnter, OnExit;
события клавиатуры	OnKeyDown, OnKeyPress, OnKeyUp.

Создание любого потомка этого класса начинается с вызова метода CreateWnd, который вначале вызывает CreateParams для инициализации записи параметров создания окна, а затем вызывает CreateWindowHandle для создания реального идентификатора окна, использующего запись параметров. Затем CreateWnd настраивает размеры окна и устанавливает шрифт элемента управления.

### 9.1.4. Класс TCustomControl

Класс TCustomControl представляет собой комбинацию классов TWinControl и TGraphicControl. Являясь прямым потомком класса TWinControl, TCustomControl наследует способность управления идентификатором окна и всеми сопутствующими возможностями. Кроме этого, как и класс TGraphicControl, класс TCustomControl обеспечивает потомков виртуальным методом Paint, ассоциированным со свойством Canvas.

Таким образом, в зависимости от того, какой компонент будет исходным (базовым) для создания нового класса, можно выделить 4 случая:

- ☐ создание Windows-элемента управления (TWinControl);
- ☐ создание графического элемента управления (TGraphicControl);
- ☐ создание нового элемента управления (TCustomControl);
- ☐ создание невизуального компонента (TComponent).

## 9.2. Создание модуля компонента и тестового приложения

Определившись с выбором компонента, можно приступить к созданию модуля компонента. Для этого необходимо выполнить следующие шаги.

Выполните команду File/ New.../ Component или Component/ New Component.

В диалоговом окне New Component (рис. 9.2.1.) установите основные параметры создания компонента: Ancestor type (имя класса-предка), Class Name (имя класса компонента), Palette Page (вкладка палитры, на которой должен отображаться компонент) и Unit file name (имя модуля компонента).

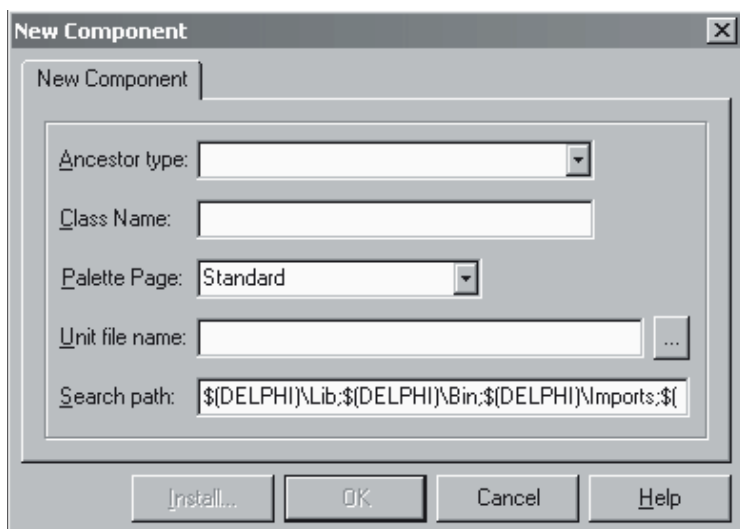


Рис. 9.2.1

После щелчка на кнопке ОК будет сгенерирован каркас нового класса.

По ходу процесса построения компонента необходимо тестировать его, не устанавливая в палитру компонентов. Тестовое приложение должно содержать код, который динамически помещает новый компонент на форму, изменяет его свойства и вызывает методы.

**Упражнение 9.2.1.** Разработайте новый компонент, который объединяет компоненты TEdit и TLabel. Компонент Label располагается выше поля редактирования (TEdit). При перемещении поля редактирования TLabel следует за ним. При удалении поля редактирования TLabel также удаляется.

### Решение

В качестве предка класса нового компонента используем TEdit.

Выполните команду Component/ New component.

Установите следующие значения параметров окна:

Ancestor type	TEdit
Class Name	TLabelEdit
Palette Page	Test
Unit file name	...\LabelEdit\LabelEdit.pas

Щелкните на кнопке ОК, автоматически будет сгенерирован следующий код:

```
unit LabelEdit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TLabelEdit = class(TEdit)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;
```

```
implementation
```

```
procedure Register;
```

```
begin
```

```
  RegisterComponents('Test', [TLabelEdit]);
```

```
end;
```

```
end.
```

В модуле описан каркас нового класса и написана процедура регистрации компонента (Register), которая помещает его на страницу Test. Сохраните файл модуля компонента.

### ***Разработка тестового приложения***

Создайте новый проект. Сохраните его файлы в папке ...\\LabelEdit: файл модуля — под именем Main.pas, файл проекта — TestApplication.dpr.

Добавьте имя модуля разрабатываемого компонента в раздел Uses формы тестового приложения:

```
...
```

```
uses ..., TLabelEdit;
```

```
...
```

В общедоступный раздел класса TForm1 добавьте поле

```
le: TLabelEdit;
```

В обработчике события OnCreate формы динамически создайте новый компонент:

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
  le:=TLabelEdit.Create(Self);
```

```
  with le do
```

```
    begin
```

```
      parent:=form1;
```

```
      left:=10;
```

```
      top:=10;
```

```
    end;
```

```
end;
```

Сохраните файлы проекта.

***Эксперимент.*** Убедитесь, что при запуске в левом верхнем углу формы появляется окно редактирования. ♦

## **9.3. Добавление свойств, методов и событий**

Свойства, как и поля класса, являются атрибутом объекта. Но если поля являются простым хранилищем некоего значе-

ния, которое может быть прочитано и изменено, то со свойством связаны некоторые действия, осуществляемые при чтении и изменении его содержимого.

Добавление свойства происходит в три этапа.

1. Создание внутреннего поля класса для хранения значения свойства.
2. Описание и разработка методов доступа к значению свойства.
3. Описание свойства.

В классе TControl свойства Caption/Text, Parent и Hint определяются так:

```
TControl = class (TComponent)
private
...
FParent: TWinControl;           {внутреннее поле свойства Parent}
FText: PChar;                   {внутреннее поле свойства Text/Caption}
FHint: string;                 {внутреннее поле свойства Hint}
function GetText: Tcaption;      {метод чтения свойства Text/Caption}
function IsCaptionStored: Boolean;
function IsHintStored: Boolean;
procedure SetText(const Value: Tcaption); {метод записи свойства Text/Caption}
...
protected
...
procedure SetParent(AParent: TWinControl); virtual;
property Caption: Tcaption read GetText write SetText
stored IsCaptionStored;
property Text: Tcaption read GetText write SetText;
...
public
...
property Parent: TWinControl read FParent write SetParent;
...
published
...
property Hint: string read FHint write FHint stored
IsHintStored;
...
end;
```

Объявление свойства имеет следующий синтаксис:

```
property <имя свойства>: тип определителя;
```

При объявлении свойства используется зарезервированное слово `property`, после которого указываются четыре ключевых фрагмента информации. Первый — имя свойства, этот идентификатор используется для ссылок на значение свойства. Таким образом, свойства получают внешний вид полей данных.

Каждое объявление свойства должно определять тип свойства, для этого используется символ двоеточие после имени свойства.

Для указания метода, который будет использоваться для осуществления выборки значения свойства, используется директива `read`. Метод должен быть функцией, чей возвращаемый тип является тем же самым, что и тип свойства.

Однако вместо метода доступа для чтения можно указать внутреннее поле хранения данных, как, например, при описании свойств `Hint` и `Parent`. Подобная форма записи приводит к тому, что значение свойства извлекается прямо из внутреннего поля данных.

За спецификацией метода чтения следует определитель метода записи, директива `write` определяет, какой метод будет использоваться для присвоения свойству значения. Метод должен быть процедурой, имеющей единственный параметр, тип которого должен совпадать с типом свойства.

При обращении к значению свойства происходит перенаправление на соответствующий метод. Например, оператор `s:=Edit1.Text`; автоматически будет преобразован в оператор `s:=Edit1.GetText`; а оператор `Edit1.Text:='Test'` — в оператор `Edit1.Text('Test')`.

Описание свойства должно содержать определитель `read` или `write` или сразу оба. Если описание свойства включает в себя только определитель `read`, то оно является свойством только для чтения. В свою очередь, свойство, чье описание включает в себя только определитель `write`, является свойством только для записи. При присвоении свойству, определенному с директивой только для чтения, какого-либо значения или при использовании в выражении свойства с директивой только для записи всегда возникает ошибка.

В отличие от внутренних полей хранения данных свойства не могут быть переданы в процедуру (или функцию) в качестве параметра переменной (параметр `var`), это объясняется тем, что свойство не существует в памяти.

Когда программист использует Инспектор объектов для изменения свойств формы или свойств компонентов, то результирующие изменения заносятся в файл формы. Файлы форм представляют собой файлы ресурсов Windows, и когда приложение запускается,

то описание формы подгружается из этого файла. Для определения того, что должно сохраняться в файле формы, служат спецификаторы памяти — необязательные директивы `stored`, `default` и `nodefault`. Эти директивы влияют на информацию о типе во время выполнения, генерируемую для свойств `published`.

Директива `stored` управляет тем, будет или нет свойство действительно запоминаться в файле формы. За директивой `stored` должны следовать либо константы `True` или `False`, либо имя поля, имеющего тип `Boolean`, либо имя метода, у которого нет параметров, и возвращающего значение типа `Boolean`. Например,

```
property Hint: string read FHint write FHint stored
IsHintStored;
```

Если свойство не содержит директиву `stored`, то оно рассматривается как содержащее ее с параметром `True`.

Директивы `default` и `nodefault` управляют значениями свойства по умолчанию. За директивой `default` должна следовать константа того же типа, что и свойство, например:

```
property Tag: Longint read FTag write FTag default 0;
```

Чтобы перекрыть наследуемое значение `default` без указания нового значения, используется директива `nodefault`. Директивы `default` и `nodefault` работают только с порядковыми типами и множествами, нижняя и верхняя границы которых лежат в промежутке от 0 до 31. Если такое свойство описано без директив `default` и `nodefault`, то оно рассматривается как с директивой `nodefault`. Для вещественных типов, указателей и строк значение после директивы `default` может быть только 0, `NIL` и "" (пустая строка) соответственно.

Когда Delphi сохраняет компонент, то просматриваются спецификаторы памяти `published` свойств компонента. Если значение текущего свойства отличается от `default` значения (или директива `default` отсутствует) и параметр `stored` равен `True`, то значение свойства сохраняется, иначе свойство не сохраняется.

Спецификаторы памяти не поддерживаются свойствами-массивами, а директива `default` при описании свойства-массива имеет другое назначение.

### 9.3.1. Простые свойства

Простые свойства — это числовые, строковые и символьные свойства. Они могут непосредственно редактироваться в Инспекторе объектов и не требуют специальных методов доступа.

Рассмотрим создание простого свойства Color, описанного в классе TControl (модуль controls.pas):

#### Type

```
TControl = class (TComponent)
private
    ...
    FColor: TColor;
    function IsColorStored: Boolean;
    procedure SetColor(Value: TColor);
protected
    ...
    property Color: TColor read FColor write SetColor stored
IsColorStored default clWindow;
    ...
end;

function TControl.IsColorStored: Boolean;
begin
    Result := not ParentColor;
end;

procedure TControl.SetColor(Value: TColor);
begin
    if FColor <> Value then
    begin
        FColor := Value;
        FParentColor := False;
        Perform(CM_COLORCHANGED, 0, 0);
        {вызов Perform позволяет обойти очередь сообщений Windows
        и послать сообщение, в данном случае — изменить цвет,
        элементу управления}
    end;
end;
```

### 9.3.2. Свойства перечислимого типа

Определенные пользователем перечислимые и логические свойства можно редактировать в окне инспектора объектов, выбирая подходящее значение свойства в раскрывающемся списке.

Рассмотрим создание свойства перечислимого типа на примере компонента Shape (модуль extctrls.pas).

#### Type

```
TShapeType = (stRectangle, stSquare, stRoundRect,
    stRoundSquare, stEllipse, stCircle);
    {вначале необходимо определить новый тип —
    перечислить возможные значения}
```

**Type**

```

TShape = class (TGraphicControl)
private
    ...
    FShape: TShapeType;
procedure SetShape(Value: TShapeType);
published
    ...
property Shape: TShapeType read FShape write SetShape
    default stRectangle;
end;
procedure TShape.SetShape(Value: TShapeType);
begin
    if FShape <> Value then
    begin
        FShape := Value;
        Invalidate;           {гарантирует перерисовку компонента}
    end;
end;

```

**9.3.3. Свойства типа множества**

Свойство типа множества при редактировании в окне Инспектора объектов выглядит так же, как множество, определенное синтаксисом языка Pascal. Простейший способ его отредактировать — развернуть свойство в Инспекторе объектов, в результате каждый его элемент станет отдельным логическим значением.

При создании свойства типа множества нужно создать соответствующий тип, описать методы доступа, после чего описать само свойство. В модуле Controls.pas свойство Align описано следующим образом:

**Type**

```

TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);
TAlignSet = set of TAlign;
TControl = class (TComponent)
private
    FAlign: TAlign;
procedure SetAlign(Value: TAlign);
public
property Align: TAlign read FAlign write SetAlign default
    alNone;
end;

procedure TControl.SetAlign(Value: TAlign);
var OldAlign: TAlign;
begin

```

```

if FAlign <> Value then
begin
  OldAlign := FAlign;
  FAlign := Value;
  Anchors := AnchorAlign[Value];
  if not (csLoading in ComponentState) and
    (not (csDesigning in ComponentState) or (Parent <> NIL))
  then
    if ((OldAlign in [alTop, alBottom])=(Value in [alRight,
      alLeft])) and not (OldAlign in [alNone, alClient]) and
      not (Value in [alNone, alClient])
    then SetBounds(Left, Top, Height, Width)
      {изменение границ компонента}
    else AdjustSize; {устанавливает заданные размеры компонента}
  end;
  RequestAlign; {инструктирует «родителя» переставить компонент
    в соответствии со значением свойства Align }
end;

```

### 9.3.4. Свойство-объект

Свойства могут являться объектами или другими компонентами. Например, у компонента `Shape` есть свойства-объекты `Brush` и `Pen`. Когда свойство является объектом, то оно может быть развернуто в окне инспектора так, чтобы его собственные свойства также могли быть модифицированы. Свойства-объекты должны быть потомками класса `TPersistent`, чтобы их свойства, объявленные в разделе `published`, могли быть записаны в поток данных и отображены в инспекторе объектов.

Для определения объектного свойства компонента необходимо сначала определить объект, который будет использоваться в качестве типа свойства. В модуле `graphics.pas` описан класс `TBrush`:

```

TBrush = class (TGraphicsObject)
private
  procedure GetData(var BrushData: TBrushData);
  procedure SetData(const BrushData: TBrushData);
protected
  function GetBitmap: TBitmap;
  procedure SetBitmap(Value: TBitmap);
  function GetColor: TColor;
  procedure SetColor(Value: TColor);
  function GetHandle: HBrush;
  procedure SetHandle(Value: HBrush);
  function GetStyle: TBrushStyle;
  procedure SetStyle(Value: TBrushStyle);

```

```
public
  constructor Create;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
  property Bitmap: TBitmap read GetBitmap write SetBitmap;
  property Handle: HBrush read GetHandle write SetHandle;
published
  property Color: TColor read GetColor write SetColor
    default clWhite;
  property Style: TBrushStyle read GetStyle write SetStyle
    default bsSolid;
end;
```

Метод Assign предназначен для копирования значения свойств экземпляра TBrush:

```
procedure TBrush.Assign(Source: TPersistent);
begin
  if Source is TBrush then
  begin
    Lock; {блокирует использование объекта}
    try
      TBrush(Source).Lock;
    try
      BrushManager.AssignResource(Self, TBrush(Source).FResource);
    finally TBrush(Source).Unlock;
    end;
  finally Unlock; {завершает секцию кода, начатую методом Lock,
    снимая блокировку объекта}
  end;
  exit;
end;
inherited Assign(Source);
end;
```

Чтобы определить свойство-объект, нужно определить внутреннее поле. Так как свойство представляет объект, его нужно создать, а по завершении — уничтожить, поэтому в код включены конструктор Create и деструктор Destroy. Кроме того, объявлен метод доступа SetBrush, предназначенный для записи свойства Brush.

```
TShape = class(TGraphicControl)
private
  FBrush: TBrush;
  procedure SetBrush(Value: TBrush);
  ...
```

```
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
  property Brush: TBrush read FBrush write SetBrush;
  ...
end;

constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  FBrush := TBrush.Create;
  FBrush.OnChange := StyleChanged;
end;
destructor TShape.Destroy;
begin
  ...
  FBrush.Free;
  inherited Destroy;
end;

procedure TShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);
end;
```

### 9.3.5. Свойство-массив

Примерами свойств-массивов могут служить такие свойства, как `TMemo.Lines`, `TScreen.Fonts`, `TStringGrid.Cells`.

Особенности свойства-массива заключаются в следующем:

- свойства-массивы объявляются с помощью индексных параметров, цель которых — указать количество и тип индексов, которые будут использоваться свойством;
- спецификации методов чтения и записи должны ссылаться на методы доступа. Методом для определителя `read` должна быть функция, список параметров которой совпадает со списком параметров, описывающих индекс свойства, и возвращающей значение того же типа, что и свойство. В свою очередь, методом в определителе `write` должна быть процедура, список параметров которой совпадает со списком параметров, описывающих индекс свойства. Список параметров такой процедуры может содержать и дополнительные свойства.

```
TCanvas = class (TPersistent)
private
    FHandle: HDC;                                {ссылка на контекст устройства,
                                                используется для отображения графической информации}
    function GetPixel(X, Y: Integer): TColor;      {метод чтения}
    procedure SetPixel(X, Y: Integer; Value: TColor);
                                                {метод записи}

    ...
protected
    ...
public
    constructor Create;
    destructor Destroy; override;
    property Pixels[X, Y: Integer]: TColor read GetPixel
write SetPixel;
    ...
end;

constructor TCanvas.Create;
begin
    inherited Create;
    ...
    CanvasList.Add(Self);    {добавляет в список ссылки на объекты}
end;

destructor TCanvas.Destroy;
begin
    CanvasList.Remove(Self); {удаляет из списка ссылки на объекты}
    ...
    inherited Destroy;
end;

function TCanvas.GetPixel(X, Y: Integer): TColor;
begin
    RequiredState([csHandleValid]);
    GetPixel := Windows.GetPixel(FHandle, X, Y);
end;

procedure TCanvas.SetPixel(X, Y: Integer; Value: TColor);
begin
    Changing;
    RequiredState([csHandleValid, csPenValid]);
    Windows.SetPixel(FHandle, X, Y, ColorToRGB(Value));
    Changed;
end;
```

Доступ к такому свойству-массиву осуществляется следующим образом:

```
Canvas.Pixels[10, 20] := clRed;
```

что означает:

```
Canvas.SetPixel(10, 20, clRed);
```

За описанием свойства-массива может следовать директива `default`. В данном случае это будет означать, что это свойство становится свойством по умолчанию для данного класса. На-пример:

```
type
TStringArray = class
public
  property Strings[Index: Integer]: string ...; default;
  ...
end;
```

Если у класса есть свойство по умолчанию, то доступ к этому свойству может быть осуществлен оператором

```
<имя компонента>[Index],
```

который эквивалентен оператору

```
<имя компонента>.<имя свойства>[Index].
```

Класс может иметь только одно свойство по умолчанию. В связи с тем, что компилятор статически определяет свойство по умолчанию у объекта, то смена свойства по умолчанию или его скрытие в наследниках класса может привести к непредсказуемым последствиям.

### 9.3.6. Массив свойств

Определитель `Index` позволяет разным свойствам иметь один и тот же метод доступа. Его описание состоит из директивы `index` и последующей за ней константой целого типа в промежутке от `-2147483647` до `2147483647`. Если у свойства есть определитель `Index`, то определители `read` и `write` должны ссылаться на методы, а не на поля. Например:

```
type
TRectangle = class
private
  FCoordinates: array[0..3] of Longint;
  function GetCoordinate(Index: Integer): Longint;
  procedure SetCoordinate(Index: Integer; Value: Longint);
public
  property Left: Longint index 0 read GetCoordinate
```

```

    write SetCoordinate;
property Top: Longint index 1 read GetCoordinate
    write SetCoordinate;
property Right: Longint index 2 read GetCoordinate
    write SetCoordinate;
property Bottom: Longint index 3 read GetCoordinate
    write SetCoordinate;
...
end;

```

Обращение к свойству, определенному с директивой `index`, например,

```

Rectangle.Right := Rectangle.Left + 100;
                                {Rectangle: TRectangle}

```

автоматически преобразуется к вызову метода,

```

Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);

```

### 9.3.7. Перекрытие и переопределение свойств

Описание свойства без указания типа называется перекрытием свойства. Самый простой способ перекрытия состоит в использовании зарезервированного слова `property` и идентификатора — имени свойства. Данный способ используется для смены видимости свойства.

Перекрытия свойств могут содержать директивы `read`, `write`, `stored`, `default` и `nodefault`. Перекрытие может заменить существующие наследуемые определители доступа, добавить недостающие, увеличить видимость свойства, но оно не может удалить существующий определитель или уменьшить видимость свойства. Следующий пример демонстрирует использование перекрытия свойств:

```

type
  TAncestor = class
    ...
  protected
    property Size: Integer read FSize;
    property Text: String read GetText write SetText;
    property Color: TColor read FColor write SetColor stored
      False;
    ...
  end;
TDerived = class(TAncestor)
  ...

```

```

protected
property Size write SetSize;
published
property Text;
property Color stored True default clBlue;
...
end;

```

Перекрытие свойства `Size` добавляет определитель `write`, что позволяет редактировать свойство, а перекрытие свойств `Text` и `Color` меняет их видимость с `protected` на `published`. Перекрытие свойства `Color` указывает, что оно должно быть сохранено, если его значение отлично от `clBlue`.

Переопределение свойства включает указание его типа, оно скрывает наследуемое свойство. Это означает, что создается новое свойство с тем же именем, что и у предка. Любое описание свойства, которое содержит указание типа, должно быть завершенным и включать в себя как минимум один определитель доступа:

```

type
TAncestor = class
...
property Value: Integer read Method1 write Method2;
end;

TDescendant = class(TAncestor)
...
property Value: Integer read Method3 write Method4;
end;

```

### 9.3.8. Создание событий

Событие — это любое происшествие, вызванное вмешательством пользователя, операционной системы или логикой программы. Событие связано с некоторым программным кодом, отвечающим на это происшествие. Совокупность события и кода, выполняющегося в ответ на это событие, называется свойством-событием и реализуется в виде указателя на некоторый метод. Метод, на который указывает это свойство, называется обработчиком события.

Свойства-события являются не более чем указателями на методы. В модуле `Controls.pas` определены стандартные свойства-события.

Описание свойства-события начинается с описания нового типа, который представляет собой процедуру, одним из пара-

метров которой, является Sender типа TObject, а директива of object делает эту процедуру методом:

```
TMouseMoveEvent = procedure (Sender: TObject; Shift: TShiftState;
  X, Y: Integer) of object;
```

Когда происходит какое-либо событие, например, перемещение мыши, в систему Win32 посылается соответствующее сообщение, в нашем случае WM\_MOUSEMOVE. Система Win32 передает это событие элементу управления, для которого оно предназначено и на которое он должен тем или иным способом ответить. Элемент управления может ответить на это событие, сначала проверив наличие кода, предусмотренного для выполнения. Для этого он проверяет, ссылается ли свойство-событие на какой-либо код. Если да, то элемент выполняет этот код, называемый обработчиком события. Операция по определению наличия метода, связанного с событием-свойством, возлагается на метод диспетчеризации. Эти методы объявляются как защищенные методы того компонента, которому они принадлежат.

Описание свойства-события состоит из двух частей: во-первых, событие требует внутреннего поля данных, которое используется для хранения указателя на метод; во-вторых, создается соответствующее свойство, которое во время проектирования дает возможность присоединения обработчиков событий:

```
TControl = class (TComponent)
private
  FOnMouseMove: TMouseMoveEvent;      {внутреннее поле события}
  procedure WMMouseMove (var Message: TWMMouseMove); message
  WM_MOUSEMOVE;
  ...
protected
  procedure MouseMove (Shift: TShiftState; X, Y: Integer);
  dynamic;                                {метод диспетчеризации}
  property OnMouseMove: TMouseMoveEvent read FOnMouseMove
  write FOnMouseMove;
  ...
end;
```

Метод диспетчеризации определяет, существует ли свойство-событие на какой-нибудь метод, и если это так, то передает управление соответствующей процедуре:

```
procedure TControl.MouseMove (Shift: TShiftState; X, Y: Integer);
begin
  if Assigned(FOnMouseMove) then FOnMouseMove (Self, Shift, X, Y);
end;
```

Чтобы обеспечить возможность переопределения обработки события, необходимо перехватить возникшее событие, обработать его стандартным образом и передать управление методу диспетчеризации:

```
procedure TControl.WMMouseMove(var Message: TWMMouseMove);
begin
  inherited;
  if not (csNoStdEvents in ControlStyle) then
    {включение csNoStdEvents во множество ControlStyle заставляет
      игнорировать стандартные события мыши, клавиатуры.
      Этот флаг позволяет ускорить запуск приложения,
      если оно при этом не нуждается в обработке этих событий}
  with Message do MouseMove(KeysToShiftState(Keys), XPos, YPos);
end;
```

### 9.3.9. Создание методов

Добавление в компонент методов не отличается от добавления методов в любой другой класс. Однако следует придерживаться следующих правил:

- исключить взаимозависимость методов;
- метод, вызываемый пользователем, не должен приводить компонент в такое состояние, при котором другие методы не действуют;
- метод должен иметь осмысленное имя.

**Упражнение 9.2.1 (продолжение).** Добавим в описание нового класса свойство объектного типа TLabel:

```
type
TLabelEdit = class (TEdit)
private
  { Private declarations }
  FLabel: TLabel; {внутреннее поле}
protected
  { Protected declarations }
  function GetLabelCaption : string; virtual;
    {метод чтения свойства Caption объектного свойства Label}
  procedure SetLabelCaption(Const Value: String); virtual;
    {метод записи свойства Caption объектного свойства Label}
public
  { Public declarations }
  constructor Create(Aowner: TComponent); override;
  destructor Destroy; override;
published
```

```
{ Published declarations }  
property LabelCaption: string read GetLabelCaption write  
SetLabelCaption;  
end;
```

В конструкторе необходимо создать экземпляр типа TLabel, сохранить ссылку на него во внутреннем поле и задать значение свойства Caption созданного объекта:

```
constructor TLabelEdit.Create(AOwner: TComponent);  
begin  
  inherited Create(Aowner);  
  FLabel := TLabel.Create(NIL);  
                                {владельца у свойства-объекта не существует}  
  FLabel.Caption := 'Label for Edit';  
end;
```

При разрушении компонента необходимо освободить ресурсы, занятые созданным объектным свойством:

```
destructor TLabelEdit.Destroy;  
begin  
  if (FLabel <> NIL) and (FLabel.parent = NIL) Then FLabel.free;  
  inherited Destroy;  
end;
```

Методы доступа чтения и записи соответственно считывают и записывают значение свойства Caption во внутреннее поле FLabel:

```
function TLabelEdit.GetLabelCaption: String;  
begin  
  Result := FLabel.Caption;  
end;  
  
procedure TLabelEdit.SetLabelCaption(Const Value: string);  
begin  
  Flabel.Caption := value;  
end;
```

*Эксперимент.* Сохраните модуль компонента. Запустите тестовое приложение. Как отображается создаваемый компонент? ♦

По-прежнему отображается только компонент Edit. Это связано с тем, что не определено свойство Parent внутреннего компонента Label. Напомним, свойство Parent задает компонент, который отвечает за прорисовку принадлежащих ему компонентов.

Добавьте в раздел `protected` описания класса `TLabelEdit` процедуру

```
procedure SetParent(value: TWinControl); override;
```

В разделе `implementation` модуля компонента опишите код процедуры:

```
procedure TLabelEdit.SetParent(Value: TWinControl);  
begin  
  if (Owner=NIL) or not (csDestroying in Owner.ComponentState)  
    {если владелец компонента не определен  
    или владелец не разрушается}  
  then FLabel.Parent:=Value;  
    {устанавливаем владельца компонента}  
  inherited SetParent(Value);  
end;
```

*Эксперимент.* Модифицируйте тестовое приложение в соответствии с рис. 9.3.1. Значения `Value` компонентов `SpinEdit` определяют положение компонента `LabelEdit`.

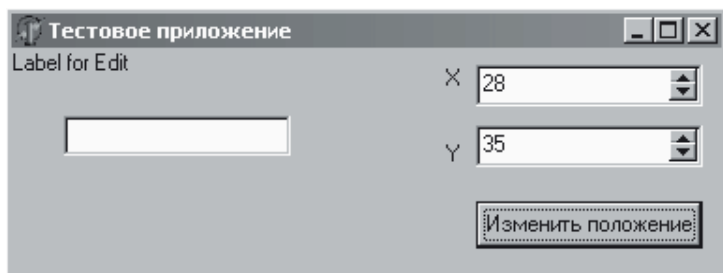


Рис. 9.3.1

Запустите тестовое приложение.

Проверьте отображение компонента `LabelEdit` при различных значениях свойств `Left` и `Top`. ♦

При перемещении компонента `Edit` надпись (`Label`) должна перемещаться за ним. Для этого необходимо перехватить событие перемещения — `WM_MOVE`. Опишите в разделе `private` описания компонента `TLabelEdit` заголовок обработчика события `WMMove`:

```
procedure WMMove(var Msg: TWMMove); message WM_MOVE;
```

Обработчик события `WMMove`, кроме стандартной обработки события, содержит операторы перемещения компонента `Label`:

```
procedure TLabelEdit.WMMove (var Msg: TWMMove);  
begin  
  inherited;  
  if Flabel <> NIL then  
    with Flabel do  
      SetBounds (Msg.XPos, Msg.Ypos-Height-5, Width, Height);  
      {procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer)  
      устанавливает сразу все граничные свойства элемента управления}  
end;
```

**Эксперимент.** Сохраните код модуля компонента. Запустите тестовое приложение, убедитесь в правильности перемещения компонента. ♦

## 9.4. Регистрация компонента в среде Delphi

В процессе регистрации компонент помещается в палитру компонентов Delphi.

Рассмотрим процесс установки компонента на примере компонента TLabelEdit, разработанного в упр. 9.2.1.

Выполните команду Component/Install Component. В диалоговом окне Install Component в строке Unit file name укажите имя модуля нового компонента — ...\\LabelEdit\\LabelEdit.pas (рис. 9.4.1). Щелкните на кнопке ОК.

Появится диалоговое окно Confirm с сообщением «Package dclusr50.bpl will be built then installed. Continue?» (Пакет dclusr50.bpl будет переустановлен. Продолжить?), щелкните на кнопке Yes.

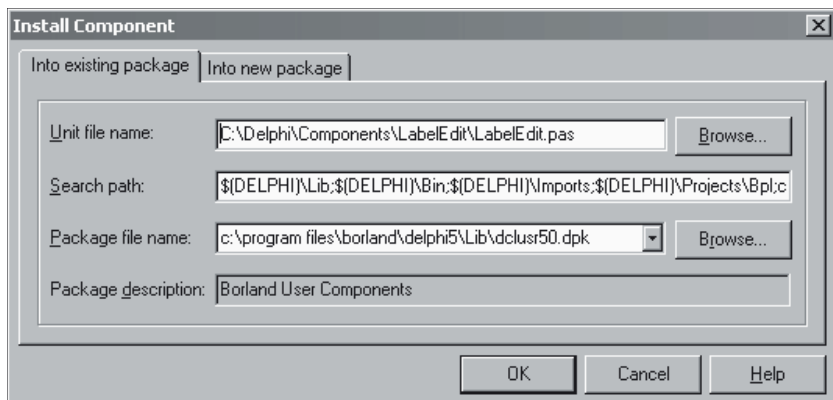


Рис. 9.4.1

Если нет ошибок в файле модуля нового компонента, то компонент будет зарегистрирован в палитре компонентов Delphi и будет отображено окно Information с сообщением «Package c:\program files\borland\delphis\Projects\Bpl\ dclusr50.bpl has been installed. The following new component(s) have been registered: TLabelEdit» (). Щелкните на кнопке OK.

В палитре компонентов на странице Test появится новый компонент (рис. 9.4.2).

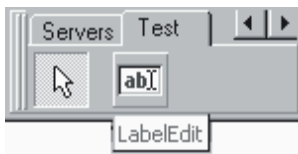


Рис. 9.4.2

**Примечание.** Чтобы изменить пиктограмму нового компонента, воспользуйтесь программой Image Editor. Выполните команду File/ New/ Component Resource File, затем Resource/ New/ Bitmap. В появившемся диалоговом окне Bitmap Properties установите размер рисунка 24×24 пикселя, установите цвет — VGA (16 color), нажмите OK и измените имя Bitmap1 на имя компонента (в нашем случае TLABELEDIT, вводите обязательно прописными символами). Затем выполните команду Resource/ Edit и создайте нужный рисунок. После этого сохраните файл в каталоге, в котором хранится модуль компонента, под тем же именем, но с расширением .DCR, установите компонент еще раз.

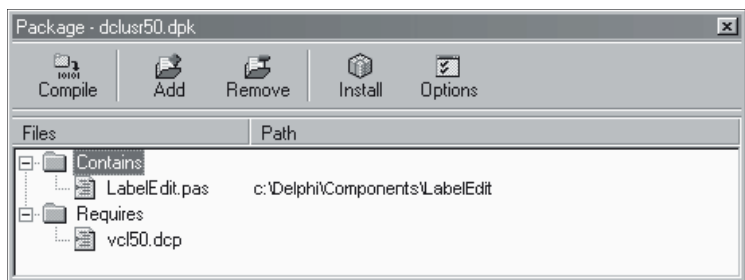
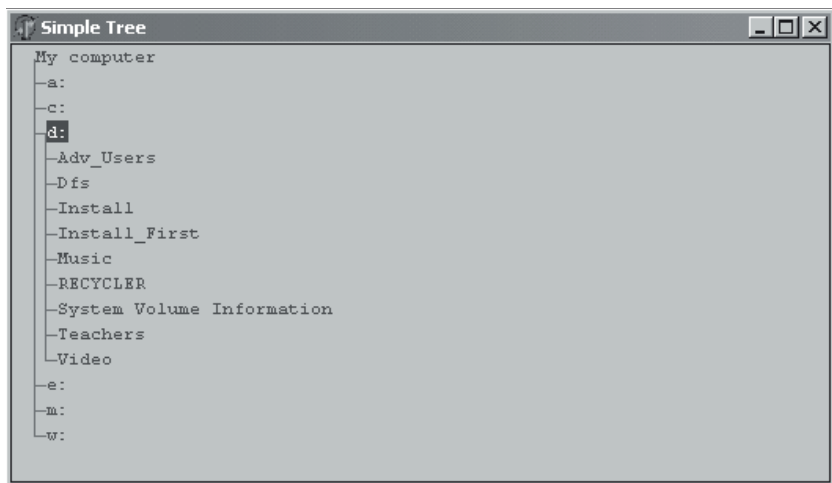


Рис. 9.4.3

В случае повторной установки компонента или же в случае наличия ошибок в модуле компонента будет отображен редактор пакета компонентов Package—dclusr50.dpk (рис. 9.4.3), который позволяет удалять, добавлять, компилировать пакет.

Для сохранения изменений, проведенных в пакете Dclusr50, щелкните на кнопке ОК в диалоговом окне Confirm «Save changes to project Dclusr50?» (Сохранить изменения в проекте Dclusr50?).

**Упражнение 9.4.1.** Разработайте компонент SimpleTree, отображающий структуру файловой системы в древовидной форме (рис. 9.4.4).



**Рис. 9.4.4**

### Решение

Создайте каталог SimpleTree.

Выполните команду File\New\Component. В диалоговом окне New Component установите основные параметры:

- ☐ введите имя класса предка — TCustomControl, так как этот класс предоставляет возможность рисования на компоненте и разрешает получать фокус ввода;
- ☐ имя создаваемого класса — TSimpleTree;
- ☐ название страницы палитры компонентов, на которую будет помещен компонент — Test;
- ☐ имя файла модуля, содержащего описания создаваемого класса, — ...\\SimpleTree\\SimleTree.pas;
- ☐ значение строки указания путей для поиска файла оставьте без изменения.

После щелчка на кнопке ОК откроется окно редактирования модуля ...\\SimpleTree\\SimleTree.pas, который содержит описание класса TSimpleTree и процедуру Register.

В разделе public описания класса опишите конструктор Create: **constructor** Create(AOwner: TComponent); **override**;

в котором определим возможность обрабатывать события мыши и установим объем рамки компонента:

```
constructor TSimpleTree.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := [csFramed, csCaptureMouse, csDoubleClicks,
    csClickEvents];
  {Свойство ControlStyle отвечает за
    различные атрибуты компонента:
    csFramed — элемент управления имеет рамку
    и нуждается в эффектах Ctrl3D;
    csCaptureMouse — данный элемент перехватывает события мыши;
    csDoubleClicks — когда на элементе дважды щелкнули мышью,
    генерируется событие OnDlClick;
    csClickEvents — когда на элементе нажата и отпущена мышь,
    генерируется событие OnClick}

  FBorder := bsSingle;
  Width := 150;
  Height := 150;
  TabStop := True;           {возможность перехода на компонент
                             при нажатии на клавишу Tab}
end;
```

## Задание для самостоятельного выполнения

### 9.2. Переопределите деструктор класса TSimpleTree.

Чтобы предоставить возможность пользователю компонента изменять внешний вид компонента и его положение на форме, создайте свойство Border и выполните перекрытие свойств Align, Anchors, Color, Ctl3D, Font, TabOrder, TabStop:

```
...
private
  FBorder: TBorderStyle
  ...
published
  property Align;
  property Anchors;
  property Border: TBorderStyle read FBorder write SetBorder
  default bsSingle;
```

```
property Color;  
property Ctl3D;  
property Font;  
property TabOrder;  
property TabStop;  
...  
end;  
...  
procedure TSimpleTree.SetBorder(const Value: TBorderStyle);  
begin  
  if FBorder = Value then begin  
    FBorder:=Value;  
    RecreateWnd;  
    {разрушает существующее окно, после чего создает заново}  
  end;  
end;
```

*Эксперимент.* Сохраните файл компонента.

Создайте тестовое приложение. Сохраните файл модуля под именем Main.pas, файл проекта Test.dpr.

Положите на форму компонент Button (измените свойство Caption на «Создать компонент»), создайте обработчик события onClick кнопки:

```
procedure TForm1.button1click(Sender: TObject);  
begin  
  Tree:=TSimpleTree.Create(Form1);  
  with Tree do begin  
    Parent:= form1;  
    Left:=5;  
    Top:=5;  
  end;  
end;
```

Опишите переменную Tree и подключите модуль SimpleTree.

Запустите тестовое приложение. После щелчка на кнопке на форме должен отобразиться экземпляр класса TSimpleTree. Закройте приложение, убедитесь, что при этом не происходит никаких ошибок. ♦

Добавим в компонент возможность вертикального скроллинга дерева (ScrollBar). Перекройте метод CreateParams, который вызывается перед созданием окна (перед вызовом функции WinAPI CreateWindow):

```
protected  
...
```

```

procedure CreateParams(var Params: TCreateParams); override;
...
procedure TSimpleTree.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  with Params do begin
    if FBorder = bsSingle then Style:=Style or WS_BORDER;
    Style:=Style or WS_VSCROLL;
  end;
end;

```

*Эксперимент.* Запустите тестовое приложение. Убедитесь в появлении вертикальной полосы прокрутки на создаваемом компоненте.

Используя справочную систему Delphi, определите, какие стили окон управления существуют и как каждый стиль влияет на функциональность окна управления. ♦

Перед отображением компонента вызывается метод Paint. Для рисования древовидной структуры файловой системы необходимо его переопределить:

```

protected
...
procedure Paint; override;
...
procedure TSimpleTree.Paint;
begin
end;

```

Рассмотрим процесс добавления большого количества элементов (узлов) в дерево. При добавлении каждого узла (до того времени, когда они отобразятся в компоненте) происходит перерисовка дерева, вызывающая мигание. Чтобы предотвратить этот эффект, создадим механизм блокировки «отрисовки» при добавлении узлов в дерево, который будет содержать два метода BeginPaint (начало блокировки) и EndPaint (окончание блокировки):

```

...
private
...
  FUpdateCount: integer;
  {в конструкторе задайте начальное значение равным нулю}
public
...
procedure BeginUpdate;

```

```
procedure EndUpdate;
...
procedure TSimpleTree.BeginUpdate;
begin
  inc(FUpdateCount);
end;
procedure TSimpleTree.EndUpdate;
begin
  Dec(FUpdateCount);
  if FUpdateCount = 0 then Invalidate;
end;
```

Для того чтобы не происходила перерисовка дерева в процессе добавления узлов, в метод Paint добавляем оператор:

```
if FUpdateCount > 0 then exit;
```

Для вычисления положения узла используем значение ширины и высоты символа «А». Введем два поля FCharWidth и FCharHeight, соответственно, длина и высота символа текста, обновление значений которых будет происходить в следующем методе:

```
procedure TSimpleTree.UpdateCharMetrics;
begin
  Canvas.Font := Self.Font;
  FCharHeight := Canvas.TextHeight('A') + 2;
  FCharWidth := Canvas.TextWidth('A');
end;
```

Метод UpdateCharMetrics будет вызываться в ответ на событие смены шрифта и размеров компонента:

```
private
...
procedure CMFontChanged(var Msg: TMessage);
  message CM_FONTCHANGED;
procedure WMSize(var Msg: TWMSize); message WM_SIZE;
...
procedure TSimpleTree.CMFontChanged(var Msg: TMessage);
begin
  inherited;
  UpdateCharMetrics;
end;

procedure TSimpleTree.WMSize(var Msg: TWMSize);
begin
  inherited;
  UpdateCharMetrics;
end;
```

Вернемся к процедуре `Paint`. Рисование дерева каталогов будем осуществлять последовательно: сначала отобразим узлы дерева, а затем, если нужно, — линии. Определите свойство `DrawLines` логического типа, значение `True` которого задает необходимость рисования линий дерева, `False` — рисование дерева без линий.

```
property DrawLines: boolean read FDrawLines write
SetDrawLines default True;
```

где

```
procedure TSimpleTree.SetDrawLines(const Value: boolean);
begin
  if FDrawLines <> Value then
    begin
      FDrawLines:=Value;
      Repaint;
    end;
end;
```

Не забудьте в конструкторе определить начальное значение поля `FDrawLines`. После этого метод `Paint` можно записать следующим образом:

```
procedure TSimpleTree.Paint;

procedure DoDrawNodes;
begin
end;

procedure DoDrawLines;
begin
end;

begin
  if FUpdateCount > 0 then exit;
  DoDrawNodes;                                     {рисует узлы}
  if FDrawLines then DoDrawLines;                 {рисует линии}
end;
```

Процедура `DoDrawNodes` рисует узлы дерева каталогов. Однако в конкретный момент времени нужно нарисовать только раскрытые пользователем узлы дерева. Список узлов дерева будем хранить в защищенном (`private`) поле `FDrawList` типа `TList` класса `TSimpleTree`.

## Задание для самостоятельного выполнения

**9.3.** В конструкторе класса `TSimpleTree` создайте `FDrawList` (экземпляра класса `TList`), а в деструкторе освободите память, ассоциированную с этой переменной.

Список `FDrawList` содержит указатели на узлы дерева. Каждый узел представляет собой экземпляр класса `TSimpleNode`:

```
TSimpleNode = class (TObject)
private
  FTree: TSimpleTree;           {указатель на дерево}
  FParent: TSimpleNode;         {родительский узел}
  FChildren: TList;             {список дочерних узлов}
  FCaption: string;            {текст для отображения}
  FLevel: integer;              {уровень узла}
  FIndex: integer;              {индекс в списке дочерних узлов родительского узла}
  FX, FY: integer;             {последние координаты, по которым рисовался узел}
  FExpanded: boolean;           {развернут ли}
  FAbsoluteIndex: integer;      {индекс узла в дереве}
  procedure Redraw; {перерисовка узла по последним координатам}
  procedure DrawAt (X, Y: integer);
                                {нарисовать узел по координатам X, Y}
  function GetChildren (Index: integer): TSimpleNode;
  function GetChildrenCount: integer;
  function GetSelected: boolean;
  procedure SetSelected (const Value: boolean);
  procedure SetCaption (const Value: string);
  procedure SetExpanded (const Value: boolean);
public
  constructor Create (ATree: TSimpleTree);
  destructor Destroy; override;
  procedure ClearChildren;           {очистить все дочерние узлы}
  property Children[Index: integer]: TSimpleNode read
    GetChildren;
  property ChildrenCount: integer read GetChildrenCount;
  property Caption: String read FCaption write SetCaption;
  property Level: integer read FLevel;
  property Selected: boolean read GetSelected write SetSelected;
                                {выбран ли узел}
  property AbsoluteIndex: integer read FAbsoluteIndex;
  property Index: integer read FIndex;
  property Expanded: boolean read FExpanded write SetExpanded;
end;
```

Обратите внимание на то, что описание класса `TSimpleTree` содержит элемент типа `TSimpleNode`, и наоборот. Чтобы сообщить компилятору о существовании класса `TSimpleTree` в разделе `Type`, опишите классы следующим образом:

**type**

```
TSimpleTree = class;
TSimpleNode = class (TObject)
...
end;
TSimpleTree = class (TCustomControl)
...
end;
```

Реализуем методы класса `TSimpleNode`. Конструктор инициализирует значения полей, а также выделяет память под переменную, которая будет содержать ссылки на подкаталоги:

**constructor** `TSimpleNode.Create (ATree: TSimpleTree);`  
**begin**

```
  inherited Create;
  FTree:=ATree;
  FParent:=nil;
  FChildren:=TList.Create;
  FLevel:=0;
  FIndex:=-1;
  FExpanded:=False;
```

**end**;

Деструктор освобождает память, ассоциированную с переменной `FChildren` (список ссылок на дочерние каталоги):

**destructor** `TSimpleNode.Destroy;`

**begin**

```
  ClearChildren;
  FChildren.Free;
  inherited Destroy;
```

**end**;

Удаление всех дочерних подкаталогов выполняет рекурсивная процедура, которая освобождает память, занятую под хранение ссылок на дочерние подкаталоги текущего подкаталога:

**procedure** `TSimpleNode.ClearChildren;`

```
  var i: Integer;
```

**begin**

```
  for i:=0 to FChildren.Count - 1 do
```

```
  begin
```

```
    Children[i].ClearChildren;
```

```
Children[i].Free;  
end;  
FChildren.Clear;  
end;
```

Метод **DrawAt** предназначен для вывода названия каталога (свойство **Caption** класса **TSimpleNode**): вычисляет координаты, определяет цвет надписи и цвет фона надписи:

```
procedure TSimpleNode.DrawAt(X, Y: Integer);  
begin  
  FY:=Y;  
  with FTree.Canvas do  
  begin  
    Brush.Color:=FTree.Color;  
    FX:=X + (Level + 1) * FTree.FCharWidth;  
    if Selected then  
    begin  
      Brush.Color:=FTree.SelBackColor;  
      Font.Color:=FTree.SelTextColor;  
    end  
    else Font.Color:=FTree.TextColor;  
    TextOut(FX + FTree.FCharWidth + 1, FY, Caption);  
  end;  
end;
```

Метод перерисовки названий подкаталогов:

```
procedure TSimpleNode.Redraw;  
begin  
  DrawAt(FX - (Level + 1) * FTree.FCharWidth, FY);  
end;
```

Обновление названия каталога:

```
procedure TSimpleNode.SetCaption(const Value: String);  
begin  
  if FCaption <> Value then  
  begin  
    FCaption:=Value;  
    FTree.Invalidate;  
  end;  
end;
```

## Задания для самостоятельного выполнения

9.4. Реализуйте методы **GetChildrenCount** (возвращает количество элементов, содержащихся в списке **FChildren**) и **GetChildren** (возвращает элемент списка **FChildren** под номером **Index**) класса **TSimpleNode**.

### 9.5. При отображении узлов дерева использовались следующие свойства класса TSimpleTree:

```

property TextColor: TColor index 0 read GetTreeColor
write SetTreeColor;
property LinesColor: TColor ...;
property SelTextColor: TColor ...;
property SelBackColor: TColor ...;

```

Используя массив свойств, реализуйте перечисленные выше свойства. Не забудьте в конструкторе Create установить начальные значения этих свойств.

К реализации методов GetSelected, SetSelected, SetExpanded вернемся немного позднее.

Таким образом, внутренняя процедура DoDrawNodes метода TSimpleTree.Paint, отображающая узлы дерева, должна выполнить такую последовательность операторов:

```

var i: Integer;
begin
  for i:=0 to FDrawList.Count - 1 do
    TSimpleNode(FDrawList[i]).DrawAt(0, i * FCharHeight);
end;

```

Метод NodeInView предназначен для проверки видимости узла (опишите в разделе private класса TSimpleTree):

```

function TSimpleTree.NodeInView(Node: TSimpleNode): Boolean;
begin
  Result:=FDrawList.IndexOf(Node) > -1;
end;

```

Для прорисовки дерева в методе Paint осуществляется рисование линий ) внутренняя процедура которого DoDrawLines метода Paint:

```

procedure DoDrawLines;
var MaxLevel: integer;
    i: integer;
    j: integer;
begin
  MaxLevel:=0;
  Canvas.Pen.Color:=LinesColor;
                                     {устанавливаем цвет рисования линий}
  for i:=0 to FDrawList.Count - 1 do
                                     {просматриваем все узлы дерева}
    with TSimpleNode(FDrawList[i]) do
      if FLevel > 0 then

```

```

begin
  Canvas.MoveTo(FX + FCharWidth, FY + FCharHeight div 2);
  Canvas.LineTo(FX, FY + FCharHeight div 2);
  if (FIndex > 0) and
    (not NodeInView(FParent.Children[FIndex - 1])) then
    Canvas.LineTo(FX, 0)
  else
    if FIndex=0 then Canvas.LineTo(FX, FY - FCharHeight div 2)
    else Canvas.LineTo(FX, FParent.Children[FIndex - 1].FY);
  if FIndex < FParent.ChildrenCount - 1 then
    if not NodeInView(FParent.Children[FIndex + 1]) then
      Canvas.LineTo(FX, ClientHeight);
    if MaxLevel < FLevel then MaxLevel:=FLevel;
  end;
for i:=1 to MaxLevel do
begin
  j:=0;
  while (j < FDrawList.Count) and
    (TSimpleNode(FDrawList[j]).Level <> i) do Inc(j);
  if j = FDrawList.Count then
    begin
      Canvas.MoveTo((i + 1) * FCharWidth, 0);
      Canvas.LineTo((i + 1) * FCharWidth, ClientHeight);
    end;
  end;
end;
end;

```

## Задание для самостоятельного выполнения

**9.6.** Поясните каждый оператор метода `TSimpleTree.DoDrawLines`. Приведите все возможные варианты выполнения метода `DoDrawLines`.

Сформируем список `FNodes` узлов. В описание класса `TSimpleTree` введем следующие элементы:

```

private
  FNodes: TList;           {глобальный массив всех узлов}
  FStartIndex: Integer;    {абсолютный индекс узла,
                           с которого начинаем рисовать. Начальное значение равно нулю}
  FMaxLinesInView: Integer;
                           {максимальное количество отображаемых узлов}
  FMaxLines: Integer;      {максимальное количество видимых узлов}

  function GetNode(Index: integer): TSimpleNode;
  function GetNodeCount: integer;
  ...

```

```

public
  property Nodes[Index: integer]: TSimpleNode
    read GetNode; default;      {возвращает узел под номером Index}
  property NodeCount: integer read GetNodeCount;
                                {общее количество узлов}
  ...
end;

```

### Задание для самостоятельного выполнения

**9.7.** Реализуйте методы `GetNode` и `GetNodeCount`. Не забудьте выделить память под переменную `FNodes` в конструкторе, а в деструкторе — освободить.

Формирование списка узлов дерева осуществляется в методе `UpdateDrawList`, который будет вызываться в ответ на каждое из следующих событий:

- ☐ изменение размеров компонента,
- ☐ добавление новых узлов,
- ☐ скроллинг,
- ☐ сворачивание или разворачивание какого-либо узла.

```

procedure TSimpleTree.UpdateDrawList;

function ListFull: Boolean;      {проверка на полноту списка}
begin
  Result:=FDrawList.Count >= FMaxLinesInView;
end;

procedure FormDrawList(Node: TSimpleNode);
                                {формирование списка}
  var i: Integer;
begin
  if not ListFull then FDrawList.Add(Node);
  Inc(FMaxLines);
  if Node.FExpanded then begin   {если узел раскрыт}
    for i:=0 to Node.ChildrenCount - 1 do
      FormDrawList(Node.Children[i]);
    Inc(FMaxLines, Node.ChildrenCount);
  end;
end;

var i, Min: Integer;
begin
  FMaxLinesInView:=(ClientHeight div FCharHeight) + 1;
  FDrawList.Clear;
  FMaxLines:=0;

```

```

if FStartIndex + FMaxLinesInView > GetNodeCount then
  Min:=GetNodeCount -FStartIndex
else Min:=FMaxLinesInView;
for i:=FStartIndex to FStartIndex + Min - 1 do
  FDrawList.Add(FNodes[i]);           {добавляем в список узлы}
for i:=0 to GetNodeCount - 1 do
  {вычисляем максимальное количество видимых узлов}
  with Nodes[i] do
    if FParent = nil then Inc(FMaxLines)
    else if FParent.FExpanded then Inc(FMaxLines);
  UpdateScrollBar;                     {обновляем состояние ScrollBar}
end;

```

Обновление состояния компонента ScrollBar осуществляет метод UpdateScrollBar:

```

procedure TSimpleTree.UpdateScrollBar;
var ScrollInfo: TScrollInfo;
    {структура, которая содержит параметры
    отображения полосы прокрутки}
begin
  if FMaxLinesInView >= FMaxLines then
    ShowScrollBar(Handle, SB_VERT, False)
    {спрятать вертикальную полосу прокрутки}
  else begin
    FillChar(ScrollInfo, SizeOf(TScrollInfo), 0);
    ScrollInfo.cbSize:=SizeOf(TScrollInfo);
    ScrollInfo.fMask:=SIF_ALL;    {ограничивает размер страницы
    пропорционально отображению полосы прокрутки,
    минимальное и максимальное значение для диапазона скроллинга }
    ScrollInfo.nMax:=FMaxLines;
    {максимальное количество отображаемых строк}
    ScrollInfo.nPage:=FMaxLinesInView; {общее количество строк}
    ScrollInfo.nPos:=FStartIndex;
    ShowScrollBar(Handle, SB_VERT, True);
    {показать вертикальную полосу прокрутки}
    SetScrollInfo(Handle, SB_VERT, ScrollInfo, True);
    {установить назначенные параметры}
  end;
end;

```

В раздел private класса TSimpleTree добавьте описание методов UpdateDrawList и UpdateScrollBar.

Метод UpdateScrollBar вызывается также и на изменение размеров компонента. Для полноценной поддержки скроллинга необходимо обрабатывать сообщение WM\_VSCROLL:

```

private
...
procedure WMVScroll(var Msg: TWMVScroll); message WM_VSCROLL;
procedure TSimpleTree.WMVScroll(var Msg: TWMVScroll);
begin
  case Msg.ScrollCode of
    SB_THUMBPOSITION: begin
      {прокручивает на абсолютную позицию.
      Текущая позиция определяется значением параметра pos}
      SetScrollPos(Handle, SB_VERT, Msg.Pos, True);
      FStartIndex:=Msg.Pos;
    end;
    SB_LINEUP: {вверх}
      if FStartIndex > 0 then Dec(FStartIndex) else exit;
    SB_LINEDOWN: {прокрутить на одну строку вниз}
      if FStartIndex < FMaxLines - FMaxLinesInView + 1 then
        Inc(FStartIndex)
      else exit;
    else exit;
  end;
  UpdateDrawList; {обновление списка}
  Invalidate; {перерисовка компонента}
end;

```

**Эксперимент.** Сохраните модуль компонента. Запустите тестовое приложение. Убедитесь, что при перемещении «бегунка» полосы прокрутки компонент ScrollBar исчезает. ♦

Добавим в обработчик события WMSize вызов методов обновления списка узлов и перерисовки полосы прокрутки:

```

procedure TSimpleTree.WMSize(var Msg: TWMSize);
begin
  inherited;
  UpdateCharMetrics;
  UpdateDrawList;
  UpdateScrollBar;
end;

```

**Эксперимент.** Запустите тестовое приложение. Отображается ли полоса прокрутки? Объясните, почему. ♦

Создаваемое дерево состоит как минимум из одного узла. Главный узел хранится в поле FMainNode типа TSimpleNode и доступен через свойство только для чтения MainNode.

Свойство SelectedNode типа TSimpleNode является указателем на выбранный узел, для записи в этот узел вызывается метод SetSelectedNode.

Опишите перечисленные свойства и методы класса TSimpleTree. Метод SetSelectedNode реализуется следующим образом:

```
procedure TSimpleTree.SetSelectedNode(const Value:
TSimpleNode);
var OldNode: TSimpleNode;
begin
  if FSelectedNode <> Value then begin
    {если выделенным должен стать другой узел}
    OldNode:=FSelectedNode;
    FSelectedNode := Value;
    if (OldNode <> nil) and NodeInView(OldNode) then
      {если узел, который был выделен ранее, виден —
      его следует перерисовать}
      OldNode.Redraw;
    if NodeInView(FSelectedNode) then
      {если выделенный сейчас узел видим, его также следует перерисовать}
      FSelectedNode.Redraw;
    end;
end;
```

В конструктор TSimpleTree.Create добавьте следующие операторы:

```
FNodes:=TList.Create;
FMainNode:=TSimpleNode.Create(Self);
FSelectedNode := FMainNode;
FNodes.Add(FMainNode);
FMainNode.FAbsoluteIndex := 0;
```

Чтобы обработать события мыши, выполним перекрытие метода MouseDown:

```
procedure TSimpleTree.MouseDown(Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
var Node: TSimpleNode;
begin
  inherited MouseDown(Button, Shift, X, Y);
  {вызываем обработчик события нажатия кнопок мыши по умолчанию}
  Node:=NodeAt[X, Y]; {определяем узел}
  if Node <> nil then SelectedNode:=Node;
  if (Shift = [ssLeft, ssDouble]) and (FSelectedNode <> nil) then
    if FSelectedNode.FExpanded {если выделенный узел раскрыт}
    then CollapseNode(FSelectedNode) {свернуть}
    else ExpandNode(FSelectedNode); {раскрыть}
end;
```

Метод `MouseDown` использует свойство-массив `NodeAt`:

**property** `NodeAt[X, Y: Integer]: TSimpleNode` **read** `GetNodeAt`;  
в котором метод доступа для чтения определим следующим образом:

```
function TSimpleTree.GetNodeAt(X, Y: Integer): TSimpleNode;
var i: Integer;
begin
  i:=0;
  Result:=nil;
  while (i < FDrawList.Count) and (Result = NIL) do
    with TSimpleNode(FDrawList[i]) do
      if PtInRect(Rect(FX, FY, FX + FCharWidth*(1 +
Length(Caption)), FY + FCharHeight), Point(X, Y))
        {функция определяет, принадлежит ли точка прямоугольнику}
      then Result:=FDrawList[i]
      else Inc(i);
end;
```

Метод, предназначенный для развертывания узла, можно описать так:

```
procedure TSimpleTree.ExpandNode(Node: TSimpleNode);
begin
  Node.FExpanded:=True;
  if NodeInView(Node) then begin                                {если узел виден}
    CheckforChildren(Node);                                       {проверка наличия внутренних
                                                                    узлов}
    UpdateDrawList;                                              {обновить список узлов}
    Invalidate;                                                  {перерисовать}
  end;
```

Аналогично реализуется метод свертывания узла:

```
procedure TSimpleTree.CollapseNode(Node: TSimpleNode);
begin
  Node.FExpanded:=False;
  Node.ClearChildren;
  UpdateNodeList;                                              {обновляем FNodes}
  if NodeInView(Node) then begin
    UpdateDrawList;
    Invalidate;
  end;
```

Для проверки наличия вложенных узлов используется метод `CheckforChildren`:

```
procedure TSimpleTree.CheckforChildren(ParentNode:
TSimpleNode);
begin
  BeginUpdate;
  if Assigned(FOnCheckforChildren) then
    FOnCheckforChildren(Self, ParentNode);
  EndUpdate;
end;
```

CheckForChildren является методом диспетчеризации события:

```
TOnCheckforChildren = procedure (Sender: TObject;
ParentNode: TSimpleNode) of object;
```

Новый тип свойства-события объявите до описания класса, внутреннее поле FOnCheckforChildren в разделе `private` класса TSimpleTree, виртуальный метод CheckForChildren в разделе `protected`, в разделе `Published` опишите новое свойство-событие OnCheckforChildren.

Обновление списка узлов происходит при вызове метода UpdateNodeList:

```
procedure TSimpleTree.UpdateNodeList;
procedure UpdateAbsoluteIndex(Node: TSimpleNode);
  var i: Integer;
begin
  Node.FAbsoluteIndex:=FNodes.Add(Node);
  for i:=0 to Node.ChildrenCount - 1 do
    UpdateAbsoluteIndex(Node.Children[i]);
  end;
begin
  FNodes.Clear;
  UpdateAbsoluteIndex(FMainNode);           {обновить номера узлов}
end;
```

При обработке события FOnCheckforChildren необходимо выполнить добавление узлов в список, это осуществляет метод AddChild:

```
function TSimpleTree.AddChild(Parent: TSimpleNode;
  const ACaption: String): TSimpleNode;

function Count(Node: TSimpleNode): Integer;
  var i: Integer;
begin
  Result:=1;
  for i:=0 to Node.ChildrenCount - 1 do
    Result:=Result + Count(Node.Children[i]);
```

```

end;

var i: Integer;
begin
  Result:=nil;
  if Parent = nil then exit;
  Result:=TSimpleNode.Create(Self);
  with Result do begin
    FParent:=Parent;
    FCaption:=ACaption;
    FLevel:=FParent.Level + 1;
    FIndex:=FParent.ChildrenCount;
    FAbsoluteIndex:=FParent.FAbsoluteIndex + Count(FParent);
  end;
  Parent.FChildren.Add(Result);
  FNodes.Insert(Result.FAbsoluteIndex, Result);
  for i:=Result.FAbsoluteIndex + 1 to FNodes.Count - 1 do
    TSimpleNode(FNodes[i]).FAbsoluteIndex:=i;
  if NodeInView(Result) then UpdateDrawList;
end;

```

Вернемся к реализации методов класса TSimpleMode. При описании свойства Selected были объявлены методы доступа SetSelected и GetSelected.

```

function TSimpleNode.GetSelected: boolean;
begin
  Result := (FTree.SelectedNode = Self);
  {FTree.SelectedNode содержит значение выделенного узла дерева;
   переменная Self содержит ссылку на узел дерева,
   относительно которого выполняется проверка}
end;

procedure TSimpleNode.SetSelected(const Value: Boolean);
begin
  if Selected then
    begin if not Value then FTree.SelectedNode:=NIL; end
  else if Value then FTree.SelectedNode:=Self;
end;

```

При выполнении щелчка по узлу дерева происходит его свертывание или разворачивание. Для установки соответствующего значения вызывается метод SetExpanded:

```

procedure TSimpleNode.SetExpanded(const Value: Boolean);
begin
  if Value <> FExpanded then
    if Value then FTree.ExpandNode(Self)
    else FTree.CollapseNode(Self);
end;

```

Для определения пути выбранной ветви в разделе **public** описания класса **TSimpleNode** опишите свойство **FullPath**:

```
property FullPath: string read GetFullPath;
```

Метод доступа для чтения свойства реализуется следующим образом:

```
function TSimpleNode.GetFullPath: string;  
begin  
  if FParent <> NIL then  
    Result:=FParent.GetFullPath + FTree.NodeSeparator + Caption  
  else Result:=Caption;  
end;
```

Итак, реализация методов классов на этом закончена. Сохраните модуль.

Внесем изменения в тестовое приложение. Используем разработанный компонент для отображения файловой системы.

Удалите компонент **Button1** и связанный с ним программный код. Внесите следующие изменения.

```
type  
  TForm1 = class (TForm)  
    procedure FormCreate(Sender: TObject);  
    ...  
  private  
    ...  
    procedure TreeCheckforChildren(Sender: TObject; ParentNode:  
      TSimpleNode);  
    end;  
    ...  
implementation  
  
const  
  sMyComputer = 'My computer';  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  Tree := TSimpleTree.Create(Self);  
  Tree.OnCheckforChildren := TreeCheckforChildren;  
  Tree.Color := clWindow;  
  Tree.Parent := Self;  
  Tree.Left := 10;  
  Tree.Font.Name := 'Courier New';  
  Tree.MainNode.Caption := sMyComputer;  
  Tree.Align := alClient;  
end;
```

```
procedure TForm1.TreeCheckforChildren(Sender: TObject;  
    ParentNode: TSimpleNode);  
  
procedure AddDrives;  
var i: Char;  
begin  
    for I := 'a' to 'z' do  
        if GetDriveType(PChar(String(i + ':\'))) > 1 then  
            Tree.AddChild(ParentNode, i + ':');  
end;  
  
procedure AddFiles;  
  
    procedure AddNode(FileName: String);  
    begin  
        if (FileName <> '.') and (FileName <> '..') then  
            Tree.AddChild(ParentNode, FileName);  
    end;  
  
var H: Integer;  
    R: TSearchRec;  
    S: String;  
begin  
    S:=ParentNode.FullPath;  
    Delete(S, 1, Length('My computer') + 1);  
    H:=FindFirst(S + '\*.*', faAnyFile, R);  
    if H = 0 then begin  
        AddNode(R.Name);  
        while FindNext(R) = 0 do AddNode(R.Name);  
    end;  
end;  
  
begin  
    if ParentNode = Tree.MainNode then AddDrives else AddFiles;  
end;
```

**Эксперимент.** Сохраните проект. Убедитесь в работоспособности компонента. ◆

**Примечание.** Программный код класса TSimpleTree приведен в приложении 2.

### Задания для самостоятельного выполнения

- 9.9. Модифицируйте компонент TLabelEdit, добавив свойство FontLabel, позволяющее изменять шрифт в компоненте Label.
- 9.10. На основе компонентов Label и Timer разработайте компонент TCreepingLine (бегущая строка).

- 9.11.** Разработайте компонент `TFloatSpinEdit`, позволяющий работать с вещественными числами.
- 9.12.** Создайте компонент `TEditAlign`, который позволяет выравнивать текст по правой части `Edit`.
- 9.13.** Разработайте компонент `TExpandedShape`, свойство `Shape` (форма) которого дополнено значением `Polygon` (многоугольник). При выборе этого значения `TShapeExpanded` отображается в виде правильного многоугольника.
- 9.14.** Модифицируйте компонент `TLabel`, дополнив его возможностью объемного (трехмерного) отображения. Определите логические свойства `recessed`, истинное значение которого отображает «утопленный» текст, и `shadow`, истинное значение которого изображает текст с тенью.
- 9.15.** На основе `TStringGrid` разработайте компонент, позволяющий
- ☐ в ячейке отображать многострочный текст;
  - ☐ выравнивать текст в ячейках;
  - ☐ вставлять и удалять строки;
  - ☐ раскрашивать ячейки в произвольный цвет.
- 9.16.** Создайте компонент `TDataAndTime`, который можно разместить на форме. Добавьте функцию будильника, который будет выдавать звуковой сигнал или выводить на экран сообщение. Для ввода даты и времени срабатывания будильника используйте поля данных списка строк.
- 9.17.** Создайте компонент `TFineRadioButton` с возможностью отображения определяемого пользователем символа (например, галочка) вместо обычной черной точки.
- 9.18.** Разработайте компонент `TColorButton` (цветная кнопка), позволяющий задавать цвет надписи.
- 9.19.** Создайте компонент мультипликационной пиктограммы, отображающий последовательность изображений.
- 9.20.** На основе класса `TImage` (`TPaintBox`) создайте компонент `TGraph` отображения графиков непрерывных функций. Для этого компонента определить объектное свойство `Diagram`, позволяющее задать вид графика, цвет его отображения

### Вопросы для повторения

1. Что такое компонент? Перечислите типы компонентов. Какие классы используются в качестве предков для каждого типа компонентов?
2. Опишите этапы построения компонента.
3. Какие виды свойств можно добавить в компонент? Опишите процесс создания нового свойства.
4. Что такое событие? В чем отличие создания свойства от события?

## Создание многопоточных приложений

### 10.1. Многозадачность и многопоточность

Сегодня, работая в современных операционных системах (ОС), примерами которых являются такие широко распространенные ОС, как Unix, Windows, OS/2 и др., мы имеем возможность выполнять одновременно несколько приложений. Например, при работе в Windows можно запустить программу поиска файла, а пока файл не найден, продолжить работу в текстовом редакторе MS Word. Таким образом, получается, что две программы — программа поиска файлов и текстовый редактор — выполняются одновременно.

Способность ОС поддерживать работу одновременно нескольких приложений называется многозадачностью. Система обеспечивает такую работу, выделяя каждому приложению определенную порцию квантов времени процессора, т. е. реально ОС не позволяет нескольким приложениям работать в один и тот же момент времени, а просто организует их последовательное выполнение в течение установленных интервалов времени. Существуют различные типы многозадачности. В Win32 используется **вытесняющая** многозадачность, подразумевающая, что управление между выполняющимися приложениями передается по истечении некоторого заранее определенного интервала времени по сигналу таймера. В Win16 использовалась **кооперативная** многозадачность, при которой передача управления выполнялась по инициативе самих приложений, что позволяло приложению «захватить» процессор, остановив выполнение других приложений.

Но каким же образом реализуется механизм одновременного выполнения сразу нескольких программ на уровне ОС. Как только Вы запускаете исполняемый файл на выполнение, в ОС стартует новый процесс. Например, если был запущен текстовый редактор MS Word, то в ОС создается новый процесс WINWORD.exe.

**Примечание.** Для того чтобы просмотреть список всех процессов, выполняющихся в данный момент на вашем компьютере в ОС Windows 2000, нажмите сочетание клавиш Ctrl+Alt+Del для вызова инспектора задач (Task Manager) и перейдите на вкладку Процессы (Processes).

Каждому процессу ОС выделяет 4 Гб виртуального адресного пространства. Однако не стоит путать виртуальное адресное пространство с действительной областью физической памяти компьютера. Виртуальное пространство памяти организуется за счет специально выделенной области жесткого диска, которая проецируется на реальную оперативную память постранично по мере надобности. Каждой программе назначается диапазон виртуальных адресов, но, как только работающей программе нужен доступ к памяти и программа обращается к некоторому виртуальному адресу, ОС переводит (назначает) этот виртуальный адрес в реальный физический адрес памяти.

Итак, процесс — это обозначение выполняющегося приложения (или отдельного экземпляра приложения). Процесс ничего не делает, он просто существует. Однако каждый процесс имеет первичный поток, в рамках которого и выполняется программный код, присутствующий в контексте данного потока. Процесс может иметь несколько потоков, однако только один из них является первичным, или главным. При создании процесса ОС автоматически создает его главный поток. В свою очередь, этот поток при необходимости может создавать дополнительные потоки.

Таким образом, когда мы говорим об одновременном выполнении нескольких приложений, речь идет об одновременном выполнении нескольких потоков. В обычных однопроцессорных системах в конкретный момент времени не может выполняться два или более потоков. На самом деле, в соответствии с вытесняющей многозадачностью, ОС выделяет для выполнения каждого потока определенный отрезок времени (квант), в течение которого и выполняется поток, а уже за счет быстрого действия компьютера создается иллюзия одновременного выполнения.

Для иллюстрации этого процесса рассмотрим такой пример: все люди любят читать книги по-разному. Одни, действуя в соответствии с принципом кооперативной многозадачности, любят читать только одну книгу и не берутся за другую до тех пор, пока не прочитают предыдущую до конца. Но есть среди нас и те, кто любит читать сразу несколько книг одновременно. В действительности же они сначала читают определенное количество страниц из первой книги, затем возьмут в руки другую и прочитают несколько страниц из нее и т. д. Именно такой принцип и лежит в основе вытесняющей многозадачности. Кроме того, чтобы зафиксировать количество прочитанных стра-

ниц и не перечитывать один и тот же фрагмент книги несколько раз, мы используем закладки. Точно так же каждый поток использует специальную структуру данных, называемую контекстом, для хранения информации о состоянии потока, точнее — о состоянии регистров ЦПУ. Как только мы хотим продолжить читать некоторую книгу, мы открываем её на заложенной странице. Так же и в компьютере: как только ОС выделяет для выполнения некоторого потока квант времени, значения регистров ЦПУ, которые хранятся в контексте этого потока, загружаются в физический ЦПУ компьютера и поток начинает выполняться с того места, куда ссылается только что загруженный указатель выполнения. После того как мы прочитали некоторое количество страниц, мы кладем закладку на текущей странице, закрываем книгу и берем другую. Так же и в ОС: после того как время выполнения некоторого потока истекает, его контекст обновляется текущим состоянием регистров ЦПУ и в память загружаются значения ЦПУ из контекста следующего потока.

Однако не стоит думать, что все потоки в системе одинаковы. Некоторые потоки получают от ОС больше времени для выполнения, другие — меньше, в зависимости от установленного для данного потока приоритета. Чем выше приоритет потока, тем больше времени для выполнения он получит от ОС.

Для разработки многопоточного приложения в Delphi можно воспользоваться функцией WinAPI — `CreateThread` или экземпляром класса `TThread`.

## 10.2. Функция `CreateThread`

Для создания дополнительного потока наряду с главным потоком приложения в Delphi можно использовать функцию Windows Api — `CreateThread`. Для этого необходимо, во-первых, создать поток, и, во-вторых, описать функцию потока, которая будет выполняться каждый раз, когда ОС будет выделять квант времени для выполнения Вашего потока.

Рассмотрим формат функции `CreateThread`:

```
CreateThread(LpThreadAttributes, DwStackSize, LpStartAddress, LpParameter, dwCreationFlags, LpThreadId);
```

Параметр `LpThreadAttributes` принимает серию атрибутов безопасности. Если этот параметр установлен в `nil`, то используются атрибуты безопасности по умолчанию. Следующий параметр `DwStackSize` имеет тип `DWORD` и определяет размер (в

байтах) стека создаваемого потока. Если этот параметр установлен в 0, то размер стека создаваемого потока будет совпадать с размером стека главного потока приложения. При определении этого параметра следует помнить, что если Вы зададите слишком большое значение, превышающее размеры доступной памяти, то система не сможет создать новый поток. Однако если заданного заранее размера стека становится недостаточно для потока, то размер стека увеличивается автоматически. Параметр `LpStartAddress` указывает название функции потока, которая вызывается тогда, когда поток начинает выполняться. Здесь указывается имя функции потока, перед которым стоит знак «@». Параметр `LpParameter` имеет тип `Pointer` и используется для передачи функции потока какого-либо значения. Параметр `DwCreationFlags` позволяет передать определенные флаги, которые ассоциируются с потоком. Если указан флаг `CREATE_SUSPENDED`, то создается приостановленный поток (приостановленный поток создается, но не запускается на выполнение до тех пор, пока Вы не вызовете функцию `ResumeThread`). Если значение этого параметра установлено в 0, то поток запускается на выполнение сразу после создания. Последний параметр `LpThreadId` имеет тип `DWORD`. После создания потока этому параметру будет присвоен уникальный идентификационный номер.

Если создание потока прошло успешно, то значением функции станет указатель на вновь созданный поток, если поток создать не удалось, то возвращаемое значение функции будет установлено в 0. После того, как Вы создали неприостановленный поток, автоматически вызывается функция потока.

**Упражнение 10.2.1.** Создайте многопоточное приложение, включающее дополнительный поток, отвечающий за рисование движущейся по периметру окна программы окружности.

### Решение

Сохраните приложение в папке `Threads_1` — файл модуля под именем `Main.pas`, файл проекта — `Threads_1.dpr`.

Так как внешний вид Вашей формы не изменяется, то перейдем сразу к написанию программного кода.

Будем создавать новый поток при выполнении пользователем щелчка на форме. Поэтому создайте обработчик события `OnClick`.

В обработчике этого события необходимо создать поток. В разделе переменных процедуры опишите переменные:



**Примечание.** Контексты устройств формируют связь между программой и определенными внешними устройствами, в частности видеоплатой. Использование контекстов внешних устройств позволяет писать программы, работающие с различными типами внешних устройств.

Реализация движения окружности по периметру окна приложения достаточно проста. Ее идея состоит в том, что в каждый момент времени окружность отображается в новом месте окна приложения, при этом ее предыдущее положение «затирается» путем рисования прямоугольника, закрашенного цветом фона.

Пусть окружность начнет движение по часовой стрелке из верхнего левого угла формы. Зададим начальные значения переменных:

```
x := 10;
y := 10;
Direction := True;           {окружность будет двигаться по горизонтали вправо}

dirx := 1;
Diry := 1;
Dc := GetDC(Form1.Handle);    {определяем контекст устройства}
```

Для описания движения окружности воспользуемся бесконечным циклом рисования окружности. На самом деле цикл не является бесконечным, он завершает свое выполнение при завершении работы приложения. Цикл будет иметь вид:

```
Repeat
  //тело цикла
Until False;
```

В теле цикла нужно «затереть» предыдущее положение окружности. Воспользуемся функцией рисования закрашенного прямоугольника **FillRect**:

```
FillRect(HDC, Iprc, Hbr).
```

Параметр **HDC** задает контекст устройства, **Iprc** определяет указатель на прямоугольную область, **Hbr** задает цвет прямоугольника.

Для определения контекста устройства возьмем переменную **dc**, переменную **Rect** используем для задания прямоугольной области:

```
with Rect do
begin
  Left:=x; Top:=y;           {левый верхний угол прямоугольной области}
  Right:=x+D; Bottom:=y+D;   {правый нижний угол прямоугольной области}
end;
```

По умолчанию цвет формы определен цветовой константой `COLOR_BTNFACE+1`, используем ее для определения цвета закрашки прямоугольной области

Таким образом, функция рисования закрашенного прямоугольника будет иметь вид:

```
FillRect(dc, Rect, COLOR_BTNFACE+1);
```

После этого определим в зависимости от направления движения окружности ее новые координаты: если окружность движется по горизонтали, то требуется изменить значение координаты *x*, иначе — значение координаты *y*:

```
if Direction then x:=x + dirx
else y:=y + diry;
```

В зависимости от текущего положения окружности необходимо изменить направление ее дальнейшего движения. Существует четыре варианта.

1. Окружность достигла правого верхнего угла (значение координаты *x* превысило значение ширины формы за вычетом расстояния до края формы, например, 10, и диаметра окружности *D*). В этом случае необходимо изменить направление движения окружности следующим образом:

```
if x > Form1.ClientWidth - 10 - D then
begin
    Dirx:=-1;           {изменяем направления горизонтального движения
                        на движение справа налево}
    Direction:=False;   {изменяем направление движения на
                        вертикальное}
    x:=Form1.ClientWidth - 10 - D;      {значение координаты x
    при вертикальном движении по левому краю формы}
end;
```

2. Окружность достигла правого нижнего угла (значение координаты *y* превысило значение высоты рабочей области формы за вычетом расстояния до края формы и диаметра окружности):

```
if y > Form1.ClientHeight - 10 - D then
begin
    Diry:=-1;           {изменяем направление вертикального
                        движения окружности на движение снизу вверх}
    Direction:=True;    {изменяем направление движения на
                        горизонтальное}
    y:=form1.ClientHeight - 10 - D;     {значение координаты y
    при горизонтальном движении по нижнему краю формы}
end;
```

3. Окружность достигла левого нижнего угла (значение координаты  $x$  стало меньше расстояния до края формы, например, 10):

```

if  $x < 10$  then
begin
    dirX:=1;           {изменяем направление горизонтального движения
                        на движение слева направо}
    Direction:=False;  {изменяем направление движения
                        на вертикальное}
    x:=10;             {задаем значение координаты  $x$ 
                        при вертикальном движении по левому краю формы}
end;

```

4. Окружность достигла верхнего левого угла (значение координаты  $y$  стало меньше расстояния до края формы, т. е. меньше 10)

```

if  $y < 10$  then
begin
    dirY:=1;
    Direction:=True;
    y:=10;
end;

```

Теперь, когда новые координаты окружности определены, необходимо нарисовать окружность с помощью функции `Ellipse`. Формат этой функции таков:

```
Ellipse(Hdc, nLeftRect, nTopRect, nRightRect, nBottomRect);
```

Параметр `Hdc` задает контекст устройства, параметры `nLeftRect` и `nTopRect` — координаты левого верхнего угла окружности, `nRightRect` и `nBottomRect` — координаты правого нижнего угла окружности. Таким образом, функция рисования окружности будет иметь следующий вид:

```
Ellipse (dc, x, y, x + D, y + D);
```

Для того чтобы уменьшить скорость движения окружности, вызовем функцию `Sleep(cMilliseconds)`, которая приостанавливает выполнение потока на количество `cMilliseconds` миллисекунд:

```
Sleep(50);
```

Ниже приведен полный текст функции потока:

```

procedure PaintEllipse;
const D=5;
var
    x, y: integer;

```

```
Direction: Boolean;
dirx, diry: -1..1;
dc: hdc;
Rect: TRect;
begin
  dc:=getdc(Form1.Handle);
  x:=10; y:=10;
  Direction:=True;
  dirx:=1; diry:=1;
  repeat
    with Rect do
      begin
        Left :=x;
        Top:=y;
        Right:=x+D;
        Bottom:=y+D;
      end;
    FillRect (dc, Rect, COLOR_BTNFACE+1);
    if Direction then x:=x+dirX else y:=y+dirY;
    if x>Form1.ClientWidth - 10 - D then
      begin
        dirX:=-1; Direction:=False; x:=Form1.ClientWidth - 10 - D;
      end;
    if x<10 then
      begin
        dirX:=1; Direction:=False; x:=10;
      end;
    if y>Form1.ClientHeight - 10 - D then
      begin
        Direction:=True; dirY:=-1; y:=form1.ClientHeight - 10 - D;
      end;
    if y<10 then
      begin
        Direction:=True; dirY:=1; y:=10;
      end;
    Ellipse (dc, x, y, x+D, y+D);
    Sleep (50);
  until False;
end;
```

*Эксперимент.* Сохраните проект. Запустите приложение, выполните щелчок левой кнопкой мыши по форме. В случае правильного написания программного кода по периметру формы начнет двигаться окружность.

Попробуйте изменить размеры формы, переместить, свернуть и восстановить ее. Допустимы ли эти операции? Объясните, почему.

Модифицируйте код обработчика события OnClick формы: вместо функции создания потока, рисующего на форме окружность, вызовите подпрограмму PaintEllipse напрямую. Запустите приложение. Попробуйте переместить форму, минимизировать ее, восстановить, изменить ее размеры. Что происходит? Объясните, почему.

**Примечание.** Для того чтобы завершить работу вашего приложения, выполните команду Run/Program Reset. ♦

Итак, в упр. 10.2.1 было создано приложение, создающее два одновременно выполняющихся потока — главный поток приложения, позволяющий изменять размеры, формы, перемещать по экрану и т. д., и поток, реализующий движение окружности.

Как уже было сказано, не все потоки, выполняющиеся в системе, одинаковы. Некоторые потоки получают от ОС больше времени для выполнения, некоторые — меньше, в зависимости от установленного для данного потока приоритета. Чем выше приоритет потока, тем больше времени для его выполнения выделяется ОС.

В Delphi для установления приоритета потока используется функция

```
SetThreadPriority(hThread, nPriority);
```

Параметр hThread определяет указатель на поток, для которого определяется приоритет, nPriority — задает приоритет потока.

Если установление приоритета потока прошло успешно, то значение функции SetThreadPriority устанавливается в True, иначе — в False.

**Задание.** Используя справочную систему Delphi, определите основные типы приоритетов потоков.

**Упражнение 10.2.2.** Модифицируйте проект Threads\_1.dpr. При выполнении пользователем первого щелчка на форме создается первый поток, при выполнении пользователем второго щелчка на форме создается второй поток. В результате по периметру окна приложения будут двигаться две окружности. При этом для первого потока установите более низкий приоритет, а для второго — более высокий.



```
HThread2:=CreateThread(nil,0,@PaintEllipse,nil,0,ThreadId2);  
if not SetThreadPriority (HThread2,THREAD_PRIORITY_HIGHEST)  
    {устанавливаем для второго потока высокий приоритет}  
then showmessage('set priority failed');  
    {в случае неудачной попытки  
    установить приоритет потока выводим сообщение об ошибке}  
if (HThread2 = 0) then ShowMessage ('CreateThread error');  
end;  
end;
```

Для того чтобы отличать первый поток от второго, диаметр двигающейся окружности будем определять случайным образом. Для этого прежде всего в обработчике события создания формы OnCreate вызовите процедуру Randomize, которая инициализирует генератор случайных чисел. В процедуре PaintEllipse величину D, хранящую значение диаметра, опишите не в разделе констант, а в разделе переменных (как переменную целого типа). В теле процедуры PaintEllipse после задания начальных значений определите значение диаметра окружности:

```
D:=5 + Random(10);
```

значение диаметра будет изменяться в диапазоне от 5 до 14.

**Эксперимент.** Сохраните проект. Запустите приложение и выполните через небольшой интервал два щелчка на форме.

Убедитесь, что после каждого щелчка на форме из левого верхнего угла формы начнет двигаться окружность некоторого радиуса. Через определенный промежуток времени вторая окружность догонит и перегонит первую окружность. Объясните, почему это происходит. ♦

## Задания для самостоятельного выполнения

**10.1.** Модифицируйте код приложения Threads\_1.dpr, чтобы

- после щелчка правой кнопкой мыши на форме все созданные потоки завершали работу;
- после каждого щелчка правой кнопкой мыши на форме работу прекращал только один из выполняющихся потоков.

**Примечание.** Воспользуйтесь функцией TerminateThread(HThread, ThreadID).

**10.2.** Используя задачу из упр. 10.2.2, исследуйте основные типы приоритетов потоков.

**10.3.** Модифицируйте задачу из упр. 10.2.2 следующим образом: при создании потока предусмотрите возможность определения его приоритета пользователем.

## 10.3. Класс TThread

Второй способ создания многопоточных приложений в Delphi — использование экземпляра класса TThread. Появление этого класса в иерархии классов можно объяснить тем, что большая часть компонентов библиотеки VCL построена в предположении, что доступ к ним в каждый конкретный момент будет выполняться только из одного потока.

Рассмотрим некоторые свойства, методы и события класса TThread.

### Свойства

FreeOnTerminate:	указывает на то, освобождать ли при завершении процесса память, выделенную под экземпляр класса потока.
Boolean;	Если значение свойства установлено в True, то при завершении потока экземпляр класса автоматически освобождается.
	По умолчанию значение этого свойства установлено в False. В этом случае освобождение памяти, выделенной под потоковый объект, должно быть прописано в программном коде;
Handle:	дескриптор процесса. Эта величина может
THandle;	быть использована для управления потоком при использовании функций WinAPI;
Priority:	определяет приоритет потока и может
TThreadPriority;	принимать одно из следующих значений: tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical;
Suspended:	показывает, в каком состоянии
Boolean;	находится поток. Если это свойство установлено в True, то поток приостановлен. Приостановленный поток не будет выполняться до тех пор, пока его выполнение не возобновится вызовом метода Resume;
Terminated:	определяет состояние потока. Значение
Boolean;	True говорит о завершении работы потока;
ReturnValue:	возвращаемое значение при завершении
Integer;	потока.

## Методы

<b>constructor</b> Create(CreateSuspended: Boolean);	создает экземпляр класса потока. Параметр CreateSuspended указывает на то, нужно ли создавать приостановленный поток (True) или сразу запускать поток после его создания (False);
<b>procedure</b> Suspend;	приостанавливает выполнение потока;
<b>procedure</b> Resume;	возобновляет выполнение приостановленного потока;
<b>procedure</b> Terminate;	полностью прекращает выполнение потока;
<b>function</b> WaitFor: Integer;	ожидает завершения потока, возвращая затем код его завершения (ReturnValue);
<b>procedure</b> Synchronize (Method: TThreadMethod);	синхронизирует выполнение метода потока, позволяя ему работать параллельно с другими потоками;
<b>procedure</b> Execute;	абстрактный метод класса TThread (он <b>virtual</b> ; <b>abstract</b> ; должен быть переопределен). В этом методе содержится программный код, выполняющийся сразу после создания неприостановленного потока.

## События

Событие OnTerminate возникает, когда поток находится в стадии завершения.

**Упражнение 10.3.1.** Создайте приложение, которое выполняет поиск файлов по маске поиска и дает возможность просмотра найденных файлов.

### Решение

Создайте новый проект. Сохраните новое приложение в папке Threads\_3 — файл модуля под именем Main.pas, файл проекта — Threads\_3.dpr.

*1-й этап. Визуальное проектирование главной формы проекта (рис. 10.3.1)*

Установите значение свойства Caption формы в 'Поиск файлов', Name — в 'MainF'.

Положите на форму метку и окно ввода. Измените заголовок метки на 'Имя:'. У компонента Edit1 очистите свойство Text, а свойство Name установите в NameEdt.

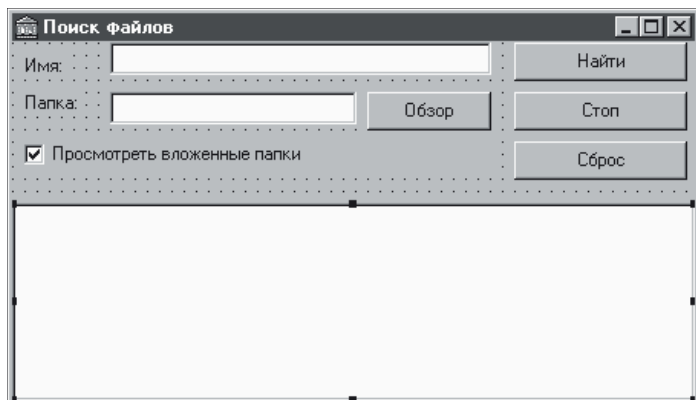


Рис. 10.3.1

Расположите на форме еще одну метку и окно ввода. Измените заголовок метки на 'Папка:'. У окна ввода очистите свойство Text, а свойство Name установите в PathEdt.

Положите на форму кнопку, измените ее заголовок на 'Обзор', а свойство Name установите в BrowseBtn. Ниже расположите компонент CheckBox. Измените заголовок компонента на 'Просмотреть вложенные папки' и установите свойство Checked в True.

Положите на форму компонент ListBox и выровняйте его по нижнему краю формы. Для этого свойство Align установите в AllBottom. В правой верхней части формы разместите друг под другом три компонента класса TButton и измените их свойства следующим образом:

Компонент	Свойство	Значение
Button1	Name	FindBtn
	Caption	Найти
	Enabled	False
Button2	Name	StopBtn
	Caption	Стоп
	Enabled	False
Button3	Name	ClearBtn
	Caption	Сброс
	Enabled	False

В результате главная форма приложения должна выглядеть так, как показано на рис. 10.3.1.

Сохраните проект.

### *2-й этап. Разработка программного кода модуля MAIN*

При запуске приложения кнопки 'Найти', 'Стоп' и 'Сброс' недоступны. Кнопка 'Найти' и кнопка 'Сброс' становятся доступными только тогда, когда в окно ввода NameEdt будет введена маска для поиска файлов. Для этого создайте обработчик события изменения содержимого окна ввода NameEdt:

```
procedure TMainF.NameEdtChange(Sender: TObject);  
begin  
  if NameEdt.Text<>' ' then  
    begin  
      ClearBtn.Enabled:= True;  
      FindBtn.Enabled:= True;  
    end  
  else  
    begin  
      ClearBtn.Enabled:= False;  
      FindBtn.Enabled:= False;  
    end;  
end;
```

При нажатии кнопки 'Сброс' окно ввода маски поиска и список файлов очищаются и все кнопки (за исключением кнопки 'Обзор') становятся недоступными. Обработчик события нажатия кнопки 'Сброс' имеет вид:

```
procedure TMainF.ClearBtnClick(Sender: TObject);  
begin  
  ListBox1.Clear;  
  NameEdt.Text :='';  
  FindBtn.Enabled:=False;  
  StopBtn.Enabled :=False;  
  ClearBtn.Enabled:=False;  
end;
```

При нажатии кнопки 'Обзор' должна появляться вторая форма, в которой пользователь будет иметь возможность выбрать нужную ему папку. Выполните команду меню File/New Form.

Измените заголовок новой формы на 'Обзор папок' (рис. 10.3.2), а имя на — 'OpenDirF'.

Положите на форму компонент DriveComboBox со страницы Win3.1 палитры компонентов. Под компонентом DriveComboBox расположите компонент DirectoryListBox. Установите значение

свойства DirList компонента DriveComboBox в DirectoryListBox1, чтобы при смене значения логического имени диска в DriveComboBox1 содержимое DirectoryListBox1 обновлялось.

В нижней части формы расположите компонент BitBtn со страницы Additional палитры компонентов и измените свойство Kind на bkOk.

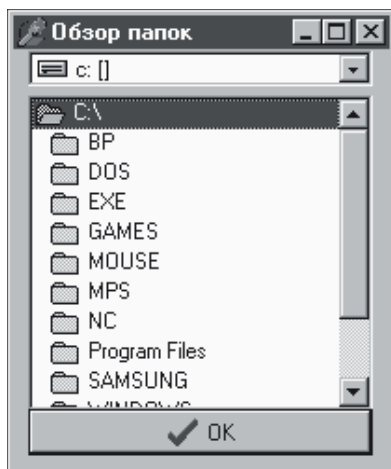


Рис. 10.3.2

Сохраните модуль под именем OpenDir.pas.

При нажатии кнопки BitBtn1 в окне ввода PathEdt главной формы проекта должен отображаться путь к выбранной пользователем папке, а форма обзора папок должна закрываться. Создайте обработчик события нажатия кнопки BitBtn1:

```
procedure TOpenDirF.BitBtn1Click(Sender: TObject);
begin
    MainF.PathEdt.Text:=DirectoryListBox1.Directory+'\';
    {отображаем путь к выбранной папке в окне ввода PathEdt
      главной формы проекта}

    close;
end;
```

В модуле OpenDir форма MainF, описанная в модуле Main.pas, недоступна. Подключите модуль главной формы, выполнив команду File/Use Unit и выбрав в отобразившемся списке модуль Main.

Вернемся к созданию программного кода в модуле Main.

При нажатии кнопки 'Обзор' должна появляться форма обзора папок. Для того чтобы вызвать эту форму, выполните команду File/Use Unit и выберите в списке OpenDir, после чего создайте обработчик события нажатия кнопки 'Обзор':

```
procedure TMainF.BrowseBtnClick(Sender: TObject);  
begin  
    OpenDirF.ShowModal;  
end;
```

**Эксперимент.** Сохраните проект. Запустите, после щелчка на кнопке 'Обзор' должно появляться окно выбора каталога.

Отображается ли название выбранного каталога в окне редактирования NameEdt? ♦

При нажатии кнопки 'Поиск' должен создаваться поток, осуществляющий поиск файлов. Для создания потока создадим новый класс, предком которого является TThread. Для этого выполните команду File/New... и в открывшемся диалоговом окне New Items дважды щелкните на иконке ThreadObject (рис. 10.3.3).

В появившемся диалоговом окне введите имя класса, например TSearchThread, и нажмите кнопку ОК.

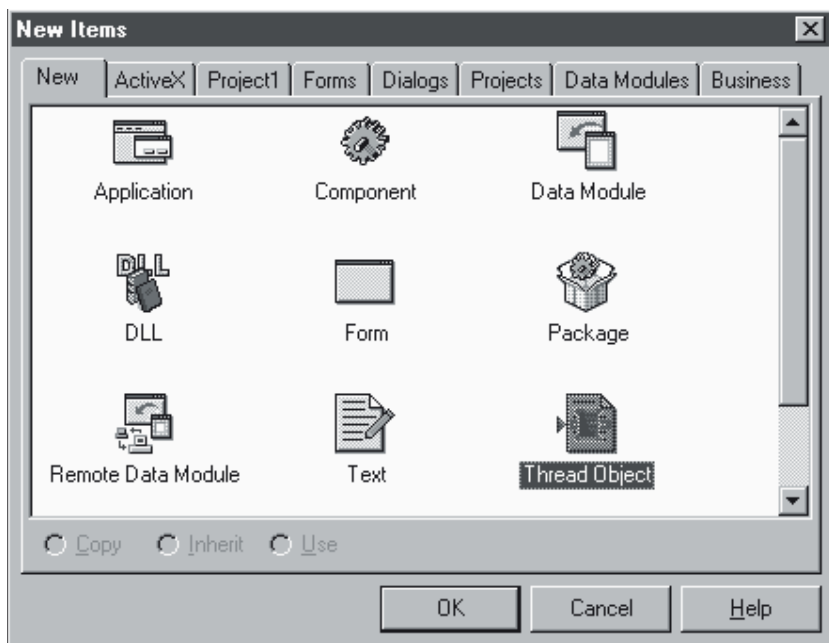


Рис. 10.3.3

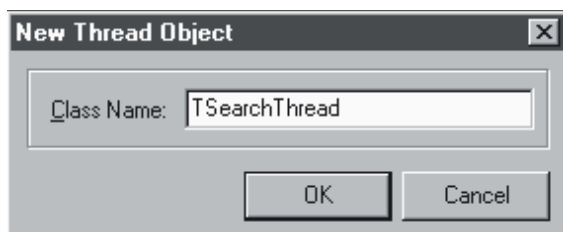


Рис. 10.3.4

Delphi автоматически создаст новый модуль, содержащий заготовку описания потомка класса `TThread`. Сохраните этот модуль под именем `SearchThread.pas`.

Итак, экземпляр класса `TSearchThread` должен выполнять поиск файлов с заданным именем в заданной папке, поэтому в разделе `Private` опишите переменные

**private**

```

FileName: string;           {для хранения имени файла}
Path: string;               {для хранения текущей папки}
SubDir: Boolean;             {если значение равно True, то поиск файлов
                             осуществляется и во вложенных папках }

```

Для создания экземпляра класса используется конструктор `TThread.Create`, в котором необходимо определить маску поиска, каталог, в котором следует искать, а также указать, осуществлять ли поиск во вложенных папках. В разделе `public` описания класса опишите

**constructor** `Create(FFileName, FPath: string; FSubdir: Boolean);`

В разделе `implementation` модуля создайте код:

```

constructor TSearchThread.Create(FfileName, FPath: string;
FSubDir: Boolean);
begin
  FileName:=FFileName;  {задаем значения глобальных переменных}
  Path :=FPath ;
  SubDir :=FSubdir;
  FreeOnTerminate := True;
  inherited Create(False); {обращаемся к методу класса-предка
                           для создания неприостановленного потока}
end;

```

После создания неприостановленного потока автоматически будет вызван метод `Execute`. Суть этого метода состоит в следу-

ющем: если значение логической переменной SubDir установлено в False, то необходимо произвести поиск заданного файла только в текущей папке, иначе если значение переменной SubDir установлено в True, то поиск заданного файла необходимо осуществить не только в текущей папке, но и во всех вложенных папках. Соответственно, необходимо определить две подпрограммы: процедуру FindFiles, реализующую поиск файла в текущей папке, и процедуру SearchDirs, реализующую рекурсивный поиск всех папок, вложенных в данную. Таким образом, метод Execute будет иметь вид:

```
procedure TSearchThread.Execute;  
begin  
  if not SubDir then FindFiles (FileName, Path)  
  else SearchDirs(FileName, Path);  
end;
```

Процедура FindFiles выполняет поиск файлов в указанной папке, соответственно в качестве параметров будем передавать маску поиска (FN) и название каталога (P). Опишите заголовок процедуры в разделе Private класса TSearchThread:

```
private  
...  
procedure FindFiles(FN, P : string);
```

Для поиска файлов воспользуемся функциями FindFirst, FindNext и процедурой FindClose.

Функция

```
function FindFirst(const Path: string; Attr: Integer;  
  var F: TSearchRec): Integer;
```

находит первый файл, заданный параметром Path (название каталога + маска поиска), с заданными атрибутами Attr. Если поиск был успешным, то функция возвращает значение ноль.

Параметр Attr определяет тип искомого файла. При этом выделяют следующие типы файлов:

- faReadOnly — файл с атрибутом только для чтения;
- faHidden — скрытый файл;
- faSysFile — системный файл;
- faDirectory — файл каталога;
- faArchive — архивный файл;
- faAnyFile — любой файл.

Если указывается тип файла faDirectory, то в список файлов будут включены каталоги.

Параметр **F** — это переменная типа **TSearchRec**, где

```
TSearchRec = record
  Time: Integer;
  Size: Integer;
  Attr: Integer;
  Name: TFileName;
  ExcludeAttr: Integer;
  FindHandle: THandle;
  FindData: TWin32FindData;
end;
```

Эта переменная хранит информацию о найденном файле. Наиболее важным значением этой переменной является поле **Name**, которое определяет имя найденного файла.

В отличие от функции **FindFirst**, функция **FindNext** предназначена для поиска следующего файла. Она принимает только один параметр — переменную типа **TSearchRec**, так как предполагается, что маска и атрибуты файла остаются теми же.

После завершения последовательности **FindFirst/FindNext** требуется вызвать процедуру **FindClose**:

```
procedure FindClose(var F: TSearchRec);
```

Для реализации алгоритма поиска файлов в заданном каталоге по маске поиска введем две переменные **SR** типа **TSearchRec** (для хранения найденного файла) и **Found** целого типа данных (для определения результата поиска):

```
procedure TSearchThread.FindFiles(FN, P : string);
var SR: TSearchRec;
    found: integer;
begin
  Found := FindFirst(P+FN, faArchiveFile, SR);
    {поиск файла в папке, заданной параметром P. Маска
    поиска определена в FN}
  while (Found = 0) and not terminated do begin
    {продолжаем процесс поиска до тех пор, пока очередной поиск
    не даст результатов либо пока выполнение потока не будет завершено}
    FileName:=SR.Name;
    {в переменную потока сохраняем имя найденного файла}
    Synchronize (AddFileFound);
    {для добавления полного имени найденного файла
    в компонент ListBox главной формы (процедура AddFileFound)
    вызовем метод synchronize}
    Found := FindNext(SR);
    {поиск следующего файла, удовлетворяющего условиям поиска}
```

```
end;  
FindClose(SR);  
end;
```

Как уже говорилось, большинство компонентов не являются потоко-безопасными. Метод `Synchronize` позволяет обезопасить работу с компонентами, вызывая некоторые методы потока (в задаче — `AddFileFound`) прямо из основного потока приложения (т. е. добавление очередного имени файла в `Listbox` будет осуществляться в потоке, выполняющем код модуля `Main`). Опишем процедуру `AddFileFound`:

```
procedure TSearchThread.AddFileFound;  
begin  
  MainF.ListBox1.Items.Add(Path+FileName);  
    {добавляем полное имя найденного файла в список главной  
    формы приложения}  
end;
```

Опишите заголовок процедуры `AddFileFound` в разделе `Private` класса `TSearchThread`, а чтобы обеспечить доступ к объектам главной формы приложения, в раздел `Uses` добавьте модуль `Main`.

Перейдем к написанию процедуры `SearchDirs`, которая должна найти все вложенные папки:

```
procedure TSearchThread.SearchDirs(FN, P : string);  
var sr: TSearchRec;  
    found: integer;  
begin  
  found:= FindFirst(P+'*.*', faDirectory, SR);  
    {поиск первой папки, вложенной в заданную. При этом в качестве  
    маски файла при вызове функции FindFirst указываем маску  
    *.* , а тип искомого файла устанавливаем в faDirectory,  
    чтобы включить каталоги в список просматриваемых файлов}  
  while (found=0) and not terminated do begin  
    if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1]<>'.') then  
      {если найденный файл, удовлетворяющий условиям поиска,  
      является каталогом}  
    begin  
      FindFiles(FN, P+SR.Name+'\\');  
        {поиск файлов в найденном подкаталоге}  
      SearchDirs(FN, P+SR.Name+'\\');  
        {поиск папок, вложенных в найденный подкаталог}  
    end;  
    found:= FindNext(SR);  
      {поиск следующей папки, вложенной в текущую}
```

```

end;
FindClose(SR);
end;

```

На этом описание методов класса TSearchThread завершено. Вернемся к написанию модуля Main.

Поток, реализующий поиск файлов, должен быть создан после активизации кнопки 'Найти'. Создайте обработчик события OnClick.

Изменим состояние кнопок: кнопку 'Стоп' нужно сделать доступной, а кнопку 'Найти' — недоступной, т. е.

```

StopBtn.Enabled := True;
FindBtn.Enabled := False;

```

Вызовем конструктор Create для создания экземпляра потока:

```

Hthread := TSearchThread.Create(NameEdt.Text, PathEdt.Text,
CheckBox1.Checked);

```

Переменную Hthread типа TSearchThread, указатель на создаваемый поток, опишем в разделе private описания класса формы.

Для определения действий, которые необходимо выполнить после того как поток закончит свою работу, опишем процедуру ThreadDone и установим на нее ссылку следующим образом:

```

HThread.OnTerminate:= ThreadDone.

```

Полный код обработчика события нажатия кнопки FindBtn будет иметь вид:

```

procedure TMainF.FindBtnClick(Sender: TObject);
begin
  StopBtn.Enabled :=True;
  FindBtn.Enabled :=False;
  Hthread:=SearchThread.Create(NameEdt.Text, PathEdt.text,
CheckBox1.Checked);
  HThread.OnTerminate :=ThreadDone;
end;

```

В процедуре ThreadDone необходимо очистить окно ввода маски поиска (NameEdt), кнопки 'Стоп' и 'Поиск' сделать недоступными, а также освободить область памяти, выделенную под поток:

```

procedure TMainF.ThreadDone(sender: TObject) ;
begin
  NameEdt.Text :='';
  FindBtn.Enabled :=False;

```

```
FindBtn.Enabled :=False;  
HThread.Free;  
end;
```

Опишите заголовок процедуры ThreadDone в разделе private описания класса формы.

**Эксперимент.** Сохраните проект. Попробуйте найти все файлы на диске c:\. Для этого в строке NameEdt введите \*.\*. Выполняется ли поиск файлов во вложенных каталогах? ♦

По условию задачи необходимо обеспечить возможность просмотра найденных файлов. Для запуска файла воспользуемся функцией ShellExecute, описание которой находится в модуле shellapi (опишите его в разделе Uses модуля Main). Эта процедура позволяет вызывать не только выполняемые программы, но и приложения, связанные с определенными расширениями файлов:

```
ShellExecute(hwnd, lpOperation, lpFile, lpParameters,  
lpDirectory, nShowCmd);
```

Параметр hwnd определяет идентификатор родительского окна, которое получает все генерируемые приложением сообщения.

Параметр LpOperation задает операцию, которую необходимо выполнить. Он может принимать одно из двух значений: 'Open' — для открытия заданного файла или 'Print' — для печати. Если значение параметра установлено в 0, то будет выполнена операция открытия файла.

Параметр lpFile задает полное имя файла. Если этот параметр определяет исполняемый файл, то способ отображения приложения задается параметром nShowCmd, в котором определяется способ отображения окна (максимального размера, свернутое в панель задач и др.)

В lpParameters описываются параметры запускаемого файла.

Параметр lpDirectory задает каталог по умолчанию.

После выполнения пользователем двойного щелчка мышью на полном имени файла будем открывать соответствующий файл. Создадим обработчик события OnDbClick для компонента ListBox1:

```
procedure TMainF.ListBox1DbClick(Sender: TObject);  
begin  
  ShellExecute(Handle, 'open', PChar(lb.Items[lb.itemindex]),  
nil, nil, SW_SHOWNORMAL);  
end;
```

**Эксперимент.** Сохраните и запустите проект. Можете ли Вы открыть один из найденных файлов, если поиск файлов еще не закончен, и почему? ♦

**Примечание.** Исходные тексты файлов модулей и форм приложения приведены в приложении 3.

### **Задания для самостоятельного выполнения**

- 10.4. Напишите многопоточковое приложение, одновременно выполняющее сортировку одного и того же массива разными способами.
- 10.5. Напишите многопоточковое приложение «Клавиатурный тренажер». Пользователь имеет возможность задать один из трех уровней: новичок, любитель, мастер. В зависимости от уровня, в каждый момент времени от верхней до нижней границы рабочей области тренажера падает одна, две или три буквы. От пользователя требуется успеть за время падения буквы нажать соответствующую клавишу на клавиатуре.
- 10.6. Напишите многопоточковое приложение, иллюстрирующее скорость работы вещественных типов данных Single, Real48, Double, Extended на примере операции сложения.
- 10.7. Напишите приложение, демонстрирующее поиск решения задачи «Лабиринт».

Формулировка задачи о лабиринте. Дано прямоугольное поле размера  $M \times N$ . На поле задана произвольная система препятствий, начальная клетка, в которой находится Черепашка, и конечная клетка. Найдите маршрут выхода Черепашки из лабиринта, если он существует, и пронумеруйте клетки маршрута в том порядке, в котором проходит их Черепашка. Черепашка может делать шаг на одну клетку в любом из четырех направлений: влево, вправо, вверх, вниз.

- 10.8. Машина Поста. Напишите программу, демонстрирующую работу машины Поста.

Описание машины Поста. Информационная лента разбита на секции, число которых бесконечно. Информация на ленту заносится в двоичном алфавите (в качестве одной из букв двоичного алфавита используется символ «V», который может быть помещен в любую секцию ленты; второй буквой будет пустота в секции). Конкретное состояние ленты с указанием, где расположена каретка, определяет состояние машины. С помощью каретки машина может распознавать, является ли конкретная секция ленты, под которой распо-

является ли конкретная секция ленты, под которой расположена каретка, отмеченной или неотмеченной. Каретка может стереть метку, если она имеется в секции, может поместить метку в пустую секцию. С помощью каретки осуществляется побуквенное преобразование конкретного двоичного слова.

Задачей машины Поста является преобразование состояния информационной ленты. Преобразование это машина осуществляет, руководствуясь алгоритмом, разработанным человеком.

Система команд Поста содержит всего шесть команд:

$a > b$  — команда, по которой машина сдвинет каретку вправо и перейдет к выполнению команды с номером  $b$  (здесь  $a$  — номер команды, выполнив которую машина двинется на одну секцию вправо;  $b$  — номер следующей команды);

$a < b$  — команда сдвига каретки влево;

$a V b$  — команда, по которой машина поставит символ «V»;

$a \wedge b$  — команда «стереть» метку;

$a !$  — команда «стоп»;

$a ? b c$  — команда передачи условия по содержимому обозреваемой секции: если в обозреваемой кареткой секции имеется метка, то в качестве следующей команды выбирается команда с номером  $b$ , если секция не была отмечена, то команда с номером  $c$ .

### Вопросы для повторения

1. Поясните термины «файл», «процесс», «поток».
2. Что такое многозадачность? Расскажите о видах многозадачности.
3. Что такое приоритет потока? Как определяется приоритет потока?
4. Как реализуется многопоточное приложение в Delphi?
5. Объясните назначение метода Synchronize экземпляра потомка класса TThread.

### 11.1. Принципы построения баз данных

Среда программирования Delphi предоставляет все необходимые инструменты для создания приложений баз данных — программ, которые могут работать с многочисленными типами баз данных, реализованных на различных платформах.

Одним из достоинств программирования баз данных в Delphi является использование открытой архитектуры доступа к данным, в основу которой положена концепция, построенная на использовании как реляционного (с помощью SQL-запросов), так и навигационного (последовательного) доступа к различным базам данных (БД). Возможность подключения к локальным и удаленным базам данных обеспечивается процессором баз данных Borland Database Engine (рис. 11.1.1).

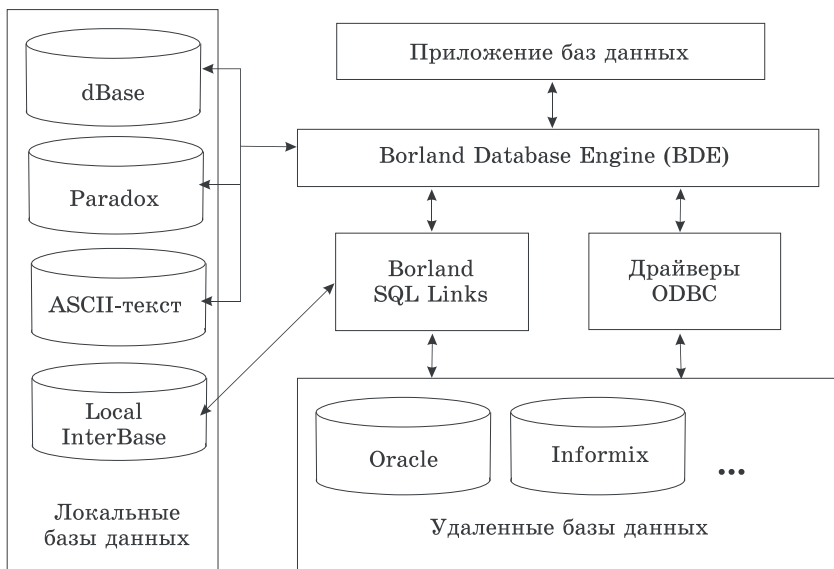


Рис. 11.1.1

Borland Database Engine (BDE) предоставляет единый набор функций (API) обработки локальных и серверных данных. Обращение к различным источникам данных основывается на концепции драйверов. Таким образом, обеспечивается интерфейс к наиболее распространенным форматам/способам хранения данных: DBase, Paradox, ASCII-файлам, для связи с серверами третьих фирм BDE использует стандартный интерфейс открытого подключения к БД (Open DataBase Connectivity (ODBC)), а к большинству популярных SQL-серверов BDE подключается через Borland SQL. Кроме того, библиотека визуальных компонент содержит компоненты библиотеки ADOExpress для работы с источниками данных Microsoft ActiveX Data Objects (ADO).

Основным достоинством этого подхода является то, что можно подключиться к новым базам данных, добавив соответствующий драйвер БД.

Прежде чем перейти непосредственно к разработке приложений баз данных в Delphi, рассмотрим основные понятия теории реляционных баз данных.

Реляционная база данных — это набор логически связанных между собой таблиц. Требования к проектированию реляционной базы данных заключаются в следующем [10, с. 552–553]:

- каждая таблица имеет уникальное в базе данных имя и состоит из однотипных строк — записей;
- каждая таблица состоит из фиксированного числа колонок (полей) и значений. В одной колонке не может быть сохранено более одного значения;
- ни в какой момент времени в таблице не найдется двух записей, дублирующих друг друга. Записи должны отличаться хотя бы одним значением, чтобы была возможность однозначно идентифицировать любую запись таблицы. Уникальность значения каждой записи таблицы задается первичным ключом (первичным индексом), в качестве которого часто используется целое число, увеличивающееся при добавлении новой записи в таблицу;
- каждому полю присваивается уникальное в пределах таблицы имя; для него устанавливается конкретный тип данных (дата, строка, ...);
- при выполнении обработки данных можно свободно обращаться к любой записи или любому полю таблицы. Значения, хранимые в таблице, не накладывают никаких ограничений на порядок обращения к данным.

Для предотвращения хранения избыточной информации в реляционных базах данных используется нормализация. Нормализация — это формальный аппарат ограничений на формирование таблиц, описывающий разбиение таблиц на две или более частей, обеспечивающий создание лучших методов добавления, изменения и удаления данных. Нормализация позволяет устранить дублирование, обеспечивает непротиворечивость хранимых данных и уменьшает трудозатраты на ввод и изменение данных.

Для получения информации из таблицы в заданном порядке используется один из способов. Первый — при каждом обращении к таблице производить поиск в таблице и сортировку ее в нужном порядке. Неоспоримым недостатком этого способа является значительное возрастание времени поиска информации в таблице. Второй — использовать вторичные индексы, что приводит к увеличению объема баз данных (для каждого вторичного индекса необходимо создать специальную служебную таблицу, в которой хранятся указатели на строки таблицы, упорядоченные в соответствии с вторичным ключом).

Между таблицами может быть установлено одно из следующих типов отношений:

- ☐ один-к-одному, если часть таблицы может использоваться несколькими таблицами одновременно или ряд полей в большинстве записей не будет заполняться;
- ☐ один-ко-многим, если для одной записи первой таблицы может существовать несколько записей во второй;
- ☐ многие-ко-многим, если для записи первой таблицы может существовать несколько записей во второй, и для записи второй таблицы — несколько записей в первой.

## 11.2. Компоненты, используемые для связи с базами данных

Компоненты для работы с БД расположены в библиотеке компонентов на страницах Data Access (рис. 11.2.1) и Data Controls (рис. 11.2.2).

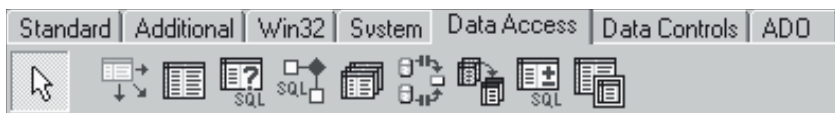


Рис. 11.2.1

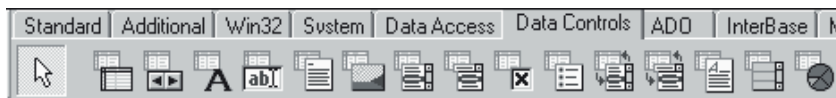


Рис. 11.2.2

Существует три типа компонентов, каждый из которых выполняет специфическую роль (рис. 11.2.3).



Рис. 11.2.3

Компоненты Table и Query служат в качестве интерфейса между физической таблицей на диске и остальной частью приложения. Эти компоненты обладают методами, которые открывают, закрывают, считывают, дополняют и выполняют другие манипуляции с дисковыми файлами. Компонент DataSource является интерфейсной прокладкой между компонентами, непосредственно связывающимися с БД, и воспринимающими данными компонентами, такими, как, DBGrid, DbImage и другими. Большинство из воспринимающих данные компонентов являются просто расширениями стандартных компонентов, которые знают, как получить информацию от DataSource.

### 11.2.1. Компонент Table

Основой архитектуры доступа к наборам данных (рис. 11.2.4) является базовый класс TDataSet, который содержит абстрактное представление записей и полей набора данных, инкапсулирующий управление, навигацию и манипулирование набором данных. Некоторые методы класса TDataSet могут быть переопределены с целью создания компонента, подключаемого к

определенному физическому формату данных. Исходя из этого, класс `TBDEDataSet` определен как производный от класса `TDataSet` и является основным классом источников данных; он вводит такие концепции, как BDE-базы данных и сеансы.



**Рис. 11.2.4**

Класс `TTable` представляет структуру и данные, содержащиеся в таблице базы данных, знает, как обрабатывать индексы и применять специальные приемы, связанные с поддержкой отношений двух таблиц типа один-ко-многим. Класс `TQuery` — набор данных, содержащий информацию, возвращенную в результате выполнения SQL-запроса.

Основные свойства класса `TTable`, унаследованные от `TDataSet`:

<code>AutoCalcFields</code>	определяет, когда вызвано событие <code>OnCalcField</code> ;
<code>Active</code>	определяет, открыт или нет набор данных;
<code>FieldDefs</code>	список имен полей из набора данных.

Свойство `DatabaseName`, унаследованное классом `TTable` от `TBDEDataSet`, определяет псевдоним базы данных, содержащей таблицу, либо имя каталога, в котором находятся файлы таблиц.

Основные свойства компонента `TTable`, унаследованные от `TBDEDataSet`:

<code>Filter</code>	выражение для отбора данных;
<code>Filtered</code>	значение <code>True</code> выполняет фильтрацию данных в соответствии с выражением, определенным свойством <code>Filtered</code> ;
<code>FilterOptions</code>	определяет, является ли фильтрование нечувствительным и разрешаются или нет частичные сравнения.

Основные свойства компонента `TTable`:

<code>MasterSource</code>	определяет источник данных для связи с другой таблицей;
<code>DataSource</code>	выбирает источник данных;

DefaultIndex	определяет, нужно ли сортировать данные в таблице;
Exclusive	определяет доступ к используемой таблице при одновременном обращении к ней нескольких приложений;
IndexDefs	содержит информацию об индексах таблицы;
IndexFieldNames	составляет список индексов для таблицы;
IndexFields	поля текущего индекса;
TableName	таблицы, доступные в данной базе данных;
TableType	тип таблицы.

#### Методы TTable:

CreateTable	создание таблицы;
Append	добавление новой записи в конец таблицы;
Delete	удаление записи, на которой стоит курсор;
DeleteTable	удаление таблицы;
Edit	перевод таблицы в режим редактирования записей;
FieldByName	обращение к значению поля записи по имени;
FindFirst	установка курсора на первую запись таблицы, соответствующей условию отбора;
First	установка курсора на первую запись таблицы;
Insert	перевод таблицы в режим вставки новой записи;
Next	перемещение курсора на следующую запись;
Post	перевод таблицы в режим завершения редактирования.

### 11.2.2. Компонент DataSource и компоненты отображения данных

Класс TDataSource обеспечивает канал связи, по которому компоненты доступа к данным могут подключаться к компонентам отображения данных.

Свойство State отображает текущее состояние связанного с ним набора данных, т. е. режиме, в котором находится таблица (вставки, редактирования, и др.).

Свойство DataSet используется для указания компонента, содержащего набор данных.

**Упражнение 11.2.1.** Создайте приложение для просмотра таблицы biolife.db.

#### Решение

*1-й способ.* Для отображения данных воспользуемся компонентом DBGrid.

Создайте новое приложение и сохраните его в папке BioLife\_1. Файл модуля — под именем Main.pas, файл проекта — BioLife\_1.dpr.

Из палитры компонентов страницы Data Access перенесите на форму компоненты Table и DataSource. Для отображения данных используем компонент TDBGrid со страницы Data Control (рис. 11.2.5).

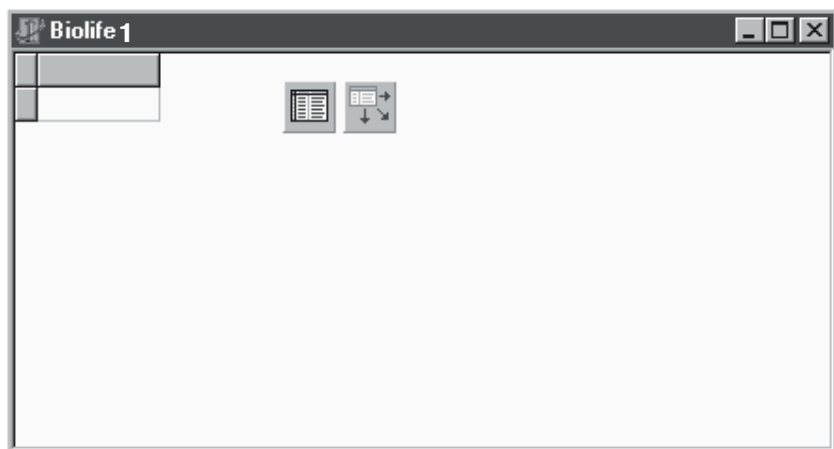


Рис. 11.2.5

Подготовка компонентов к отображению данных.

Установите свойство DataSource компонента DBGrid1 в значение DataSource1 (в выпадающем списке этого свойства перечислены все имеющиеся на форме источники данных).

Для связи компонентов DataSource1 и Table1 определите значение свойства DataSet компонента DataSource1 равным Table1.

Присвойте свойствам компонента Table1 следующие значения:

DataBaseName	DBDemos
TableName	biolife.db
Active	True

**Примечание.** Установка свойств компонента Table производится только в указанном порядке: DataBase, TableName, Active. Перед изменением имени набора данных (имени таблицы) свойству Active присвойте значение False.

В результате будет получено приложение, подобное изображенному на рис. 11.2.6.

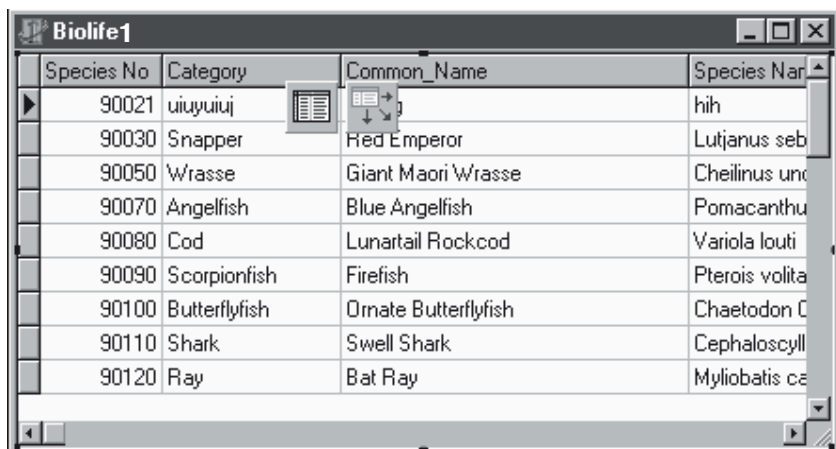


Рис. 11.2.6

**Эксперимент.** Запустите проект. Убедитесь, что полосы прокрутки позволяют просматривать все поля и записи таблицы. Что отображается в полях Graphic и Notes? ♦

**2-й способ.** Для отображения данных воспользуемся компонентами со страницы DataControl таким образом, чтобы данные отображались не в таблице, а в отдельных полях на форме.

Создайте новое приложение, сохраните файлы проекта в папке BioLife\_2 файл модуля под именем Main.pas, файл проекта — BioLife\_2.dpr.

Положите на форму четыре компонента DBEdit, четыре компонента Label и по одному компоненту DBImage, DBMemo, DataSource, Table (рис. 11.2.7.). Выделите компоненты воспроизведения данных и установите значение свойства DataSource в DataSource1.

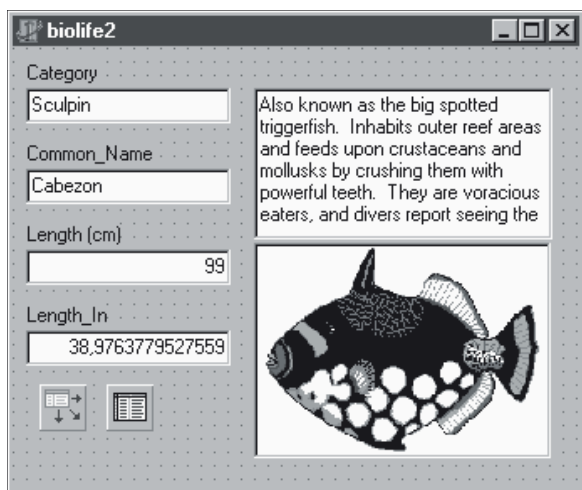


Рис. 11.2.7

Установите значения свойств компонентов следующим образом:

компонент	свойство	значение
DataSource1	DataSet	Table1
Table1	DataBaseName	DBDEMOS
	TableName	Biolife.db
	Active	True
DBEdit1	DataField	Category
DBEdit2	DataField	Common_Name
DBEdit3	DataField	Length (cm)
DBEdit4	DataField	Length_In
DBMemo1	DataField	Notes
DBImage1	Stretch	True
	DataField	Graphic

Свойству Caption компонентов Label присвойте значение, равное свойству DataField соответствующего компонента DBEdit.

**Эксперимент.** Запустите проект. Можно ли просмотреть все записи таблицы? ♦

Для навигации по таблице используется компонент DBNavigator (рис. 11.2.8) со страницы DataControl.



Рис. 11.2.8

Положите на форму компонент DBNavigator (рис. 11.2.9).

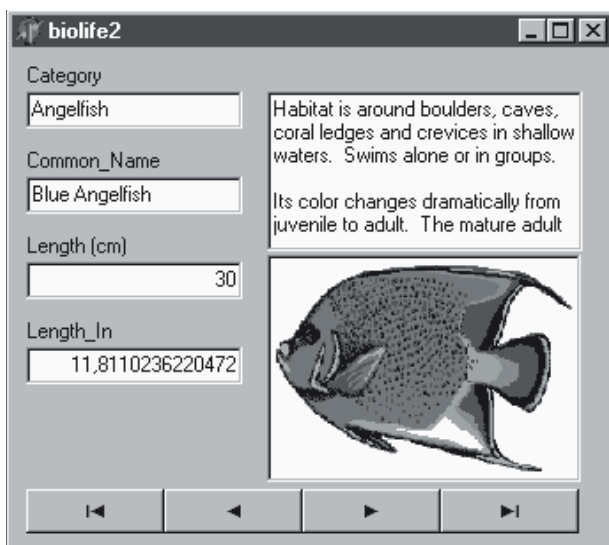


Рис. 11.2.9

Компонент объединяет в себе ряд кнопок:

- NbFirst      перемещение к первой записи;
- NbPrior      перемещение к предыдущей записи;
- NbNext      перемещение к следующей записи;

NbLast        перемещение к последней записи;  
NbInsert     вставка новой записи в место, указанное курсором;  
NbDelete     удаление текущей записи;  
NbEdit       редактирование текущей записи;  
NbPost       внесение изменений после редактирования в БД;  
NbCancel     отмена результатов редактирования или добавления  
              новой записи;  
NbRefresh    очистка буфера, связанного с набором данных.

**Примечание.** Для перемещения по записям таблицы можно воспользоваться методами компонента Table — First (на первую запись), Next (на следующую запись), Last (на последнюю запись), Pred (на предыдущую запись). Для определения начала и конца набора данных используются свойства EOF и BOF.

Для связи навигатора с набором данных установите свойство DataSource компонента DBNavigator в DataSource1.

Для просмотра таблицы biolife необходимы кнопки NbFirst, NbPrior, NbNext, NbLast. Удалите лишние кнопки навигатора, изменив значение свойства VisibleButtons (установите значение False для всех кнопок, кроме необходимых для работы приложения).

**Эксперимент.** Запустите приложение. Убедитесь, что кнопки навигатора позволяют перемещаться по записям таблицы. ♦

Чтобы грамотно оформить приложение, работающее с БД, необходимо добавить соединение с набором данных в момент начала работы и разорвать его в момент окончания.

Установите значение свойства Active компонента Table в False.

Создайте обработчик события OnCreate формы:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  Table1.Active := true;  
end;
```

В обработчике события OnDestroy формы напишите:

```
procedure TForm1.FormDestroy(Sender: TObject);  
begin  
  Table1.Active := false;  
end;
```

**Эксперимент.** Запустите приложение. Убедитесь, что ничего не изменилось в функционировании приложения. ♦

**3-й способ.** Использование редактора полей.

Отображение данных на форме в отдельных полях можно сделать автоматически.

Создайте новое приложение, сохраните файлы приложения в каталоге BioLife\_3, файл проекта под именем Main.pas, файл проекта — BioLife\_3.dpr.

Положите на форму компоненты DataSource и Table.

Измените значения свойств так же, как и при выполнении первого и второго способов.

Выполните двойной щелчок левой кнопкой мыши на компоненте Table1, появится редактор полей (рис. 11.2.10). Щелкните правой кнопкой мыши и выберите из всплывающего меню Add all Fields (рис. 11.2.11).

В Инспекторе объектов измените свойство DisplayLabel каждого поля на значение, соответствующее заголовку, например, Notes — описание. Из всплывающего меню выберите Select all. Перетащите поля на форму, в результате чего автоматически создаются компоненты, отображающие данные.

Положите на форму DBNavigator для перемещения по записям таблицы.

**Эксперимент.** Запустите приложение. Убедитесь в работоспособности созданного приложения. ♦



Рис. 11.2.10

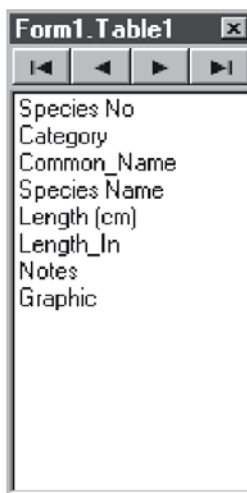


Рис. 11.2.11

**Упражнение 11.2.2.** Создайте приложение «Employees» для просмотра таблицы Employee базы данных DBDEMOS. Используя фильтрацию данных,

а) отобразите только тех работников, у которых зарплата больше 40000;

б) отобразите записи, поле EmpNo которых является двузначным числом;

с) выведите всех сотрудников, чьи фамилии начинаются на букву «L».

### Решение

Создайте новый проект. Сохраните файлы проекта в папке Employee.

Разработайте интерфейс приложения одним из трех описанных выше способов.

*Эксперимент.* Запустите приложения, обратите внимание на количество отображаемых записей. ♦

а) Фильтрация определяется свойствами Filter, Filtered, FilterOptions компонента Table.

В свойство Filter введите условие:

```
Salary > '40000',
```

так как нужно выбрать тех работников, у которых зарплата больше 40000.

Свойство Filtered установите в True.

*Эксперимент.* Запустите приложение. Осталось ли прежним количество отображаемых записей? ♦

При записи условий можно использовать операции отношения (>, <, =, >=, <=, <>) и логические операции (and, or, not). Например, фильтр

```
(FirstName = 'M*') and (Salary > '40000')
```

будет выводить людей, у которых имя начинается на букву «М» и зарплата больше 40000.

б) Процедура ApplyRange позволяет установить фильтр, который ограничивает диапазон записей для просмотра, используя индексированное поле.

В таблице Employee.db первое поле EmpNo является ключевым. Воспользуемся им для фильтрации данных.

Положите на форму кнопку «Фильтр», создайте обработчик события OnClick кнопки, в теле которого опишите такую последовательность действий.

Вызовите процедуру `SetRangeStart` и установите начала диапазона, используя свойство `Fields`:

```
Table1.SeyRangeStart;  
Table1.Fields[0].AsInteger:=10;  
                                {самое маленькое двузначное число}
```

По умолчанию свойство `Fields` содержит значения строкового типа. Свойство `AsInteger` позволяет преобразовать значение поля записи к целочисленному значению. Похожие свойства преобразуют значения полей к логическому (`AsBoolean`), вещественному (`AsFloat`) и формату даты (`AsDate`).

Вызовите процедуру `SetRangeEnd` и установите начала диапазона, используя свойство `Fields`:

```
Table1.SeyRangeEnd;  
Table1.Fields[0].AsInteger:=99;  
                                {самое большое двузначное число}
```

Вызовите команду `ApplyRange` для выполнения фильтра:

```
Table1.ApplyRange;
```

Положите на форму кнопку «Отменить фильтр». Чтобы отменить результаты вызова `ApplyRange`, используйте метод `CancelRange`:

```
Table1.CancelRange;
```

**Эксперимент.** Запустите приложение. Отфильтрованы ли записи в соответствии с поставленным условием? ♦

с) Фильтрация при помощи события `OnFilterRecord`.

Событие `OnFilterRecord` позволяет устанавливать фильтры на неключевых полях.

Создадим обработчик события `OnFilteredRecord` компонента `Table1`:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;  
  var Accept: Boolean);  
begin  
  accept:=(table1['LastName']>='L*') and (table1['LastName']<'M*');  
end;
```

**Примечание.** Для обращения к полю записи также можно воспользоваться способом, описанным выше:

```
accept := (table1.Fields[1].AsString >= 'L*') and  
(table1.Fields[1].AsString < 'M*');
```

Положите на форму компонент `CheckBox`. Измените свойство `Caption` на «Фильтр». Установка флажка будет приводить к фильтрованию записей, отмена флажка — к сбросу фильтра. Создайте обработчик `OnClick` компонента `CheckBox`:

```
procedure TForm1.CheckBox1Click(Sender: TObject);  
begin  
  Table1.Filtered:=CheckBox1.Checked;  
end;
```

*Эксперимент.* Запустите приложение. Сколько записей таблицы `Employee` отображается? ♦

### Задания для самостоятельного выполнения

**11.1.** Напишите приложение «Просмотрщик таблиц баз данных», позволяющее просматривать базы данных \*.db, \*.dbf. В заголовке формы должно отображаться название таблицы, а в строке — общее количество записей и номер текущей записи.

**Примечание.** Для получения имени таблицы воспользуйтесь функцией `ExtractFileName(<полное имя файла>)`, а для выделения пути до файла — `ExtractFilePath(<полное имя файла>)`.

**11.2.** Создайте приложение, демонстрирующее решение упр. 11.2.1.2 каждым из описанных выше способов (если это возможно).

**11.3.** Создайте программу, позволяющую отображать записи таблицы по произвольному фильтру.

**Примечание.** Для отображения названий полей воспользуйтесь свойствами `Table1.FieldCount` (количество полей таблицы) и `Table1.Fields[0].FieldName` (возвращает имя первого поля набора данных).

## 11.3. Создание баз данных с помощью Database Desktop

Для работы со структурой таблиц баз данных в Delphi используется программа `Database Desktop`, которая поставляется вместе с Delphi. `DataBase Desktop` — это программа-утилита, являющаяся своеобразным текстовым редактором для таблиц баз данных. Она позволяет создавать, редактировать и изменять структуру таблиц баз данных.

Выполните команду `Пуск\ Программы\ Borland Delphi 5\ Database Desktop`. Создайте таблицу базы данных СУБД `Paradox7` (рис. 11.3.1).

Реляционная база данных состоит из нескольких таблиц, каждая из которых имеет свою структуру.

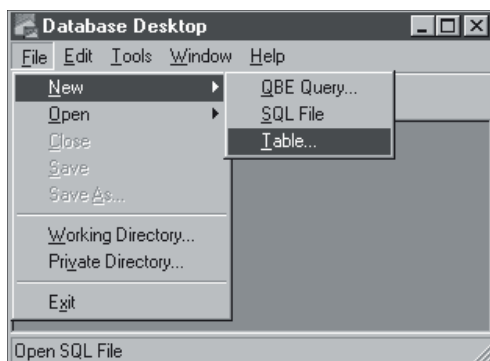


Рис. 11.3.1

Структура таблицы базы данных задается следующими свойствами:

- ☐ список полей, их тип и размер;
- ☐ наличие ключевых полей;
- ☐ наличие проверочных шаблонов для ввода данных;
- ☐ выбор национальной азбуки;
- ☐ наличие вторичных индексов;
- ☐ существование специальных ссылок к другой таблице.

Все это можно задать в главном окне программы Database Desktop (рис. 11.3.2).

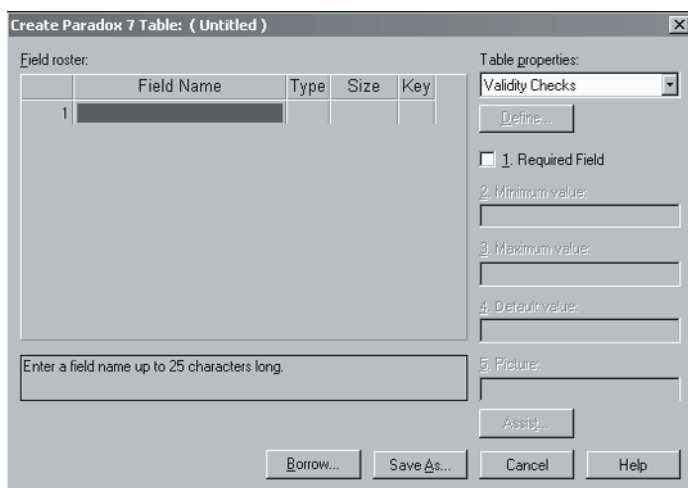


Рис. 11.3.2

Каждое поле таблицы определяется именем (Field Name), содержащим до 25 символов, и типом (Type).

Тип полей определяется форматом базы данных. Типы полей таблицы Paradox представлены на следующей странице.

Для некоторых типов необходимо задавать размер (Size). Например, для строкового типа Alpha размер — это число символов.

Первичный ключ (Key) — это поле, которое содержит данные, однозначно определяющие каждую запись в таблице. Ключевые поля должны быть отмечены символом '\*'. Двойной щелчок мыши (или клавиша пробел) ставит или удаляет этот значок.

В правой части окна (рис. 11.3.2) задаются свойства таблицы (Table Properties):

**Validity Checks (контроль правильности).** Это свойство определяет, какие данные можно вводить в текущее поле, а какие — нет. Способ определения зависит от типа условия — Required Field (обязательное поле), Minimum Value (минимальное значение), Maximum Value (максимальное значение), Default Value (значение по умолчанию) и Picture (шаблон).

**Table Lookup (связанные таблицы).** Наличие ссылочной таблицы позволяет ограничить значения, которые можно вводить в поле главной таблицы, значениями, которые содержатся в первом поле ссылочной таблицы.

**Secondary Indexes (вторичные индексы).** Вторичный индекс — это справочная таблица, которая определяет логический порядок записей в физической таблице на диске.

**Referential Integrity (справочная целостность)** обеспечивает связи между данными отдельных таблиц.

**Password Security (защита паролем)** позволяет задавать для таблицы или полей таблицы пароли.

**Table Language (выбор языка данных таблицы).**

**Dependent Tables (зависимые таблицы)** позволяет просмотреть список зависимых таблиц, связанных на уровне ссылок Referential Integrity.

После оформления структуры таблицы необходимо сохранить таблицу, выполнив щелчок на кнопке Save as... (рис. 11.3.2).

Для открытия таблицы используется команда File/Open.../Table.

Обозначение	Тип	Значение	Размер	Описание
A	Alpha		1..255	Строковый
N	Number	-10 <sup>907</sup> to 10 <sup>408</sup>		Вещественный
\$	Money			Денежный
S	Short	-32768..32767		Короткое целое
I	Long Integer	-2147483648 .. 2147483647		Длинное целое
#	BCD		0..32	Поле двоично-десятичного кода
D	Data			Даты
T	Time			Время
@	TimeStamp			Время и Дата
M	Memo		1..240	Текст
F	Formatted Memo		0..240	Форматированные тексты
G	Graphic			Графический
O	OLE			Содержат объекты, размещенные в таблице из других приложений Windows, которые поддерживают стандарт OLE
L	Logical			Логический
+	Autoincrement			Содержит целое значение, которое увеличивается на единицу при каждой новой записи в таблицу
B	Binary		0..240	Содержит данные, которые Paradox не может интерпретировать (например, звуковые)
Y	Bytes		1..255	Содержит данные, которые Paradox не может не только интерпретировать, но и читать

Команда Table/EditData переводит таблицу в режим редактирования записей.

Команда Table/Info Structure позволяет просмотреть структуру таблицы, а Table/Restructure — изменить структуру таблицы.

**Упражнение 11.3.1.** Создайте приложение «Автостоянка». Это приложение отображает работу нескольких автостоянок. Автомобили могут приезжать или уезжать с какой-либо автостоянки. Все изменения фиксируются в БД.

### Решение

Создайте каталог CarPark.

Написание приложения начнем с разработки базы данных. Создайте подкаталог BD каталога CarPark для хранения таблиц базы данных.

Таблица autostation.db предназначена для хранения информации о стоянках:

	Название поля	Тип (размер)	Первичный ключ
Номер автостоянки	NAutostation	Alpha (5)	*
Количество мест	Seats	Short	
Количество занятых мест	BuzySeats	Short	

В таблицу ncars.db будем записывать информацию об автомобилях, находящихся на стоянках:

	Название поля	Тип (размер)	Первичный ключ
Номер машины	NumberCar	Alpha (10)	*
Номер автостоянки	NAutostation	Alpha (5)	

Разработка визуального интерфейса приложения (рис. 11.3.3).

Положите на форму по два компонента Table, DataSource, DBGrid. Установите значения свойств следующим образом:

компонент	свойство	значение
Table1	Name	ParkTbl
	DataBaseName	... \CarPark\DB
	TableName	autostation.db
	Active	True
Table2	Name	CarsTbl
	DataBaseName	... \CarPark\DB
	TableName	ncars.db
	Active	True
DataSource1	Name	ParkDsc
	DataSet	ParkTbl
DataSource2	Name	CarsDsc
	DataSet	CarsTbl
DBGrid1	Name	ParkDbg
	DataSource	ParkDsc
DBGrid2	Name	CarsDbg
	DataSource	CarsDsc

**Эксперимент.** Сохраните файл модуля под именем Main.pas, файл проекта — CarPark.dpr. Запустите приложение. Убедитесь, что данные таблиц отображены на форме. ♦

Для определения взаимоотношения типа один-ко-многим между таблицами autostation.db и ncars.db воспользуемся свойством «связанные курсоры».

Запустите DataBase DeskTop и создайте для таблицы ncars.db вторичный индекс по полю NAutostation:

- ☐ откройте таблицу ncars.db;
- ☐ выполните команду Table/ Restructure...;
- ☐ измените свойство таблицы SecondaryIndexes. Для этого щелкните на кнопке Define, в левом столбце выберите поле NAutostation и переместите его в правый столбец;
- ☐ щелкните на кнопке ОК;
- ☐ в диалоговом окне Save Index As установите имя в ByNAutostation:
- ☐ сохраните таблицу и закройте DataBase Desktop.

Измените значения следующих свойств компонента CarsTbl:

MasterSource      ParkDsc  
MasterFields      NAutostation  
IndexName        ByNAutostation

**Эксперимент.** Запустите приложение. Как влияет изменение значений свойств компонента CarsTbl на отображение данных?

Попробуйте внести изменения в данные таблицы Cars.db.

**Примечание.** Для добавления новой записи нажмите кнопку Insert, для удаления записи — Ctrl+Delete, для редактирования — F2. ♦

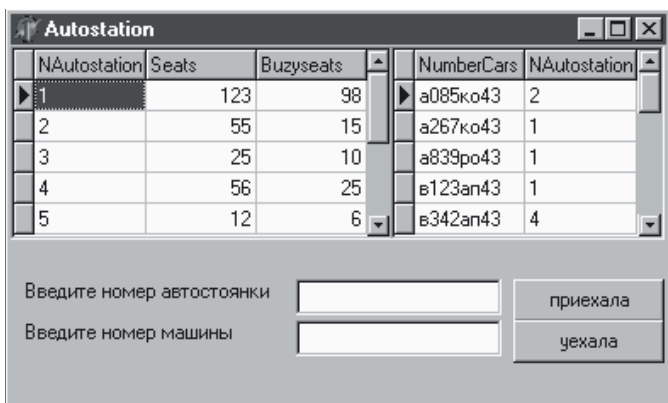


Рис. 11.3.3

Дополним приложение возможностью аудита автостоянки: по прибытии машины в приложение вводится номер машины и номер автостоянки (в TEdit2); если свободные места на выбранной автостоянке есть, запись добавляется в БД, иначе выводится сообщение, что мест нет. После освобождения машиной места на автостоянке соответствующая запись удаляется.

Положите на форму по два компонента Label, Edit и Button. В первый компонент Edit будем заносить номер автостоянки, во второй — номер автомобиля. После щелчка на первой кнопке машина добавляется в БД, на второй — удаляется.

Измените значения свойств Name компонентов следующим образом:

Edit1                    NAutostationEdt  
Edit2                    NCarEdt  
Button1                ArriveBtn  
Button2                LeaveBtn

Установите значения свойств Caption компонентов в соответствии с рис. 11.3.3.

Для определения количества свободных мест создайте вычисляемое поле.

Выделите компонент ParkTbl.

Правой кнопкой мыши вызовите контекстное меню и выберите редактор полей (Fields Editor).

Добавьте в редактор полей все поля таблицы: вызовите контекстное меню редактора полей и выполните команду Add fields (добавить поля). В появившемся диалоговом окне щелкните на кнопке ОК.

Для добавления нового поля еще раз вызовите контекстное меню и выполните команду New field (новое поле). В диалоговом окне создания нового поля установите свойства поля:

Name	FreeSeats
Type	integer
Size	0
Field type	Calculated

Создайте обработчик события OnCalcFields компонента ParkTbl:

```
procedure TForm1. ParkTblCalcFields(DataSet: TDataSet);  
begin  
    ParkTblFreeSeats.Value := ParkTblSeats.Value -  
        ParkTblBuzySeats.Value;  
end;
```

Создайте обработчик события OnClick компонента ArriveBtn.

Для определения наличия свободных мест на стоянке воспользуемся возможностью поиска записи по ключевому полю (NAutostation). Если номер стоянки введен верно и на ней есть свободные места, то занимаем одно из свободных мест, а в таблицу CarsTbl вносим номер машины. Если свободных мест нет, то выводится сообщение об отсутствии свободных мест.

```
procedure TForm1.ArriveBtnClick(Sender: TObject);  
begin  
    ParkTbl.SetKey;  
        {переключение таблицы в режим поиска по ключевому полю}  
try  
        ParkTbl.FieldName('NAutostation').AsString:=  
            NAutostationEdt.Text;  
                {установка значения для поиска}  
if not ParkTbl.GotoKey then raise EDataBaseError.create('');
```

```

        {ParkTbl.GotoKey — осуществляет поиск; если искомая запись
          не найдена, возбуждаем исключительную ситуацию}
    if ParkTbl.FreeSeats.Value=0 then begin
        {если свободных мест нет, вывод сообщения и
          удаление из NAutostationEdt номера выбранной автостоянки}
        MessageDlg('Извините, мест нет. Выберите другую автостоянку',
            mtInformation, [mbOk], 0);
        NAutostationEdt.Text:='';
    end
    else begin
        {если свободные места на стоянке есть}
        ParkTbl.Edit; {переключение в режим редактирования записей}
        ParkTbl.BuzySeats.Value:= ParkTbl.BuzySeats.Value+1;
            {увеличение счетчика занятых мест}

        ParkTbl.Post;
        CarsTbl.Insert;
            {переключение таблицы Cars.db в режим вставки новой записи}
        CarsTbl.FieldName('NumberCar').AsString := NCarEdt.Text;
        CarsTbl.FieldName('NAutostation').AsString :=
            NAutostationEdt.Text;
        CarsTbl.Post;
    end
except
    on EDataBaseError do
        begin
            MessageDlg('Номер автостоянки введен неверно', mtError,
                [mbOk], 0);
            NAutostationEdt.Text:='';
        end;
    end;
end;
end;

```

**Эксперимент.** Запустите приложение. Добавьте автомашину на стоянку, где есть свободные места, и туда, где их нет. Правильно ли работает приложение? ♦

## Задания для самостоятельного выполнения

### 11.1. Модифицируйте приложение «Автостоянка».

- ☐ Напишите обработчик события `OnClick` кнопки `LeaveBtn`.
- ☐ Замените компонент `NAutostationEdt` на `DBLookupComboBox1`, в котором должны отображаться номера всех автостоянок.
- ☐ Положите на форму еще одну кнопку, при нажатии на которую будет добавляться новая автостоянка.

## 11.4. Основы языка SQL

Акроним SQL обозначает Structured Query Language (структурированный язык запросов). В Delphi операторы SQL используются для просмотра таблиц, выполнения соединений между таблицами, создания отношений «один-ко-многим», а также для выполнения многих других операций, связанных с БД.

Общие правила синтаксиса SQL:

- ☐ не чувствительны к регистру;
- ☐ в конце каждого оператора ставится точка с запятой;
- ☐ комментарии заключаются между комбинацией символов `/* ... */`

### 11.4.1. Оператор выбора

Оператор выбора Select возвращает набор записей, удовлетворяющих условию

```
Select <список имен полей через запятую>
From <таблица>
[Where <условие отбора>]
[Order By <список имен полей>]
```

Раздел Select управляет списком полей, которые будут включены в набор данных. Если вместо списка указать символ «\*», то в набор данных будут включены все поля таблицы. В списке могут быть не только поля, но и любые выражения, построенные на основе полей и использующие арифметические операции.

После выражения может записываться псевдоним выражения в форме:

```
as <псевдоним> ,
```

который при отображении результатов будет фигурировать в заголовке.

Кроме того, оператор может содержать ключевые поля DISTINCT (в результирующий набор данных не будут включены повторяющиеся записи) и ALL (включение всех записей):

```
Select distinct <поле>
```

Ключевое слово From описывает таблицы или представления, связанные с изменяемой таблицей.

С помощью конструкции Where можно сузить количество обрабатываемых записей, определив одно или несколько условий. В результате в набор данных будут включены только данные, в которых заданные условия истинны.

Условие может включать имена полей, константы, логические выражения, арифметические операции и операции отношений.

=	Равно
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно
!= или <>	Не равно
and	Логическое умножение
like	Наличие заданной последовательности символов
between...and	Диапазон значений
not	Логическое отрицание
or	Логическое сложение
in	Соответствие элементу множества

Раздел Order By используется, когда необходимо отсортировать данные в результирующем наборе. Сортировка по убыванию значений осуществляется при помощи ключевого слова DESC

Order By <поле> DESC

**Упражнение 11.4.1.** Напишите выражение SQL, отображающее все записи таблицы biolife.db, в которых значение поля length не превышает 20 см. Отсортируйте записи результирующего набора по убыванию.

### Решение

```
Select *
From biolife.db
Where length_In<=20
Order By length_In desc
```

## 11.4.2. Объединение таблиц

В запросе можно объединить данные двух или нескольких таблиц. Объединение выдает записи независимо от того, есть ли соответствующее поле во второй таблице. Существуют три типа объединения:

- left outer join ... on — левое объединение. Включает в результат все записи первой таблицы, даже те, для которых не имеется соответствия во второй.

- `right outer join ... on` — правое объединение, включает в результат все записи второй таблицы, даже те, для которых не имеется соответствия в первой.
- `full outer join ... on` — полное объединение, объединение записей обеих таблиц, независимо от их соответствия.

### 11.4.3. Операции с записями

Вставка новой записи в таблицу осуществляется командой

```
insert into <имя таблицы> (<список полей>)  
values (<список значений>)
```

В списке перечисляются только те поля, значения которых известны. Для пропущенных полей значения берутся по умолчанию или остаются пустыми.

Изменение данных чаще всего производится с помощью команды `Update`, позволяющей выполнить как простое обновление данных в колонке, так и сложные операции модификации данных во множестве строк таблицы:

```
Update <имя таблицы> set <список вида<поле>=<выражение>  
Where <условие>
```

Удаление данных из таблицы выполняется построчно. За одну операцию можно выполнить удаление как одной строки, так и нескольких тысяч строк:

```
Delete from <имя таблицы>  
Where <условие>
```

### 11.4.4. Операции над таблицей

Для создания таблицы используется команда `Create Table`:

```
Create Table <имя таблицы>  
(  
  ИмяПоля ТипДанных  
  [NULL | NOT NULL]  
  [IDENTITY [<Начальное Значение, Шаг>] ]  
  [DEFAULT <Выражение>]  
  CHECK (<Логическое Выражение>  
)  
  PRIMARY KEY <Поле[, ...n]>  
  UNIQUE <Поле[, ...n]>  
  [FOREIGN KEY] <Имя Таблицы> (Поле Таблицы)
```

Чтобы создать таблицу, прежде всего необходимо определить ее имя, после чего следует описание полей таблицы, состоящее из имени поля и типа хранимых в нем данных. При описании могут быть использованы следующие ключевые слова:

NULL NOT NULL	разрешает или запрещает соответственно использование в данном поле неопределенных значений;
IDENTITY	предписывает осуществлять заполнение поля автоматически, для этого указывается начальное значение и шаг приращения. При этом значения поля не могут быть изменены, недопустимы значения NULL. В таблице может быть только одно поле с таким ограничением, имеющее либо целый, либо десятичный тип данных;
DEFAULT	определяет значение по умолчанию, которое будет использовано, если при вводе строки явно не указано другое значение.

Для определения свойств таблицы используются следующие ключевые слова:

**PRIMARY KEY** определяет колонку как первичный ключ таблицы;

**CHECK** накладывает на поле проверочное ограничение;

**FOREIGN KEY** определяет поле как внешний ключ таблицы.

**Упражнение 11.4.2.** Напишите выражение SQL, которое создает таблицу, содержащую сведения о студентах.

### Решение

```
CREATE TABLE Students  
(StudentID int IDENTITY(1,1)  
  LastName char(20) NULL,  
  FirstName char(20) NULL,  
  BirthDate datetime NULL)
```

## 11.4.5. Изменение структуры таблицы

Процесс изменения структуры таблицы практически ничем не отличается от процесса ее создания. К числу возможных модификаций относится добавление, удаление и изменение полей, добавление и удаление различных ограничений, которые осуществляются в результате команды:

```
ALTER TABLE <Имя Таблицы>  
ADD <Имя Поля>  
DROP <Имя Поля>
```

Для добавления нового поля в таблицу указывается ключевое слово **ADD**, после которого описываются параметры нового поля, так же как и при создании поля, командой **CREATE TABLE**.

Для удаления полей указывается ключевое слово **DROP**.

#### 11.4.6. Удаление таблиц

Для удаления таблицы необходимо выполнить команду

```
DROP TABLE <Имя Таблицы>
```

### 11.5. Компонент Query

«Предком» компонента **Query**, так же как и **Table**, является **TBDDataset**. Основное назначение компонента **Query** — обработка SQL-запросов.

#### Свойства TQuery

<b>DataSource</b>	позволяет строить приложения, содержащие связанные друг с другом таблицы;
<b>Params</b>	содержит параметры для выражений SQL;
<b>SQL</b>	формирование выражения SQL.

#### Методы TQuery

<b>ExecSQL</b>	осуществление любого запроса;
<b>ParamByName</b>	обращение к значению параметра SQL выражения;
<b>Prepare</b>	подготовка к выполнению запроса.

**Упражнение 11.5.1.** Используя компонент **Query**, отобразите информации о рыбах (таблица **biolife.bd**), длина которых не превосходит определенного значения.

#### Решение

Создайте новое приложение, сохраните файлы проекта в папке **BioLife\_Query**.

Положите на форму компоненты **Query**, **DataSource**, **DBGrid**.

Установите следующие значения свойств:

компонент	свойство	значение
DBGrid1	DataSource	DataSource1
DataSource1	DataSet	Query1
Query1	DatabaseName	DBDemos
	SQL	select *From biolife.db
	Active	True

**Эксперимент.** Запустите приложение. Убедитесь, что в таблице отображены все поля таблицы. ♦

Модифицируем приложение в соответствии с условием упражнения.

Положите на форму компоненты Label, Edit и Button, задайте следующие значения свойствам:

компонент	свойство	значение
Label1	Caption	Запрос: длина рыбы
Edit1	Text	
Button1	Caption	Выполнить запрос

Для решения задачи воспользуемся возможностью построения динамических запросов. Для формирования динамического запроса SQL используют параметры, которые применяют вместо имен таблиц, полей и значений SQL-выражений. В запросе параметр задается как

:<имя параметра>

Измените свойство Query1.SQL на следующее:

```
select * From biolife.db
where (Length_In > :ln)
```

Параметр :ln будет содержать значение длины, определяемое пользователем приложения.

Чтобы задать значения параметра, необходимо написать программный код. Все указанные в запросе параметры должны быть описаны в свойстве Params компонента Query.

Для каждого параметра необходимо установить следующие свойства:

DataType	тип данных параметра;
Name	имя параметра;
ParamType	тип параметра;
Value	значение параметра по умолчанию;
Type (Value)	тип значения по умолчанию.

При программном доступе параметры являются объектами типа TParam, образующими массив Params. Значения определяются свойствами, такими, как Value, asString, asInteger и т. п.

Создайте обработчик события OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Query1.Close;  
  Query1.Prepare;  
  Query1.Params[0].AsFloat:=StrToFloat(Edit1.Text);  
  Query1.Open;  
end;
```

**Эксперимент.** Запустите приложение. Убедитесь, что отображаются только записи, удовлетворяющие введенному условию запроса. ♦

Другой способ обращения к параметрам — это использование метода ParamByName:

```
Query1.ParamByName('ln').AsFloat:=StrToFloat(Edit1.Text);
```

**Упражнение 11.5.2.** Приложение «Расписание». Необходимо составить расписание преподавателей, т. е. распределить, в какой день, в какое время, в какой группе и в каком кабинете будут проходить занятия.

Предусмотреть возможность редактирования расписания и организацию поиска (по известным данным отображаются все соответствующие им данные).

### Решение

Для реализации приложения необходимо разработать следующие таблицы:

Teachers.db	— справочник, содержащий информацию о преподавателях;
Lessons.db	— справочник с номерами занятий и временем их начала;
WeekDay.db	— справочник о днях недели;

Group.db — справочник об учебных группах;  
 TimeTable.db — сводная таблица, представляющая расписание.

Создайте каталог TimeTable для хранения файлов приложения, внутри каталога создайте каталог DB для хранения таблиц базы данных.

Используя Database Desktop, разработайте структуру таблиц.

*Таблица TimeTable.db:*

	Название поля	Тип (размер)	Первичный ключ
День недели	WeekDay	Short	*
Номер пары	N_Para	Short	*
Номер группы	N_Group	Short	*
Код преподавателя	Teacher	Short	
Кабинет	Class	Alpha (15)	

Свойства таблицы:

Table Language (язык таблицы) установите в «Paradox Cyril 866»  
 Validity Checks (проверка корректности):

День недели (required field, minimum value=1, maximum value=7)

Номер пары (required field, minimum value=1, maximum value=8)

*Таблица Teachers.db:*

	Название поля	Тип (размер)	Первичный ключ
Код преподавателя	Kod_Teachers	Autoincrement	*
Фамилия	SecondName	Alpha (30)	

Свойства таблицы:

Table Language (язык таблицы) установить в «Paradox Cyril 866»

Таблица *Lessons.db*:

	Название поля	Тип (размер)	Первичный ключ
Номер пары	Lessons	Short	*
Время начала	StartTime	Time	

Свойства таблицы:

Table Language (язык таблицы) установить в «Paradox Cyrr 866»  
Validity Checks (проверка корректности):

Номер пары (required field, minimum value=1, maximum value=8)

Таблица *WeekDay.db*:

	Название поля	Тип (размер)	Первичный ключ
Код дня недели	Kod_WeekDay	Autoincrement	*
День недели	WeekDay	Alpha (30)	

Свойства таблицы:

Table Language (язык таблицы) установить в «Paradox Cyrr 866»

Таблица *Groups.db*:

	Название поля	Тип (размер)	Первичный ключ
Номер группы	Kod_Group	Autoincrement	*
Название группы	GroupName	Alpha (15)	

Свойства таблицы:

Table Language (язык таблицы) установить в «Paradox Cyrr 866»

### Задание для самостоятельного выполнения

Воспользуйтесь приложением, разработанным в результате выполнения задания 11.1.1. для заполнения таблиц. Введите в каждый справочник хотя бы по две записи, в таблицу TimeTable.db — не менее пяти (должны содержать записи по крайней мере двух преподавателей).

Разработайте визуальный интерфейс приложения.

Для отображения данных таблицы TimeTable.db положите на форму компоненты Table1, DataSource1, DBGrid1, измените свойство Name компонентов на TimeTableTbl, TimeTableDsc и TimeTableDBG соответственно. Измените свойства компонентов для отображения данных таблицы.

Сохраните приложение, файл модуля под именем Main.pas, файл проекта — TimeTable.dpr.

**Эксперимент.** Запустите приложение. Удобно ли для использования предлагаемое отображение данных? ♦

В связи с тем, что в таблицу TimeTable.db помещаются только коды, связанные с соответствующими данными, воспользуемся компонентами DBLookupComboBox для отображения данных из таблицы просмотра (т. е. будут отображать полные сведения о кодах, хранящихся в таблице TimeTable.db).

Поместите на форму компонент DBLookupComboBox1 для отображения фамилий преподавателей, Table1, DataSource1.

Установите свойства компонентов следующим образом:

компонент	свойство	значение
Table1	Name	TeacherTbl
DataSource1	Name	TeacherDsc
DBLookupComboBox1	Name	TeacherDcb
	DataSource	TimeTableDsc
	DataField	Teacher
	KeyField	Kod_Teachers
	ListField	SecondName
	ListSource	TeacherDsc

**Эксперимент.** Запустите приложение. Убедитесь, что компонент TeacherDcb отображает фамилию преподавателя, связанную с выбранной в таблице записью. ♦

Итак, в результате установленных свойств TeacherDcb получил список фамилий (ListField = SecondName) из таблицы Teachers (ListSource = TeacherDsc). Значение DataSource, равное TimeTableDsc, устанавливает связь с таблицей TimeTable.db. Используя свойство KeyField (ключевое поле), установили связь с полем Teacher из таблицы TimeTable.db.

**Задание для самостоятельного выполнения:** добавьте еще три компонента DBLookupComboBox для отображения дня недели, названия группы и времени начала занятия (рис. 11.5.1).

Для проверки корректности внесенных в таблицу TimeTable.db изменений поместите на форму компонент Query с именем CheckQur. Установите свойство DatabaseName в значение ...\\TimeTable\\DB.

Необходимо отследить правильность введенных данных. Например, не должно быть поставлено занятие для разных групп в конкретной аудитории в одно и то же время. Напишем код, предотвращающий такую ситуацию.

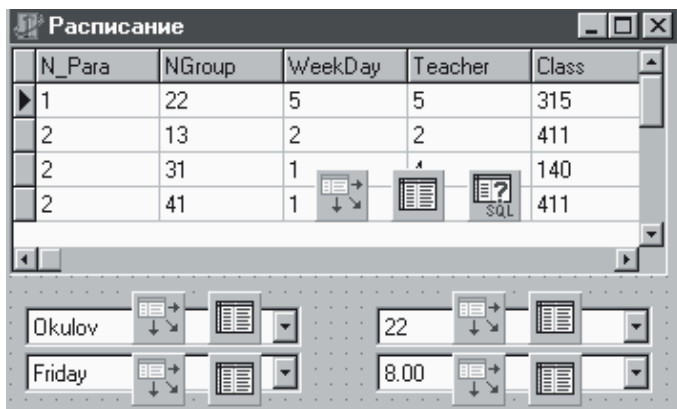


Рис. 11.5.1

### SQL-выражение

```
select * from TimeTable.db
where (N_Para = :NNPara) and (WeekDay = :NWeekDay)
and (Class Like :NClass) and not (N_Group = :NNGroup)
```

позволяет выбрать все записи, соответствующие занятиям, проходящим в одно время (номер пары и день недели) в одной и той же аудитории в разных группах. Сохраните запрос в текстовом файле Check\_Class.sql.

Воспользуемся обработчиком события AfterPost компонента TimeTableTbl.

В свойство SQL компонента CheckQur считаем выражение SQL из файла Check\_Class.sql, зададим значения каждому па-

параметру запроса. Если количество записей в полученном наборе данных оказалось больше одной, то необходимо откорректировать расписание: отображаем сообщение об ошибке и удаляем введенную запись:

```
procedure TForm1.TimeTableTblAfterPost(DataSet: TDataSet);  
begin  
    CheckQur.Close;  
    CheckQur.SQL.Clear;  
    CheckQur.SQL.LoadFromFile('Check_Class.sql');  
    CheckQur.Params[0].AsInteger:=  
        TimeTableTbl.Fields.FieldByName('N_Para').AsInteger;  
    CheckQur.Params[1].AsInteger:=  
        TimeTableTbl.Fields.FieldByName('WeekDay').AsInteger;  
    CheckQur.Params[2].AsString:=  
        TimeTableTbl.Fields.FieldByName('Class').Text;  
    CheckQur.Params[3].AsInteger:=  
        TimeTableTbl.Fields.FieldByName('N_Group').AsInteger;  
    CheckQur.Open;  
    if CheckOur.RecordCount>=1 then  
    begin  
        ShowMessage('Некорректный ввод данных. Данный кабинет  
            '+ CheckQur.Fields.FieldByName('Class').Text+' в это время  
            занят');  
        CheckOur.Close;  
        TimeTableTbl.Delete;  
    end;  
end;
```

**Эксперимент.** Запустите приложение. Убедитесь в правильности работы приложения.

Дополните запросы таким образом, чтобы предусматривались все способы некорректного ввода данных. ♦

### **Организация поиска**

Будем использовать компоненты DBLookupComboBox для ввода данных:

TeacherSearchDBC	—	фамилия учителя;
GroupSearchDBC	—	номер группы;
WeekDaySearchDBC	—	день недели;
ParaSearchBDC	—	начало занятия.

Свяжем компоненты с таблицами Teachers.db, WeekDay.db, Groups.db, Lesson.db. Для этого установим их свойства ListSource в TeacherDsc, WeekDayDsc, GroupDsc, LessonDsc соответственно. Установите свойства KeyField и ListField.

Положите на форму компонент DBGrid для отображения результатов поиска, компонент Query и DataSource.

Измените значения свойств на следующие:

компонент	свойство	значение
Query1	Name	SearchQur
	DatabaseName	...\TimeTable\DB\
DataSource1	Name	SearchDsc
	DataSet	SearchQur
DBGrid1	Name	SearchDbg
	DataSource	SearchDsc

Выражение SQL для поиска записей по выбранным названию группы, началу занятия, дню недели и фамилии преподавателя:

```
select * from TimeTable.db
where (N_Para = :CurPara) and (N_Group = :CurGroup)
and (WeekDay = :CurWeekDay) and (Teacher = :NTeacher)
```

Сохраните запрос в текстовом файле Search\_All.sql.

Обратите внимание, что в компонентах DBLookupComboBox отображаются фамилии, название дня недели, начало занятия и название группы, а в SearchDbg — только их коды. Следовательно, необходимо провести соответствие между этими компонентами, для чего понадобятся четыре дополнительных компонента Query: TeacherSearchQur, GroupSearchQur, WeekDaySearchQur и ParaSearchQur (не забудьте задать значения свойств DatabaseName).

TeacherSearchQur содержит запрос на выборку данных из таблицы Teachers, где фамилия учителя совпадает с введенной в TeacherSearchDBC. Запишем запрос в свойстве SQL компонента TeacherSearchQur.

```
select * from Teachers.db
where (SecondName like :NName)
```

**Задание для самостоятельного выполнения:** определите соответствующие свойства компонентов GroupSearchQur, WeekDaySearchQur и ParaSearchQur.

Положите на форму кнопку SearchBtn (рис. 11.5.2), при нажатии на которую будет осуществляться поиск.

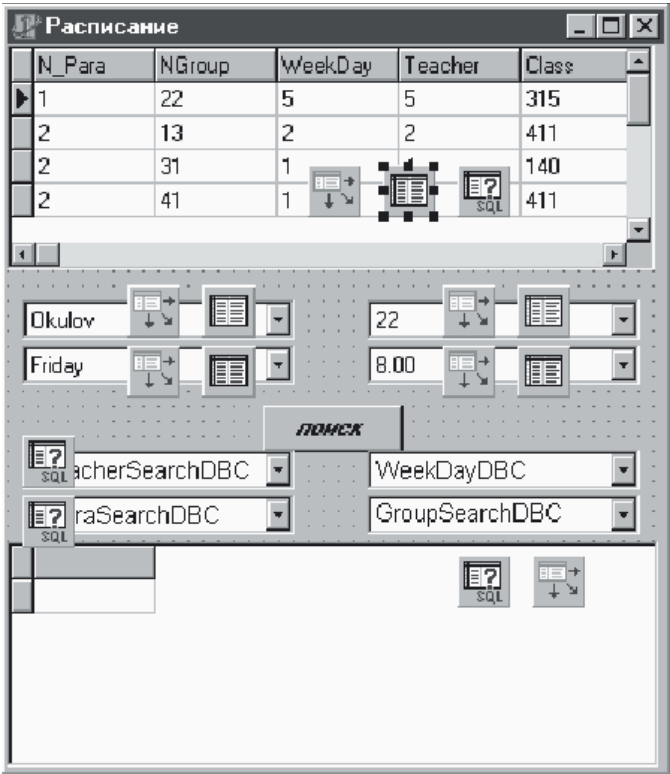


Рис. 11.5.2

В обработчике события OnClick кнопки зададим значения параметров запроса.

В параметре :Nname компонента TeacherSearchQur определим значение, содержащееся в TeacherSearchDBC:

```
TeacherSearchQur.Params[0].AsString:=TeacherSearchDBC.Text;
```

Результат запроса — значение поля Kod\_Teacher — занесем в переменную Teacher:

```
Teacher:=  
TeacherSearchQur.Fields.FieldByName('Kod_Teachers').AsInteger;
```

Разработайте аналогичный код для определения значений переменных WeekDay, Para, Group.

После выполнения предварительных операций осуществим запрос на выборку в SearchQur.

```
procedure TForm1.SearchBtnClick(Sender: TObject);  
var Teacher, WeekDay, Para : integer;  
    Group: string;  
begin  
    TeacherSearchQur.Params[0].AsString:= TeacherSearchDBC.Text;  
    TeacherSearchQur.Active:=true;  
    Teacher:=  
        TeacherSearchQur.Fields.FieldName('Kod_Teachers').AsInteger;  
    TeacherSearchQur.Active:=false;  
  
    GroupSearchQur.Params[0].AsString:= GroupSearchDBC.Text;  
    GroupSearchQur.Active:=true;  
    Group:=GroupSearchQur.Fields.FieldName('Kod_Group').Text;  
    GroupSearchQur.Active:=false;  
  
    WeekDaySearchQur.Params[0].AsString:= WeekDaySearchDBC.Text;  
    WeekDaySearchQur.Active:=true;  
    WeekDay:=  
        WeekDaySearchQur.Fields.FieldName('Kod_WeekDay').AsInteger;  
    WeekDaySearchQur.Active:=false;  
  
    ParaSearchQur.Params[0].AsTime:= StrToTime(ParaSearchDBC.Text);  
    ParaSearchQur.Active:=true;  
    Para:=ParaSearchQur.Fields.FieldName('Kod_Para').AsInteger;  
    ParaSearchQur.Active:=false;  
  
    SearchQur.Close;  
    SearchQur.SQL.Clear;  
    SearchQur.SQL.LoadFromFile('Search_All.sql');  
    SearchQur.Params[0].AsInteger:=Para;  
    SearchQur.Params[1].AsString:=Group;  
    SearchQur.Params[2].AsInteger:=WeekDay;  
    SearchQur.Params[3].AsInteger:=Teacher;  
    SearchQur.Open;  
end;
```

***Эксперимент.*** Запустите приложение. Определите, какое занятие будет у Вас первой парой в понедельник. ♦

**Задания для самостоятельного выполнения:**

**11.5.** Напишите приложение, выполняющее SQL-запрос,

- а) считанный из файла (используйте диалоговое окно открытия файла);
- б) созданный во время работы приложения, например, в компоненте Мемо.

Предусмотрите возможность обработки исключительной ситуации.

**11.6.** «Универсальный построитель SQL-выражений», который позволяет генерировать выражения SQL: предоставляет возможность выбирать таблицы и названия их полей, которые необходимо отобразить, а также условие отбора.

**Примечание.** Используйте динамическое создание компонентов.

**11.7.** «Личная библиотека». Картотека домашней библиотеки: выходные данные книги (авторы, название, издательство и т. д.), раздел библиотеки (специальная литература, хобби, домашнее хозяйство и т. д.), происхождение и наличие книги в данный момент, субъективная оценка книги. Выбор книги по произвольному запросу, инвентаризация библиотеки.

**11.8.** «Картотека Интерпола». Данные по каждому зарегистрированному: фамилия, имя, кличка, рост, цвет волос и глаз, особые приметы, гражданство, место и дата рождения, последнее место жительства, знание языков, преступная профессия, последнее дело и т. д. Преступные и мафиозные группировки (данные о подельниках). Перенос «завязавших» в архив, удаление — после смерти.

**11.9.** «Биржа труда». База безработных: паспортные данные, профессия, образование, место последней работы и должность, причина увольнения, семейное положение, жилищные условия, контактные координаты, требования к будущей работе. База вакансий: фирма, должность, условия труда и оплаты, жилищные условия, требования к специалисту. Поиск и регистрация вариантов с той и другой стороны, формирование объявлений для печати, удаление в архив после трудоустройства, полное удаление при отказе от услуг.

- 11.10.** «Записная книжка». Фамилия, имя, отчество, адрес, телефон, место работы или учебы, должность, характер знакомства, деловые качества и т. д. Автоматическое поздравление с днем рождения (по текущей дате). Упорядочение по алфавиту, по дате последней корректировки. Поиск по произвольному шаблону.
- 11.11.** «Касса аэрофлота». Расписание: номер рейса, маршрут, пункты промежуточных посадок, время отправления, дни полета. Количество свободных мест на каждом рейсе. Выбор ближайшего рейса до заданного пункта (при наличии свободных мест), оформление заданного числа билетов по согласованию с пассажиром (с уменьшением числа свободных мест), оформление посадочной ведомости.
- 11.12.** «Магазин». Компьютер вместо кассового аппарата. База наличия товаров: наименование, единица измерения, цена единицы, количество, дата последнего завоза. Регистрация поступления товара (как старых, так и новых наименований). Оформление покупки: выписка чека, корректировка БД. Проблема уценки и списания. Инвентаризация остатков товара с вычислением суммарной стоимости.
- 11.13.** «Отдел кадров». БД о сотрудниках фирмы: паспортные данные, образование, специальность, подразделение, должность, оклад, даты поступления в фирму и последнего назначения. Выбор по произвольному шаблону. Сокращение штатов: выбор для увольнения лиц пенсионного и предпенсионного возраста, подготовка приказа.
- 11.14.** «Администратор гостиницы». Список номеров: класс, число мест. Список гостей: паспортные данные, даты приезда и отъезда, номер. Поселение гостей: выбор подходящего номера (при наличии свободных мест), регистрация, оформление квитанции. Отъезд: выбор всех постояльцев, отъезжающих сегодня, освобождение места или оформление задержки с выпиской дополнительной квитанции. Возможность досрочного отъезда с перерасчетом. Поиск гостя по произвольному признаку.
- 11.15.** «Справочник меломана». База групп и исполнителей; база песен; база дисков с перечнем песен (в виде ссылок). Выбор всех песен заданной группы; всех дисков, где встречается заданная песня.

- 11.16.** «Справочник работников ГИБДД». Марка, цвет, заводской и бортовой номера, дата выпуска, особенности конструкции и окраски, дата последнего техосмотра транспортного средства (автомобиля, мотоцикла, прицепа и т. д.), паспортные данные владельца. Выбор транспортных средств по произвольному шаблону. Формирование приглашений на техосмотр в соответствии со сроком.
- 11.17.** «Справочник владельца видеотеки». База видеофильмов: название, студия, жанр, год выпуска, режиссер, исполнители главных ролей, краткое содержание, субъективная оценка фильма. Факт наличия фильма в видеотеке. Оформление выдачи и возврата кассеты.
- 11.18.** В MS Excel разработайте базу намечаемых мероприятий, включающую поля: дата, время и протяженность, место проведения. Используя компонент Database, разработайте приложение, обеспечивающее доступ к базе данных и выполняющее автоматическое напоминание ближайшего дела: по текущей дате и времени; удаление вчерашних дел либо перенос на будущее. Кроме того, приложение должно обеспечивать анализ «коллизий» (пересечений) планируемых дел, а также просмотр дел на завтра, послезавтра и т. д.
- 11.19.** В MS Access разработайте базу данных, в которой хранится информация о преподавателях и читаемых ими курсах, сведения о студентах и посещаемых курсах.

Установлено, что

- ☐ студент может посещать любое количество курсов;
- ☐ преподаватели могут вести несколько курсов;
- ☐ курс читается в некоторой аудитории;
- ☐ в одной аудитории читается только один курс.

Таким образом, выделяются следующие таблицы: *Преподаватели*, *Студенты*, *Курсы*, *Аудитории*, *Регистрация*. Каждый объект должен содержать следующую информацию.

**Преподаватели:**

- ☐ фамилия, имя, отчество;
- ☐ дата рождения — не может быть больше сегодняшней даты и меньше 1900 года;
- ☐ домашний адрес;
- ☐ телефон — задается в формате 00-00-00,

- ☐ дата найма — не может быть больше даты сегодняшнего дня;
- ☐ стаж — вычисляемое поле, определяется сегодняшней датой и датой найма.

***Студенты:***

- ☐ фамилия, имя, отчество;
- ☐ дата рождения — не может быть больше даты сегодняшней даты и меньше 1900 года;

***Курсы:***

- ☐ наименование курса;
- ☐ фамилия преподавателя, читающего курс, список посещающих курс студентов, аудитория — внешний ключ.

***Аудитории:***

- ☐ номер;
- ☐ тип аудитории.

***Регистрация:***

- ☐ код курса;
- ☐ код студента.

Разработайте приложение, которое для доступа к базе данных использует компонент Database и отображает на форме информацию о студентах, посещающих выбранный курс у заданного преподавателя.

**Вопросы для повторения**

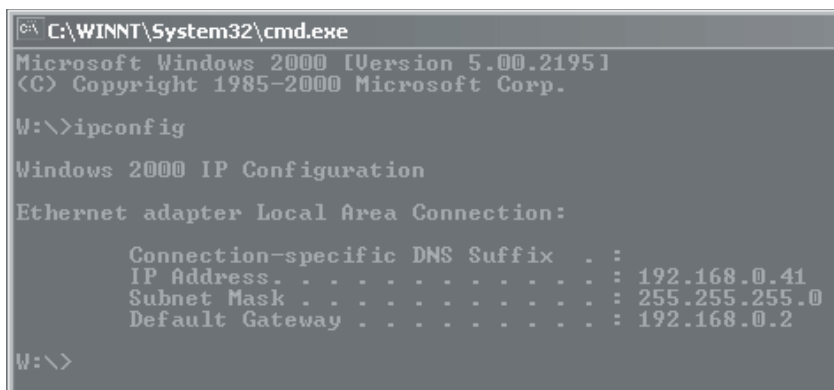
1. В чем состоит суть принципа открытой архитектуры доступа к базам данных?
2. С какими форматами баз данных позволяет работать BDE?
3. Какие существуют типы компонентов для разработки приложений баз данных? Поясните роль каждого типа.
4. Что такое вычисляемые поля? Каким образом они создаются в Delphi?
5. Для чего предназначен редактор полей (Field Editor)?

## Программирование сокетов

Сокеты позволяют организовать обмен информацией между компьютерами. Компоненты сокетов Delphi основаны на протоколе TCP/IP и низкоуровневых сокетах Windows.

Протокол — это набор правил, определяющий взаимодействие двух устройств. TCP/IP — это комбинация двух протоколов Transmission Control Protocol/Internet Protocol, совместная работа которых обеспечивает соединение в Интернете. IP отвечает за определение и маршрутизацию дейтаграмм (блок данных, передающийся через Интернет), а также задает схему адресации. TCP отвечает за надежную доставку данных на транспортном уровне служб.

Каждое устройство в сети имеет IP-адрес, представляющий собой 32-разрядное число. Для того чтобы узнать IP-адрес компьютера из командной строки (команда Пуск/Выполнить/cmd), запустите программу ipconfig (рис. 12.1). Строка IP Address отображает IP-адрес компьютера. По этому адресу происходит обмен информацией с другими компьютерами. При тестировании сетевых приложений можно использовать так называемый «возвратный адрес» — 127.0.0.1., поскольку отправленные по этому адресу данные «возвращаются» на тот же самый компьютер.



```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

W:\>ipconfig

Windows 2000 IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix . . . : 
    IP Address. . . . . : 192.168.0.41
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.0.2

W:\>
```

Рис. 12.1

Каждый компьютер имеет имя, по которому можно обратиться к компьютеру (в этом случае имя компьютера будет автоматически преобразовано в IP-адрес). Имя компьютера можно посмотреть в свойствах компьютера на странице Сетевая идентификация (рис. 12.2).

Каждое TCP-соединение осуществляется через порт. Порт представлен в виде 16-разрядного числа. Пара из IP-адреса и TCP-порта полностью определяет Интернет-соединение или, используя более точный термин, сокет. В файле services определены порты, стандартно используемые определенными протоколами и службами. Для сокетов на стороне сервера номер TCP-порта служит идентификационным номером соединения, на котором этот сокет сервера будет ожидать соединения от клиентов. На стороне клиентского сокета номер порта используется как указатель на требуемое серверное соединение. Номер порта обычно применяется для определения службы, которую клиент хочет использовать на серверном приложении.

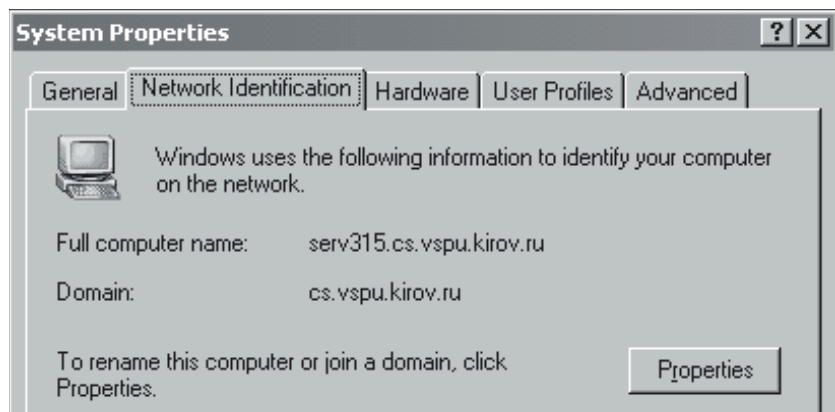


Рис. 12.2

На странице Internet (рис. 12.3) библиотеки компонентов Delphi расположены компоненты ClientSocket и ServerSocket для реализации сетевого взаимодействия через сокеты.



Рис. 12.3

## 12.1. Компонент ClientSocket

Класс TClientSocket описывает сокет клиента.

### Свойства:

Active: boolean	значение True показывает, что сокет открыт, False — закрыт;
Host: string	хост-имя компьютера, к которому следует подключиться;
Address: string	IP-адрес компьютера, к которому следует подключиться. В отличие от Host, здесь может содержаться лишь IP. Отличие в том, что если вы укажете в Host символьное имя компьютера, то IP адрес, соответствующий этому имени, будет запрошен у DNS;
Service: string	служба (ftp, http, pop и т. д.), к порту которой произойдет подключение;
Port: integer	номер порта (от 1 до 65535) для установления соединения. Обычно номера портов берутся начиная с 1001, так как номера меньше 1000 могут быть заняты системными службами (например, порт 21 зарезервирован для FTP); <b>Примечание.</b> Описание зарезервированных портов системы описано в файле: %SystemRoot%\system32\drivers\etc\services
ClientType:TClientType; TClientType = (ctNonBlocking, ctBlocking);	тип соединения: ctNonBlocking — асинхронная передача данных, т. е. посылать и принимать данные по сокету можно одновременно с помощью OnRead и OnWrite; ctBlocking — синхронная передача данных, т. е. события OnRead и OnWrite не работают (этот тип соединения полезен для организации обмена данными с помощью потоков).

## Методы:

Open	открытие сокета (аналогично присвоению значения True свойству Active);
Close	заккрытие сокета (аналогично присвоению значения False свойству Active).

## События

OnConnect: TSocketNotifyEvent;

где

```
TSocketNotifyEvent = procedure (Sender: TObject; Socket: TCustomWinSocket) of object;
```

Это событие возникает при установлении соединения, в обработчике этого события можно начинать авторизацию или прием/передачу данных.

OnConnecting: TSocketNotifyEvent;

Это событие возникает при установлении соединения. Отличие от события OnConnect состоит в том, что соединение еще не установлено. Обычно такие промежуточные события используются для обновления статуса.

OnDisconnect: TSocketNotifyEvent;

Это событие возникает при закрытии сокета. Закрытие может произойти вследствие завершения работы приложения, может быть вызвано со стороны удаленного компьютера либо из-за сбоя сети.

OnError: TSocketErrorEvent;

где

```
TSocketErrorEvent = procedure (Sender: TObject;  
    Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;  
    var ErrorCode: Integer) of object;
```

Это событие возникает в результате ошибки в работе сокета.

**Примечание.** Заметим, что обработчик этого события будет бесполезен при обработке ошибок в момент открытия сокета (метод Open). Для того чтобы избежать появления стандартного Windows-сообщения об ошибке, необходимо операторы открытия сокета заключить в блок try...except.

Параметр `ErrorCode` определяет тип ошибки, может принимать одно из следующих значений:

Константа	значение
<code>eeSend</code>	ошибка записи данных в сокетное соединение;
<code>eeReceive</code>	ошибка чтения данных из сокетного соединения;
<code>eeConnect</code>	принятый запрос соединения не завершен;
<code>eeDisconnect</code>	ошибка закрытия сокета;
<code>eeAccept</code>	попытка принять запрос клиентского приложения;
<code>eeGeneral</code>	сокет получил ошибочное сообщение, которое не соответствует ни одному из перечисленных выше категорий.

`OnLookup: TSocketNotifyEvent;`

возникает при попытке определения серверного сокета, с которым должно осуществиться соединение.

`OnRead: TSocketNotifyEvent;`

возникает, когда удаленный компьютер отправил какие-либо данные. При возникновении этого события возможна обработка данных.

`OnWrite: TSocketNotifyEvent;`

возникает, когда клиентский сокет пишет данные в сокетное соединение.

**Упражнение 12.1.1.** Напишите приложение, проверяющее открытые порты компьютера.

### Решение

Создайте каталог `Sniffer`, файл модуля сохраните под именем `Main.pas`, файл проекта — `Sniffer.dpr`.

#### *1-й этап. Визуальное проектирование*

Расположите компоненты в соответствии с изображением на рис. 12.1.1.

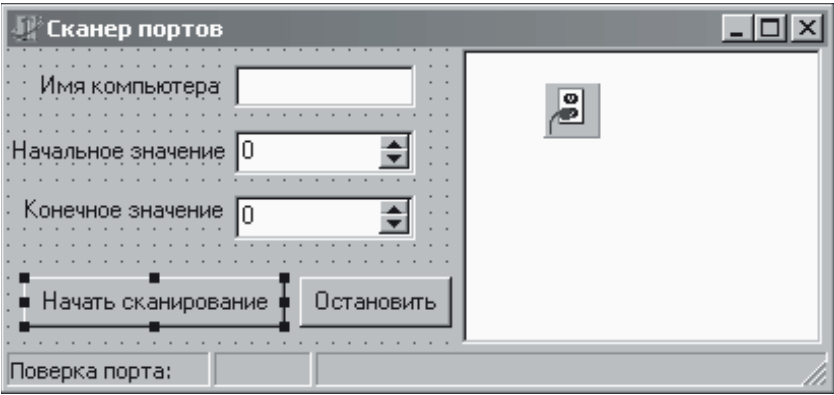


Рис. 12.1.1

Измените значения свойств следующим образом:

компонент	свойство	значение
Form1	Name	SnifferFrm
	Caption	Сканер портов
Label1	Caption	Имя компьютера
ClientSocket1	Name	ClientSocket
Label2	Caption	Начальное значение
Label3	Caption	Конечное значение
	Text	
SpinEdit1	Name	BeginSE
	Value	0
SpinEdit2	Name	EndSE
	Value	0
Button1	Name	BeginBtn
	Caption	Начать сканирование
Button2	Name	EndBtn
	Caption	Остановить
ListBox1	Name	PostList
StatusBar1.Panels[0]	Text	Проверка порта
StatusBar1.Panels[1]	Alignment	taRightJustify

Сохраните приложение.

### *2-й этап. Создание программного кода*

В разделе **Private** класса **TSnifferFrm** опишите переменную **port\_number** типа **integer**, предназначенную для хранения значения сканируемого порта.

Создайте обработчик события **OnClick** кнопки **BeginBtn**:

```
procedure TSnifferFrm.BeginBtnClick(Sender: TObject);
begin
  if HostEdt.Text<>' ' then    {если имя компьютера определено, то}
begin
    ClientSocket.Host := HostEdt.Text;
    {свойству Host клиентского сокета присваиваем это значение}
    port_number := BeginSE.Value;
    {устанавливаем начальное значение порта}
    Scanner;           {процедура обработки текущего значения порта}
  end;
end;
```

Опишем алгоритм, который определяет возможность работы с текущим значением порта:

```
procedure TSnifferFrm.Scanner;
begin
  if port_number <= EndSE.Value then
    {если текущее значение порта не превосходит конечного значения}
  with ClientSocket do
begin
    Active := False;           {закрываем текущее соединение}
    Port := port_number;       {устанавливаем новое значение порта}
    Active := True;            {открываем новое соединение}
  end;
end;
```

Попытка установления соединения генерирует событие **OnLookup** компонента **ClientSocket**:

```
procedure TSnifferFrm.ClientSocket1Lookup(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  StatusBar1.Panels[1].Text := IntToStr(port_number);
  {в строке состояния отобразим значение сканируемого порта}
end;
```

**Эксперимент.** Запустите приложение. При попытке установить соединение сокета со значением закрытого порта приложение генерирует исключительную ситуацию (рис. 12.1.2). ♦

Чтобы предотвратить сбой в работе программы, создайте обработчик события `OnError` компонента `ClientSocket1`:

```
procedure TSnifferFrm.ClientSocketError(Sender: TObject;  
    Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;  
    var ErrorCode: Integer);  
begin  
    ErrorCode:=0; {игнорируем произошедшую ошибку}  
    Inc(port_number);  
        {увеличиваем значение тестируемого порта на единицу}  
    Scanner; {процедура обработки текущего значения порта}  
end;
```

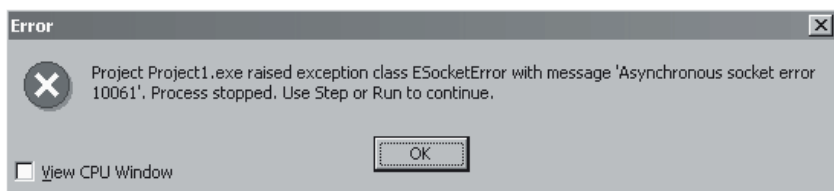


Рис. 12.1.2

**Эксперимент.** Запустите приложение. Убедитесь, что соединение сокета не прекращает работу приложения. ♦

В случае удачного установления соединения возникает событие `OnConnect` компонента `ClientSocket`:

```
procedure TSnifferFrm.ClientSocketConnect(Sender: TObject;  
    Socket: TCustomWinSocket);  
begin  
    PortsList.Items.Add(inttostr(Socket.RemotePort));  
        {добавляем значение открытого порта в список PortsList}  
    ClientSocket.Active := False; {закрываем соединение}  
    inc(port_number); {устанавливаем новое значение порта}  
    Scanner; {процедура обработки текущего значения порта}  
end;
```

**Эксперимент.** Запустите приложение. Определите, какие порты открыты на Вашем компьютере. ♦

Для завершения сканирования портов создайте обработчик события `OnClick` кнопки `Остановить`:

```

procedure TSnifferFrm.EndButtonClick(Sender: TObject);
begin
    port_number := EndSE.value+1;
    {сканирование порта осуществляет процедура Scanner. Обращение
     к ней осуществляется при возникновении трех событий: OnClick
     кнопки Начать сканирование; OnConnect и OnError клиентского
     сокета. События OnConnect и OnError могут возникнуть только при
     открытии сокетного соединения, которое осуществляется в процедуре
     Scanner (если значение port_number меньше либо равно EndSE.Value).
     Поэтому установка значения переменной port_number равным
     EndSE.value+1 прекращает работу приложения.}
end;

```

**Эксперимент.** Запустите приложение. Убедитесь в работоспособности приложения. Попробуйте ввести имя компьютера, соединение с которым невозможно. Что произошло? ♦

Для избежания ошибок, возникающих при установлении соединения с недоступным компьютером, внесите следующие изменения в код приложения:

```

procedure TSnifferFrm.BeginBtnClick(Sender: TObject);
    var Addr : TInAddr;           {тип TInAddr описан в модуле Winsock}
begin
    if Edit1.text<>' ' then
    begin
        Addr := ClientSocket.Socket.LookupName(HostEdt.Text);
        {генерирует адрес указанного в параметре компьютера}
        if Addr.S_addr = 0 then
        begin
            {если значение равно нулю, то в сети нет компьютера
             с указанным в параметре именем}
            MessageDlg('Сервер ' + HostEdt.Text +
                ' не существует!', mtError, [mbOk], 0);
            StatusBar1.SimpleText := 'Введите имя сервера';
            HostEdt.SetFocus;
        end
        else
        begin
            ClientSocket.Host := HostEdt.Text;
            port_number := BeginSE.Value;
            Scanner;
        end
    end;
end;

```

**Эксперимент.** Запустите приложение. Убедитесь, что при неправильном введении имени компьютера приложение остается работоспособным. ♦

## Задания для самостоятельного выполнения

**12.1.** Используя справочную систему Delphi, изучите класс `TCustomWinSocket`. Напишите приложение, которое после установления соединения позволяет отправлять и получать сообщения.

**Примечание.** Для отправки сообщения воспользуйтесь методом `Sendtext` объектного свойства `Socket`:

```
ClientSocket1.Socket.SendText(<текстовое сообщение>);
```

Событие `OnRead`, определенное в классе `TClientSocket`, принимает посланные данные:

```
s := Socket.ReceiveText;                                {s : string}
```

## 12.2. Компонент TServerSocket

Сервер, основанный на сокетном протоколе, позволяет обслуживать сразу множество клиентов. Для каждого подключенного клиента сервер открывает отдельный сокет, по которому происходит обмен данными с клиентом, кроме этого для каждого подключения создается отдельный поток. Для реализации сервера используется класс `TServerSocket`.

### Свойства

<code>Socket: TServerWinSocket</code>	<code>TServerWinSocket</code> описывает текущие активные сокетные соединения, а также информацию о потоках, которые кэшируются для использования серверным сокетом;
<code>ServerType: TServerType</code>	тип сервера. Может принимать одно из двух значений: <code>stNonBlocking</code> — синхронная работа с клиентскими сокетами (при таком типе сервера возможна работа с клиентами через события <code>OnClientRead</code> и <code>OnClientWrite</code> ); <code>stThreadBlocking</code> — асинхронный тип (для каждого клиентского сокетного канала создается отдельный процесс — <code>Thread</code> );

<code>ThreadCacheSize: Integer</code>	количество клиентских процессов (Thread), которые будут кэшироваться (постоянно находиться в памяти) сервером;
<code>Active: Boolean</code>	значение True указывает, что сервер открыт и готов к приему клиентских соединений;
<code>Port: Integer</code>	номер порта для установления соединений с клиентами. Значения этого свойства у сервера и у клиентов должны быть одинаковыми;
<code>Service: string</code>	строка, определяющая службу (ftp, http, pop и т. д.), порт которой будет использован, — это своеобразный справочник соответствия номеров портов различным стандартным протоколам.

## Методы

<code>Open</code>	запускает сервер, выполнение этого метода идентично присвоению значения True свойству <code>Active</code> ;
<code>Close</code>	останавливает сервер, выполнение метода идентично присвоению значения False свойству <code>Active</code> .

## События

<code>OnClientConnect:</code> <code>TSocketNotifyEvent</code>	возникает, когда клиент установил сокетное соединение и ждет ответа сервера ( <code>OnAccept</code> );
<code>OnClientDisconnect:</code> <code>TSocketNotifyEvent</code>	отсоединение клиента от сокетного канала;
<code>OnClientError:</code> <code>TSocketErrorEvent</code>	неудачное завершение текущей операции;
<code>OnClientRead:</code> <code>TSocketNotifyEvent</code> ;	клиент передал серверу данные. Параметр <code>Socket: TCustomWinSocket</code> обеспечивает доступ к полученным сервером данным;
<code>OnClientWrite:</code> <code>TSocketNotifyEvent</code>	сервер может отправить данные клиенту;

OnGetSocket:	TGetSocketEvent = procedure
TGetSocketEvent	(Sender: TObject; Socket: TSocket; var ClientSocket: TServerClientWinSocket) of object; <b>серверному сокету необходимо создать новый объект типа TServerClientWinSocket для отображения серверного соединения для клиентского сокета;</b>
OnGetThread:	TGetThreadEvent = procedure
TGetThreadEvent	(Sender: TObject; ClientSocket: TServerClientWinSocket; var SocketThread: TServerClientThread) of object; <b>в обработчике этого события дает возможность определить уникальный процесс (Thread) для каждого клиентского соединения, присвоив параметру Socket Thread соответствующую подзадачу;</b>
OnThreadStart:	TThreadNotifyEvent = procedure
TThreadNotifyEvent	(Sender: TObject; Thread: TServerClientThread) of object <b>возникает, когда процесс клиентского сокетного соединения запускается;</b>
OnThreadEnd:	в обработчике события необходимо
TThreadNotifyEvent	выполнить действия, связанные с завершением клиентского соединения;
OnAccept:	возникает в момент присоединения
TSocketNotifyEvent	клиентского сокета;
OnListen:	возникает, когда сервер переходит
TSocketNotifyEvent	в режим ожидания подсоединения клиентов.

При работе с сокетами в основном используются события OnClientRead и OnClientWrite, параметр ClientSocket которых задает сокет клиента. Свойство Socket класса TServerSocket наследует от TServerWinSocket следующие свойства, характеризующие серверный сокет:

ActiveConnections: Integer	количество подключенных клиентов;
ActiveThreads: Integer	количество работающих потоков;
property Connections[Index: Integer]: TCustomWinSocket;	массив, состоящий из отдельных классов TCustomWinSocket для каждого подключенного клиента. Например, команда: ServerSocket1.Socket.Connections[0].SendText('Hello!'); отсылает первому подключенному клиенту сообщение 'Hello!'. Для работы с элементами этого массива используются методы SendText, ReceiveText, Send Buffer, ReceiveBuffer, Send Stream, ReceiveBuffer;
IdleThreads: Integer	количество свободных потоков, эти потоки кэшируются сервером;

## Задания для самостоятельного выполнения

### 12.2. Исследуйте, в какой последовательности происходят события на сервере

**Примечание.** Создайте серверное приложение, положите на форму компонент ServerSocket и установите значения свойств (Port и Active), компонент ListBox будет служить для отображения событий, происходящих с компонентом ServerSocket. Создайте обработчики событий этого компонента (TServerSocket), в каждом из которых напишите один оператор — добавление в ListBox названия произошедшего события.

Для тестирования событий серверного сокета воспользуйтесь приложением, созданным в задании 12.1. (значение свойства порт компонента ClientSocket должно совпадать со значением свойства Port, установленным у ServerSocket).

### 12.3. Посылка/прием сложных данных

Иногда необходимо пересылать/получать по сети не только простые текстовые сообщения, но и сложные структуры (типа record в Паскале) или даже файлы. Для этого можно воспользо-

ваться следующими методами, описанными в классе `TCustomWinSocket`.

```
function SendBuf(var Buf; Count: Integer): integer;
```

посылает содержимое буфера через сокет.

Буфером может являться любой тип. Буфер указывается параметром `Buf`, второй параметр (`Count`) должен содержать размер пересылаемых данных в байтах.

```
function ReceiveBuf(var Buf; Count: Integer): Integer;
```

В переменную `Buf` из сокетного соединения считывает данные, объем которых определен параметром `Count`. Результат — количество полученных данных (в байтах), или `-1`, если данные не были получены.

**Упражнение 12.3.1.** Создайте приложение `Chat` для локальной сети. `Chat` — это программа, предназначенная для диалога между людьми, находящимися в различных местах сети.

### Решение

Приложение `Chat` является клиент-серверным приложением, т. е. предполагает наличие выделенного сервера и клиентских программ. Если пользователь захочет воспользоваться услугами `Chat`, то он запускает клиентскую часть программы и подключается к серверу. Взаимодействие клиентов происходит только через сервер (рис. 12.3.1).

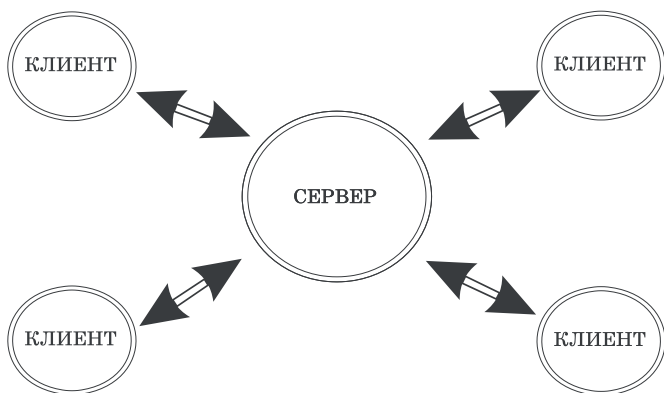


Рис. 12.3.1

При написании приложения воспользуемся возможностью создания группы проектов. Группа проектов — это специальное средство, предоставляемое Delphi для написания и отладки программ, которые состоят из нескольких проектов.

Приложение «Chat» состоит из двух приложений: сервер и клиент; объединим их в одну группу.

Создайте новое приложение, выполнив команду главного меню File/ New.../ New Application. Создайте каталог Chat. Сохраните проект под именем Server.dpr, а главный модуль — под именем ServerUnit.pas. Это приложение будет сервером. Измените имя формы на значение ServerForm.

Выполните команду View/ Project Manager. Для диалогового окна Project Manager (рис. 12.3.2.) вызовите контекстное меню, сохраните группу проектов под именем ChatGroup, выполнив команду Save Project Group.

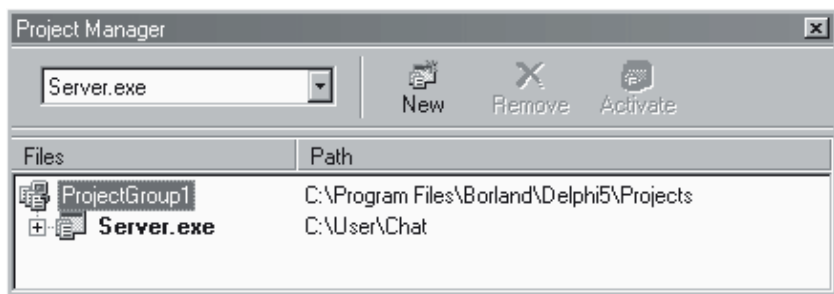


Рис. 12.3.2

Добавьте в группу проектов новое приложение. Для этого щелкните на кнопке New диалогового окна (рис. 12.3.2). Сохраните проект под именем Client.dpr, модуль — ClientUnit.pas. Свойство Name формы установите в ClientForm.

Переключаться между проектами можно либо используя окно Менеджера проектов, либо выбирая проект из подменю кнопки Run.

### ***Серверное приложение***

Используя Project Manager, сделайте активным серверное приложение.

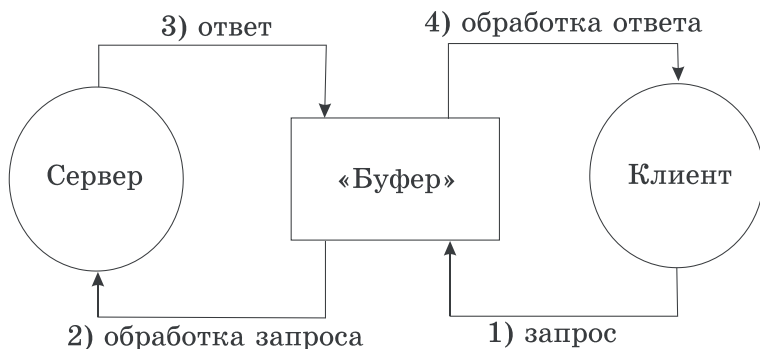
*1-й этап. Визуальное проектирование*

Положите на форму компонента ServerSocket.

Name	ServerSocket
Port	1024
Active	True

Итак, сервер будет находиться в режиме ожидания клиентов по порту 1024. Этот порт уникально идентифицирует создаваемую службу чата, а при подключении пользователей сервер будет обрабатывать все запросы.

Опишем схему процесса взаимодействия клиента и сервера (рис. 12.3.3)



**Рис. 12.3.3**

Взаимодействие клиента и сервера будет осуществляться посредством некоего «буфера». Клиент делает запрос, записывая данные в «буфер». Сообщение о записи данных в «буфер» приходит серверу, он, считывая и обрабатывая их, записывает ответ в «буфер». Новые данные из «буфера» обрабатываются клиентом.

Договоримся запрос и ответ называть сообщениями. Необходимо стандартизировать сообщения, чтобы клиент и сервер смогли понять друг друга. Выделим 3 вида сообщений:

- ☐ установление соединения с пользователем;
- ☐ получение текстовой информации от пользователя (части диалога между пользователями);
- ☐ разрыв соединения с пользователем.

Каждому сообщению сопоставим уникальный номер.

Создайте новый модуль (File/ New.../ Unit), сохраните его под именем ChatMsgs.pas. В разделе interface модуля опишите константы, идентифицирующие номера сообщений:

**const**

```
MSG_USERCONNECT = 1;
MSG_USERSENDTEXT = 2;
MSG_USERDISCONNECT = 3;
```

Структура каждого сообщения различна и может измениться с новой версией программы. Опишем каждое сообщение в виде структур следующего вида:

**type**

```
TUserConnectMsg = record
  MsgNumber: Integer;
  UserName: ShortString;
```

**end;**

```
PUserConnectMsg = ^TUserConnectMsg;
```

**Примечание.** При добавлении таких возможностей, как введение пароля при подключении, выбор цвета отображения текста и т. д., структура описания сообщения может быть расширена.

```
TUserSendTextMsg = record
```

```
  MsgNumber: Integer;
```

```
  Text: ShortString;
```

**end;**

```
PUserSendTextMsg = ^TUserSendTextMsg;
```

```
TUserDisconnectMsg = record
```

```
  MsgNumber: Integer;
```

```
  UserName: ShortString;
```

**end;**

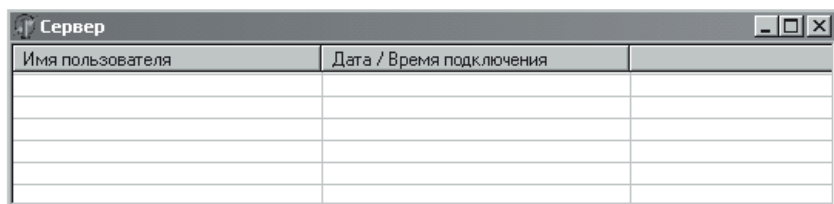
```
PUserDisconnectMsg = ^TUserDisconnectMsg;
```

Сохраните изменения, внесенные во вспомогательный модуль.

Вернемся к проектированию сервера. Для отображения информации о состоянии чата положите на форму компонент List-View (страница Win32).

Name	List
Align	Client
Active	True
FlatScrollBars	True
GridLines	True
RowSelect	True
ReadOnly	True
ViewStyle	vsReport (для отображения колонками)

Компонент `ListView` будет отображать информацию обо всех подключившихся пользователях: Имя пользователя, Дата/Время подключения (рис. 12.3.4). Для создания колонок с соответствующими заголовками выполните команду `Columns Editor...` контекстного меню компонента.



**Рис. 12.3.4**

### *2-й этап. Разработка программного кода*

Для обработки сообщений, приходящих от клиентских приложений, создайте обработчик события `OnClientRead` компонента `ServerSocket`:

```
procedure TServerForm.ServerSocketClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
var Msg: Pointer;
    Size: integer;
begin
    Size:=Socket.ReceiveLength;
    GetMem(Msg, Size);
    {выделяется буфер под данные, посланные пользователем}
    Socket.ReceiveBuf(Msg^, Size);
    HandleMessage(Socket, Msg, Size);
    {процедура обработки принятых данных}
    FreeMem(Msg);
end;
```

Процедура `HandleMessage` в зависимости от номера, идентифицирующего тип сообщения, вызывает тот или иной обработчик:

```
procedure TServerForm.HandleMessage(Socket:
    TCustomWinSocket; Msg: Pointer; Size: integer);
begin
    case Integer(Msg^) of
        MSG_USERCONNECT: ConnectUser(Socket, Msg, Size);
        MSG_USERSENDTEXT: SendText(Msg, Size);
        MSG_USERDISCONNECT: DisconnectUser(Socket, Msg, Size);
    end;
end;
```

Процедура *ConnectUser* вызывается в момент установления соединения с пользователем, при этом необходимо всем участникам разговора сообщить о появлении нового участника, а пользователю, который только что присоединился, послать список всех подключенных к серверу:

```
procedure TServerForm.ConnectUser(Socket: TCustomWinSocket;
  Msg: PUserConnectMsg; Size: integer);
var i: integer;
      M: TUserConnectMsg;
begin
  M.MsgNumber:=MSG_USERCONNECT;
  for i:=0 to ServerSocket.Socket.ActiveConnections - 2 do
begin      {ActiveConnection — общее количество присоединений,
                                                    отсчет начинается с нуля}
    ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);
    M.UserName:=List.Items[i].Caption;
    Socket.SendBuf(M, SizeOf(TUserConnectMsg));
  end;
  with List.Items.Add do {добавляем в список нового пользователя}
begin
    Caption:=Msg.UserName;
    SubItems.Add(DateTimeToStr(Now));
    Data:=Socket;      {в поле Data будем хранить указатель на сокет}
  end;
end;
```

Процедура *SendText* выполняется, когда пользователь посылает текст:

```
procedure TServerForm.SendText(Msg: PUserSendTextMsg;
  Size: integer);
var i: integer;
begin
  for i:=0 to ServerSocket.Socket.ActiveConnections - 1 do
    ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);
    {передача сообщения всем подключившимся пользователям}
end;
```

Процедура *DisconnectUser* обрабатывает отключение пользователя:

```
procedure TServerForm.DisconnectUser(Socket:
  TCustomWinSocket;
  Msg: PUserDisconnectMsg; Size: integer);
var i, Index: integer;
begin
  for i:=0 to ServerSocket.Socket.ActiveConnections - 1 do
```

```
ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);  
    {широковещательное уведомление всех пользователей  
    об отключении одного из них}  
Index:=IndexOf(Msg.UserName);  
List.Items.Delete(Index);  
    {удаление из списка пользователя, разорвавшего соединение}  
end;
```

Опишем функцию поиска индекса элемента списка по его имени:

```
function TServerForm.IndexOf(Caption: string): integer;  
begin  
    for Result:=0 to List.Items.Count - 1 do  
        if AnsiSameText(Caption, List.Items[Result].Caption) then  
            Exit;  
    Result:=-1;  
end;
```

Сохраните приложение.

### ***Клиентское приложение***

Используя Project Manager, активируйте проект клиентского приложения.

#### *1-й этап. Визуальное проектирование*

Измените свойство Caption формы на значение Клиент.

Положите на форму компонент ClientSocket. Установите следующие значения его свойств:

Name	ClientSocket
Port	1024

Компонент RichEdit (страница Win32) будет использоваться для отображения сообщений. Расположите его так, как показано на рис. 12.3.5, измените значения свойств на следующие:

Name	RichEdit
ReadOnly	True

Для отображения подключенных к серверу клиентов положите на форму компонент ListBox (страница Standart). Измените имя на значение List.

Компонент Edit служит для ввода текстового сообщения. Установите свойство Name в значение Edit, а MaxLength — 255 (максимальная длина передаваемого сообщения).

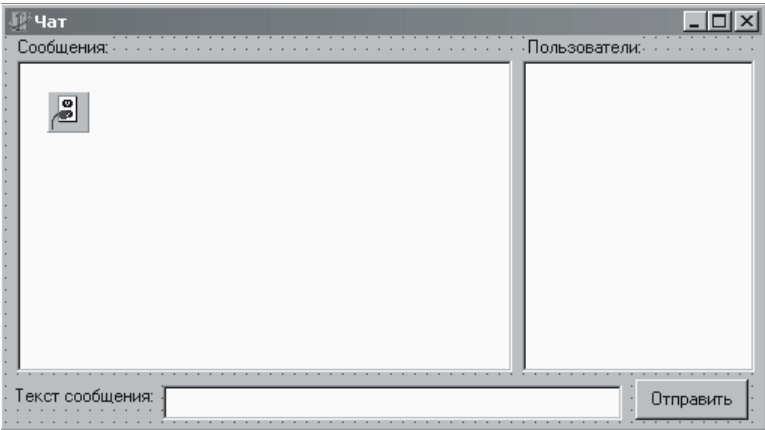


Рис. 12.3.5

После щелчка на компоненте Button текстовое сообщение, введенное в Edit, будет передано всем участникам чата. Измените значения свойств Button1 на следующие:

Name	Button
Caption	Отправить

Положите на форму компоненты Label, расположите их так, как показано на рисунке, и установите соответствующие значения свойства Caption.

*Эксперимент.* Попробуйте изменить размер формы. Как можно заметить, компоненты остаются на месте, никак не реагируя на происходящее.

Воспользуемся свойством Anchors, которое есть у всех визуальных компонентов. Оно заставляет стороны компонента быть привязанными к сторонам родительского компонента, т.е. при изменении размера родительского компонента изменяются размеры и дочернего компонента. Измените свойство Anchors компонентов следующим образом:

Имя компонента	akLeft	akTop	akRight	akBottom
Label1	True	True	False	False
Label2	False	True	True	False
RichEdit	True	True	True	True
ListBox	False	True	True	True

Label3	True	False	False	True
Edit	True	False	True	True
Button	False	False	True	True

Попробуйте сейчас изменить размеры формы. Размеры компонентов изменяются пропорционально изменению размеров формы. ◆

### *2-й этап. Создание программного кода*

Обработчик события OnRead компонента ClientSocket аналогичен соответствующему обработчику события OnRead компонента ServerSocket:

```
procedure TClientForm.ClientSocketRead(Sender: TObject;
  Socket: TCustomWinSocket);
var Msg: Pointer;
    Size: Integer;
begin
  Size := Socket.ReceiveLength;
  GetMem(Msg, Size);
  Socket.ReceiveBuf(Msg^, Size);
  HandleMessage(Msg);
  FreeMem(Msg);
end;
```

Соответственно, процедура выбора конкретного обработчика по номеру сообщения будет похожей:

```
procedure TClientForm.HandleMessage(Msg: Pointer);
begin
  case Integer(Msg^) of
    MSG_USERCONNECT: ConnectUser(Msg);
    MSG_USERSENDTEXT: SendText(Msg);
    MSG_USERDISCONNECT: DisconnectUser(Msg);
  end;
end;
```

Обратите внимание, что из списка параметров исключены параметры Socket и Size. Опишем используемые обработчики событий.

Процедура ConnectUser вызывается при подключении к чату нового пользователя. В список ListBox необходимо добавить имя нового участника разговора:

```
procedure TClientForm.ConnectUser(Msg: PUserConnectMsg);
begin
  ListBox.Items.Add(Msg.UserName);
end;
```

При поступлении сообщения о том, что получено текстовое сообщение, вызывается процедура `SendText`:

```
procedure TClientForm.SendText (Msg: PUserSendTextMsg);  
begin  
    RichEdit.Lines.Add (Msg.Text);  
end;
```

Выбытие из разговора одного из собеседников приводит к выполнению процедуры `DisconnectUser`, которая удаляет из списка `ListBox` запись, соответствующую выбывшему участнику:

```
procedure TClientForm.DisconnectUser (Msg: PUserDisconnectMsg);  
var Index: Integer;  
begin  
    Index:=ListBox.Items.IndexOf (Msg.UserName);  
    ListBox.Items.Delete (Index);  
end;
```

Процесс подключения клиентского приложения начинается с запроса имени сервера и имени пользователя. Используем для этого обработчик события `OnCreate` формы:

```
procedure TClientForm.FormCreate (Sender: TObject);  
var ServerName: String;  
begin  
    ServerName := '';  
    FUserName := '';  
    if InputQuery ('Введите имя сервера', 'Имя сервера:', ServerName)  
    and InputQuery ('Введите имя пользователя',  
        'Имя пользователя:', FUserName) and (ServerName <> '')  
    and (FUserName <> '') then begin  
        ClientSocket.Host := ServerName;  
        ClientSocket.Active := True;  
    end  
    else Application.Terminate;  
end;
```

Функция `InputQuery` отображает окно для ввода строковой информации и, если была нажата кнопка ОК, возвращает `True`, иначе `False`.

Если ввод имени сервера и имени пользователя прошел удачно, то мы устанавливаем значение свойства `Host` равным введенному имени сервера. Следующий этап — это попытка подключиться к серверу (устанавливаем значение свойства `ClientSocket.Active` равным `true`).

При удачном подключении возникает событие OnConnect компонента ClientSocket, в ответ на которое необходимо передать серверу сообщение с идентификатором MSG\_USERCONNECT:

```
procedure TClientForm.ClientSocketConnect(Sender: TObject;  
    Socket: TCustomWinSocket);  
var Msg: TUserConnectMsg;  
begin  
    Msg.MsgNumber:=MSG_USERCONNECT;  
    Msg.UserName:=FUserName;  
    Socket.SendBuf(Msg, SizeOf(TUserConnectMsg));  
end;
```

На этом процесс подключения клиентского приложения завершается, клиентский сокет переходит в режим отправки/приема сообщений.

*Эксперимент.* Запустите серверное приложение, после чего запустите клиентское приложение. Убедитесь, что при правильном указании имени сервера устанавливается соединение с сервером и в список сервера вносится имя пользователя.

Что происходит после завершения работы клиентского приложения?

*Примечание.* Для тестирования отдельных приложений клиента и сервера не обязательно иметь несколько компьютеров. Достаточно иметь лишь один, на котором можно одновременно запустить и сервер, и клиент, при этом в качестве имени компьютера, к которому надо подключиться, использовать хост-имя localhost или IP-адрес — 127.0.0.1. ♦

По окончании работы клиентского приложения необходимо завершить соединение:

```
procedure TClientForm.FormCloseQuery(Sender: TObject;  
    var CanClose: Boolean);  
begin  
    ClientSocket.Socket.Active := False;  
end;
```

В ответ на закрытие сокета возникает событие OnDisconnect компонента ClientSocket:

```
procedure TClientForm.ClientSocketDisconnect(Sender: TObject;  
    Socket: TCustomWinSocket);  
var Msg: TUserConnectMsg;  
begin  
    Msg.MsgNumber:=MSG_USERDISCONNECT;  
    Msg.UserName:=FUserName;
```

```
Socket.SendBuf(Msg, SizeOf(TUserConnectMsg));  
                                {отправляем серверу имя пользователя,  
                                чтобы удалить его из списков активных пользователей}  
Close;  
end;
```

**Эксперимент.** Запустите клиентское приложение (серверное приложение было уже запущено). Убедитесь, что при закрытии клиентского приложения имя пользователя удаляется из списка подключенных пользователей сервера. ♦

**Примечание.** Исходные тексты файлов модулей и файлов форм приведены в приложении 4.

В ответ на щелчок на кнопке Отправить необходимо переслать введенное текстовое сообщение на сервер:

```
procedure TClientForm.ButtonClick(Sender: TObject);  
var Msg: TUserSendTextMsg;  
begin  
    Msg.MsgNumber:=MSG_USERSENDEXTXT;  
    Msg.Text:=Edit.Text;  
    ClientSocket.Socket.SendBuf(Msg,  
    SizeOf(TUserSendTextMsg));  
end;
```

**Эксперимент.** Запустите два клиентских приложения. Убедитесь в работоспособности обработчика события OnClick кнопки отправить. ♦

## Задания для самостоятельного выполнения

### 12.3. Модифицируйте код приложения «Чат для локальной сети»:

- а) в серверном приложении отобразите сообщения, посылаемые пользователями;
- б) реализуйте фильтр для текстовых приложений: предположим, что некоторые слова недопустимы в контексте чата, эти слова должны заменяться на символ «☺».

**Примечание.** Воспользуйтесь функцией StringReplace (гл. 8);

- с) дополните пересылаемое пользователем сообщение именами отправителя и получателя;
- д) определите возможность настройки цвета для отображения текстовых сообщений (текст каждого пользователя отображается выбранным им при входе в чат цветом);
- е) создайте ini-файл для хранения настроек клиентского приложения (имя сервера, имя пользователя и др.);

- f) обеспечьте возможность поиска пользователя с определенным именем, подключенным к чату;
- g) каждый пользователь имеет свой уникальный псевдоним (nickname). Введите защиту идентификации пользователя (псевдонима) от возможности дублирования.

## 12.4. Посылка файлов через сокет

Чтобы переслать файл через сокет, необходимо открыть этот файл как файловый поток (TFileStream) и отправить его через сокет (SendStream):

```
procedure SendFileBySocket(filename: string);  
var temp: TFileStream;  
begin  
    temp := TFileStream.Create(filename, fmOpenRead);  
                                {открываем файл с именем filename}  
    ServerSocket1.Socket.Connections[0].SendStream(temp);  
                                {посылаем данные первому подключенному клиенту}  
    temp.Free;                                {закрываем файл}  
end;
```

Принимать данные из сокета можно через событие OnRead, используя универсальный метод ReceiveBuf:

```
procedure TForm1.ClientSocket1Read(Sender: TObject;  
Socket: TCustomWinSocket);  
var len: Integer;  
    buf: PChar;  
    temp: TFileStream;  
begin  
    len := Socket.ReceiveLength;  
                                {определяем размер полученного блока}  
    GetMem(buf, len+1);        {выделяем память для буфера}  
    Socket.ReceiveBuf(buf, len);  
                                {записываем в буфер полученный блок}  
    temp := TFileStream.Create('myfile.tmp', fmOpenReadWrite);  
                                {открываем временный файл для записи}  
    temp.Seek(0, soFromEnd);    {перемещаем позицию в конец файла}  
    temp.WriteBuffer(buf, len); {записываем буфер в файл}  
    temp.Free;                  {закрываем файл}  
    FreeMem(buf);               {освобождаем память}  
end;
```

### Задания для самостоятельного выполнения

- 12.4.** Дополните приложение «Chat» возможностью пересылки файлов.
- 12.5.** Напишите приложение «Морской бой» (рис. 12.4.1). Игроют 2 человека. Поле слева размером  $10 \times 10$  предназначено для расстановки собственных кораблей (из списка в нижней части формы выбирается корабль и ставится на поле), поле справа — для «стрельбы» по кораблям противника.

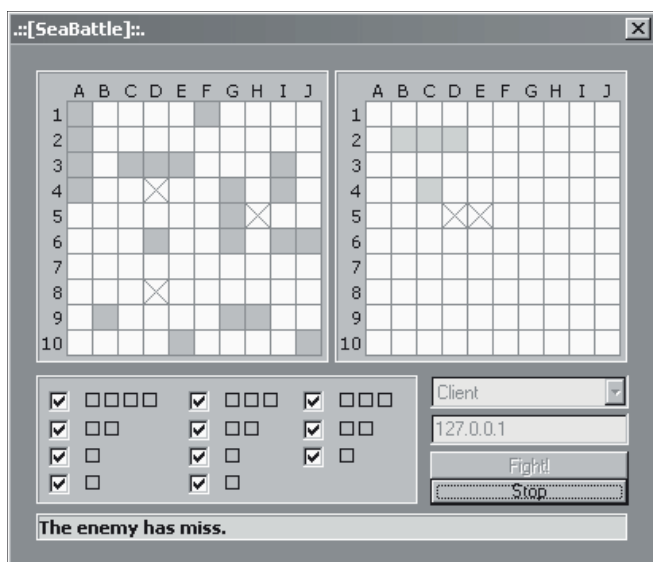


Рис. 12.4.1

После расстановки кораблей начинается сражение. Первый игрок делает выстрел: если выстрел был неудачен — ход передается противнику, при попадании в корабль противника предоставляется право еще одного выстрела. Игра заканчивается, когда все корабли одного из игроков будут подбиты.

- 12.6.** «Блоки». На поле размером  $6 \times 6$  находятся картинки (по две одинаковые), невидимые для игроков. Необходимо открыть все картинки. За каждый ход игрок может открыть по две картинки (открываемые картинки должны быть показаны в сетках всех игроков), при этом если

- картинки совпадают, то они остаются открытыми, а игроку предоставляется сделать еще один ход;
- картинки не совпадают — ход передается сопернику.

Написать сетевую версию этой игры для  $N$  участников ( $N$  может принимать значения от 1 до 6).

**12.7. «Последовательность».** Играют двое. Дана последовательность из  $N$  положительных чисел. Игрок выбирает число, расположенное на левом или правом конце последовательности. Выбранное число удаляется, ход переходит сопернику. Игра заканчивается, когда все числа уже выбраны. Выигрывает тот игрок, у которого сумма выбранных чисел больше.

**12.8. «Уголки».** На доске размером  $9 \times 9$  в левом нижнем и правом верхнем углах расставлены шашки (рис. 12.4.2). Начинают «белые». За один ход можно переместиться на одну клетку влево, вправо, вперед, назад либо перепрыгнуть через шашку в одном из указанных направлений. Цель игры — переставить белые шашки на место черных (или черные на место белых). Выигрывает тот, кто быстрее переставит фишки.

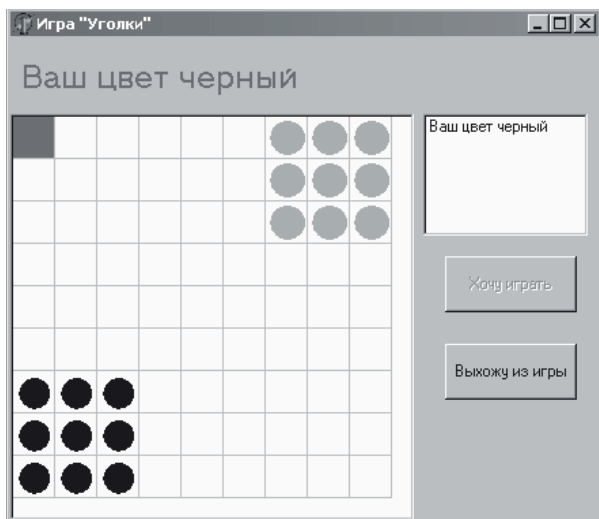


Рис. 12.4.2

**12.9.** Игра «Балда». Число игроков не менее двух. В среднем ряду квадратной таблицы размерности  $N$  ( $N$  — нечетное число) записывается слово (рис. 12.4.3). Используя буквы таблицы и обязательно добавив одну букву, необходимо составить слово. Слово — это существительное, записанное в именительном падеже, единственном числе (не может быть именем собственным), оно может располагаться в один ряд, столбец или с поворотами на 90 градусов (рис. 12.4.3). Слова не должны повторяться, т. е. нельзя вписать еще одно слово «мечта», подписав букву М в первом столбце второго ряда. За каждое слово игрок зарабатывает количество очков, равное числу букв в слове.

М	Е	Ч	Т	А
	Д			

**Рис. 12.4.3**

Игроки ходят по очереди, причем на придумывание слова отводится не более двух минут, по истечении этого времени, если слово не составлено, ход передается следующему игроку.

Игра заканчивается, когда все клетки поля заполнены либо ни один игрок не может придумать очередное слово. Выигрывает тот, кто набрал большее количество очков.

**12.10.** Напишите клиент-серверное приложение для работы с базой данных. Серверное приложение служит для хранения данных, а клиентское — для ввода данных.

### Идея решения

Клиентское приложение содержит форму для ввода данных (рис. 12.4.4) и использует таблицу (создается при запуске приложения и хранится в текущем каталоге) со строковыми полями Company (имя компании), Address (адрес), State (штат), Country (страна), Email и Contact (имя контактного лица), а также полем с плавающей точкой для идентификатора компании (CompID).

После щелчка на кнопке SendAll все новые записи пересылаются серверу. Новые записи определяются значением поля CompID, которое формируется серверным приложением при пересылке данных. Новые записи таблицы клиентская программа упаковывает в список строк (TStringList), используя структуру Имя\_поля=Значение\_поля. Затем длинная строка, соответствующая всему списку, посылается на сервер:

```
Procedure TForm1.btnSendAllClick (Sender: TObject);  
var  
    Data: TStringList;  
    I: integer;  
begin  
    ClientSocket1.Address := EditServer.Text;  
    ClientSocket1.Active := True;  
    Application.ProcessMessages;  
        {Application.ProcessMessages позволяет операционной системе  
        перерисовать нужные элементы окна и дает время  
        другим программам. Если бы этого оператора не было и данные  
        довольно долго не поступали, то система бы слегка «подвисла»}  
    Data := TStringList.Create;  
        {записываем данные из базы в список строк}  
  
    try  
        Table1.First;  
        while not Table1.EOF do begin  
            if Table1.FieldName('CompID').IsNull or  
            (Table1.fieldbyname('CompID').AsInteger=0) then  
                {если запись еще не записана}  
            Iblog.Items.Add ('Sending' + Table1.fieldbyname('Company').  
AsString);  
            Data.Clear;  
            for i := 0 to Table1.FieldCount-1 do  
                {создаем строки вида Имя_поля=Значение_поля}  
            Data.Values [Table1.Fields[i].FieldName] :=  
Table1.Fields[i].AsString;  
            ClientSocket1.Socket.SendText (Data.Text);  
                {посылаем запись}  
  
            fWaiting := true;  
            while fWaiting do Application.ProcessMessages;  
                {Цикл завершит свое выполнение, если серверное приложение  
                пришло сообщение, что запись получена, либо пользователь нажмет  
                на кнопку Stop}  
  
            end;  
            Table1.Next;  
        end;
```

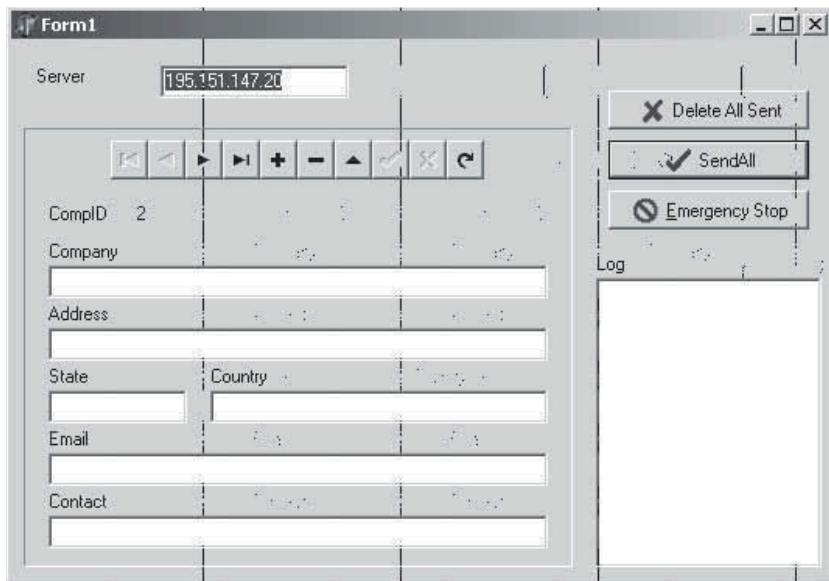
**finally**

```
Data.Free;           {освобождаем ресурсы и закрываем соединение}
ClientSocket1.Active := False;
```

**end;**

**end;**

После передачи записи клиентская программа переходит в режим ожидания, выход из которого происходит при получении ответа от сервера, при этом изменяется значение переменной `fWaiting` и каждая запись, которая была отправлена на сервер, получает значение идентификатора компании.



**Рис. 12.4.4**

Серверное приложение использует таблицу, хранящуюся в текущем каталоге. В отличие от таблицы клиентского приложения, она содержит еще два поля — `LoggedBy` (строковое поле) и `LoggedOn` (поле даты). Значения этих полей вычисляются автоматически при получении данных:

```
procedure TForm1.ServerSocket1.ClientRead (Sender: TObject;
Socket: TCustomWinSocket);
var
  strCommand: string;
```

```

strFeedback: string;
Data: TStringList;
I: integer;
begin
    strCommand := Socket.ReceiveText;
                                {считываем данные, поступающие от клиента}
    IbLog.Items.Add (strCommand);
    Data := TStringList.Create;                                {воссоздаем данные}
    try
        Data.Text := strCommand;
        Table1.Insert;
        for i := 1 to Table1.FieldCount-1 do
            {устанавливаем поля в соответствии с полученными строками}
            Table1.Fields[i].AsString :=
Data.Values[Table1.Fields[i].FieldName];

            Table1.FieldName('CompID').AsInteger := GetTickCount;
            Table1.FieldName('LoggedBy').AsString :=
Socket.RemoteAddress;
            Table1.FieldName('LoggedOn').AsDateTime := Date;
                                {дополняем данные случайным идентификатором, адресом
                                отправителя и датой}

            Table1.Post;

            strFeedback:=Table1.CompID.AsString;                {получаем значение,
                                                                которое нужно вернуть клиенту}
            IbLog.Items.Add (strFeedback);                      {возвращаем результат}
            Socket.SendText (strFeedback);

        finally
            Data.Free
        end
    end
end

```

### Вопросы для повторения

1. Что такое сокет?
2. Опишите схему работы серверного сокета (последовательность происходящих событий).
3. После выполнения следующих двух операторов серверного сокета

```

ServerSocket1.Socket.Connections[0].SendText('Hello, ');
ServerSocket1.Socket.Connections[0].SendText('world!');

```

клиентский сокет получил единственное сообщение «Hello, world!». Объясните, почему это могло произойти.
4. Для чего предназначен класс TCistomWinSocket?

# Приложение 1

## Класс TCanvas

```
TCanvas = class (TPersistent)  
private  
  FHandle: HDC;  
  State: TCanvasState;  
  FFont: TFont;  
  FPen: TPen;  
  FBrush: TBrush;  
  FPenPos: TPoint;  
  FCopyMode: TCopyMode;  
  FOnChange: TNotifyEvent;  
  FOnChanging: TNotifyEvent;  
  FLock: TRTLCriticalSection;  
  FLockCount: Integer;  
  FTextFlags: Longint;  
  procedure CreateBrush;  
  procedure CreateFont;  
  procedure CreatePen;  
  procedure BrushChanged(ABrush: TObject);  
  procedure DeselectHandles;  
  function GetCanvasOrientation: TCanvasOrientation;  
  function GetClipRect: TRect;  
  function GetHandle: HDC;  
  function GetPenPos: TPoint;  
  function GetPixel(X, Y: Integer): TColor;  
  procedure FontChanged(AFont: TObject);  
  procedure PenChanged(APen: TObject);  
  procedure SetBrush(Value: TBrush);  
  procedure SetFont(Value: TFont);  
  procedure SetHandle(Value: HDC);  
  procedure SetPen(Value: TPen);  
  procedure SetPenPos(Value: TPoint);  
  procedure SetPixel(X, Y: Integer; Value: TColor);  
protected  
  procedure Changed; virtual;  
  procedure Changing; virtual;  
  procedure CreateHandle; virtual;  
  procedure RequiredState(ReqState: TCanvasState);
```



```

function TextHeight(const Text: string): Integer;
                                {высота шрифта}
procedure TextOut(X, Y: Integer; const Text: string);
                                {вывод текста}
procedure TextRect(Rect: TRect; X, Y: Integer; const Text:
string);
function TextWidth(const Text: string): Integer;
                                {ширина текста}

function TryLock: Boolean;
procedure Unlock;
property ClipRect: TRect read GetClipRect;
property Handle: HDC read GetHandle write SetHandle;
property LockCount: Integer read FLockCount;
property CanvasOrientation: TCanvasOrientation read
GetCanvasOrientation;
property PenPos: TPoint read GetPenPos write SetPenPos;
property Pixels[X, Y: Integer]: TColor read GetPixel write
SetPixel;
                                {определяет цвет точки}
property TextFlags: Longint read FTextFlags write FTextFlags;
property OnChange: TNotifyEvent read FOnChange write FOnChange;
property OnChanging: TNotifyEvent read FOnChanging write
FOnChanging;
published
property Brush: TBrush read FBrush write SetBrush;
                                {кисть, заполнение}
property CopyMode: TCopyMode read FCopyMode write
FCopyMode default cmSrcCopy;
property Font: TFont read FFont write SetFont;           {шрифт}
property Pen: TPen read FPen write SetPen;
                                {перо, линия}
end;

```

TFont = **class**(TGraphicsObject)

**private**

```

FColor: TColor;
FPixelsPerInch: Integer;
FNotify: IChangeNotifier;
procedure GetData(var FontData: TFontData);
procedure SetData(const FontData: TFontData);

```

**protected**

```

procedure Changed; override;
function GetHandle: HFont;
function GetHeight: Integer;
function GetName: TFontName;
function GetPitch: TFontPitch;

```

```

function GetSize: Integer;
function GetStyle: TFontStyles;
function GetCharset: TFontCharset;
procedure SetColor(Value: TColor);
procedure SetHandle(Value: HFont);
procedure SetHeight(Value: Integer);
procedure SetName(const Value: TFontName);
procedure SetPitch(Value: TFontPitch);
procedure SetSize(Value: Integer);
procedure SetStyle(Value: TFontStyles);
procedure SetCharset(Value: TFontCharset);
public
  constructor Create;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
  property FontAdapter: IChangeNotifier read FNotify write
FNotify;
  property Handle: HFont read GetHandle write SetHandle;
  property PixelsPerInch: Integer read FPixelsPerInch write
FPixelsPerInch;
published
  property Charset: TFontCharset read GetCharset write SetCharset;
                                     {список шрифтов}
  property Color: TColor read FColor write SetColor;      {цвет}
  property Height: Integer read GetHeight write SetHeight;
                                     {высота}
  property Name: TFontName read GetName write SetName;
                                     {название}
  property Pitch: TFontPitch read GetPitch write SetPitch
default fpDefault;
  property Size: Integer read GetSize write SetSize stored
False;                                     {размер}
  property Style: TFontStyles read GetStyle write SetStyle;
                                     {стиль: подчеркнутый, полужирный и др.}
end;

```

TPen = **class**(TGraphicsObject)

```

private
  FMode: TPenMode;
  procedure GetData(var PenData: TPenData);
    procedure SetData(const PenData: TPenData);
protected
  function GetColor: TColor;
  procedure SetColor(Value: TColor);
  function GetHandle: HPen;
  procedure SetHandle(Value: HPen);

```

```

procedure SetMode(Value: TPenMode);
function GetStyle: TPenStyle;
procedure SetStyle(Value: TPenStyle);
function GetWidth: Integer;
procedure SetWidth(Value: Integer);
public
  constructor Create;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
  property Handle: HPen read GetHandle write SetHandle;
published
  property Color: TColor read GetColor write SetColor default
    clBlack; {цвет}
  property Mode: TPenMode read FMode write SetMode default
    pmCopy; {режим рисования линии}
  property Style: TPenStyle read GetStyle write SetStyle default
    psSolid; {стиль рисования линии}
  property Width: Integer read GetWidth write SetWidth default 1;
    {толщина линии}
end;

```

TBrush = **class** (TGraphicsObject)

```

private
  procedure GetData(var BrushData: TBrushData);
  procedure SetData(const BrushData: TBrushData);
protected
  function GetBitmap: TBitmap;
  procedure SetBitmap(Value: TBitmap);
  function GetColor: TColor;
  procedure SetColor(Value: TColor);
  function GetHandle: HBrush;
  procedure SetHandle(Value: HBrush);
  function GetStyle: TBrushStyle;
  procedure SetStyle(Value: TBrushStyle);
public
  constructor Create;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
  property Bitmap: TBitmap read GetBitmap write SetBitmap;
  property Handle: HBrush read GetHandle write SetHandle;
published
  property Color: TColor read GetColor write SetColor default
    clWhite; {цвет}
  property Style: TBrushStyle read GetStyle write SetStyle
default bsSolid; {стиль}
end;

```

## Приложение 2

### Файл модуля, содержащий описание компонента SimpleTree, отображающего структуру файловой системы в древовидной форме

```
unit SimpleTree;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TSimpleTree = class;
  TSimpleNode = class(TObject)
  private
    FTree: TSimpleTree;
    FParent: TSimpleNode;
    FChildren: TList;
    FCaption: String;
    FLevel: Integer;
    FIndex: Integer;
    FX, FY: Integer;
    FExpanded: Boolean;
    FAbsoluteIndex: Integer;
    function GetChildren(Index: Integer): TSimpleNode;
    function GetChildrenCount: Integer;
    procedure SetCaption(const Value: String);
    function GetFullPath: String;
    function GetSelected: Boolean;
    procedure SetSelected(const Value: Boolean);
    procedure Redraw;
    procedure DrawAt(X, Y: Integer);
    procedure SetExpanded(const Value: Boolean);
  public
    constructor Create(ATree: TSimpleTree);
    destructor Destroy; override;
    procedure ClearChildren;
```

```

property Children[Index: Integer]: TSimpleNode read
GetChildren;
property ChildrenCount: Integer read GetChildrenCount;
property Caption: String read FCaption write SetCaption;
property FullPath: String read GetFullPath;
property Level: Integer read FLevel;
property Selected: Boolean read GetSelected write SetSelected;
property AbsoluteIndex: Integer read FAbsoluteIndex;
property Index: Integer read FIndex;
property Expanded: Boolean read FExpanded write SetExpanded;
end;

TOnCheckforChildren = procedure (Sender: TObject;
ParentNode: TSimpleNode) of object;
TSimpleTree = class (TCustomControl)
private
  FBorder: TBorderStyle;
  FCharWidth, FCharHeight: Integer;
  FUpdateCount: Integer;
  FOnCheckforChildren: TOnCheckforChildren;
  FNodeSeparator: String;
  FMainNode: TSimpleNode;
  FNodes: TList;
  FStartIndex: Integer;
  FDrawList: TList;
  FMaxLinesInView: Integer;
  FSelectedNode: TSimpleNode;
  FTreeColors: array[0..3] of TColor;
  FMaxLines: Integer;
  FDrawLines: Boolean;
procedure SetBorder(const Value: TBorderStyle);
procedure CMFontChanged(var Msg: TMessage); message
CM_FONTCHANGED;
procedure WMSize(var Msg: TWMSize); message WM_SIZE;
procedure WMVScroll(var Msg: TWMVScroll); message WM_VSCROLL;
procedure UpdateCharMetrics;
procedure UpdateDrawList;
procedure UpdateScrollBar;
procedure UpdateNodeList;
function NodeInView(Node: TSimpleNode): Boolean;
procedure SetSelectedNode(const Value: TSimpleNode);
function GetNodeAt(X, Y: Integer): TSimpleNode;
function GetTreeColor(Index: Integer): TColor;
procedure SetTreeColor(const Index: Integer; const Value:
TColor);
function GetNode(Index: Integer): TSimpleNode;

```

```

function GetNodeCount: Integer;
procedure SetDrawLines(const Value: Boolean);
protected
procedure CreateParams(var Params: TCreateParams); override;
procedure Paint; override;
procedure CheckforChildren(ParentNode: TSimpleNode); virtual;
procedure MouseDown(Button: TMouseButton; Shift:
TShiftState; X, Y: Integer); override;
public
constructor Create(AOwner: TComponent); override;
destructor Destroy; override;
procedure BeginUpdate;
procedure EndUpdate;
function AddChild(Parent: TSimpleNode; const ACaption:
String): TSimpleNode;
property NodeAt[X, Y: Integer]: TSimpleNode read GetNodeAt;
property Nodes[Index: Integer]: TSimpleNode read GetNode;
default;
property NodeCount: Integer read GetNodeCount;
procedure ExpandNode(Node: TSimpleNode);
procedure CollapseNode(Node: TSimpleNode);
property MainNode: TSimpleNode read FMainNode;
property SelectedNode: TSimpleNode read FSelectedNode
write SetSelectedNode;
published
property Align;
property Anchors;
property Border: TBorderStyle read FBorder write
SetBorder default bsSingle;
property Color;
property Ctl3D;
property Font;
property TabOrder;
property TabStop;
property NodeSeparator: String read FNodeSeparator write
FNodeSeparator;
property TextColor: TColor index 0 read GetTreeColor
write SetTreeColor;
property LinesColor: TColor index 1 read GetTreeColor
write SetTreeColor;
property SelTextColor: TColor index 2 read GetTreeColor
write SetTreeColor;
property SelBackColor: TColor index 3 read GetTreeColor
write SetTreeColor;
property DrawLines: Boolean read FDrawLines write
SetDrawLines default True;

```

```

property OnCheckforChildren: TOnCheckforChildren read
FOnCheckforChildren write FOnCheckforChildren;
end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSimpleTree]);
end;

{ TSimpleTree }

function TSimpleTree.AddChild(Parent: TSimpleNode; const
ACaption: String): TSimpleNode;

function Count(Node: TSimpleNode): Integer;
  var i: Integer;
begin
  Result := 1;
  for i := 0 to Node.ChildrenCount - 1 do
    Result := Result + Count(Node.Children[i]);
end;

var i: Integer;
begin
  Result := nil;
  if Parent = nil then exit;
  Result := TSimpleNode.Create(Self);
  with Result do
    begin
      FParent := Parent;
      FCaption := ACaption;
      FLevel := FParent.Level + 1;
      FIndex := FParent.ChildrenCount;
      FAbsoluteIndex := FParent.FAbsoluteIndex + Count(FParent);
    end;
    Parent.FChildren.Add(Result);

    FNodes.Insert(Result.FAbsoluteIndex, Result);
    for i := Result.FAbsoluteIndex + 1 to FNodes.Count - 1 do
      TSimpleNode(FNodes[i]).FAbsoluteIndex:=i;
    if NodeInView(Result) then UpdateDrawList;
  end;

```

```
procedure TSimpleTree.BeginUpdate;
begin
    Inc(FUpdateCount);
end;

procedure TSimpleTree.CheckforChildren(ParentNode: TSimpleNode);
begin
    BeginUpdate;
    if Assigned(FOnCheckforChildren) then
        FOnCheckforChildren(Self, ParentNode);
    EndUpdate;
end;

procedure TSimpleTree.CMFontChanged(var Msg: TMessage);
begin
    inherited;
    UpdateCharMetrics;
end;

procedure TSimpleTree.CollapseNode(Node: TSimpleNode);
begin
    Node.FExpanded := False;
    Node.ClearChildren;
    UpdateNodeList;
    if NodeInView(Node) then
        begin
            UpdateDrawList;
            Invalidate;
        end;
end;

constructor TSimpleTree.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    ControlStyle := [csFramed, csCaptureMouse,
        csDoubleClicks, csClickEvents];
    FBorder := bsSingle;
    Width := 150;
    Height := 150;
    TabStop := True;
    FUpdateCount := 0;
    FNodeSeparator := '\';
    FStartIndex := 0;
    FMainNode := TSimpleNode.Create(Self);
    FDrawList := TList.Create;
    FNodes := TList.Create;
    FSelectedNode := FMainNode;
    FNodes.Add(FMainNode);
    FMainNode.FAbsoluteIndex := 0;
```

```
FTreeColors[0] := clGreen;
FTreeColors[1] := clOlive;
FTreeColors[2] := clWhite;
FTreeColors[3] := clNavy;
FDrawLines := True;
end;

procedure TSimpleTree.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  with Params do
    begin
      if FBorder = bsSingle then Style:=Style or WS_BORDER;
      Style:=Style or WS_VSCROLL;
    end;
  end;

destructor TSimpleTree.Destroy;
begin
  FMainNode.ClearChildren;
  FMainNode.Free;
  FDrawList.Free;
  FNodes.Free;
  inherited Destroy;
end;

procedure TSimpleTree.EndUpdate;
begin
  Dec(FUpdateCount);
  if FUpdateCount = 0 then Invalidate;
end;

procedure TSimpleTree.ExpandNode(Node: TSimpleNode);
begin
  Node.FExpanded:=True;
  if NodeInView(Node) then
    begin
      CheckforChildren(Node);
      UpdateDrawList;
      Invalidate;
    end;
  end;

function TSimpleTree.GetNode(Index: Integer): TSimpleNode;
begin
  Result := nil;
  if (Index >= 0) and (Index < FNodes.Count) then
    Result := FNodes[Index];
  end;
```

```
function TSimpleTree.GetNodeAt(X, Y: Integer): TSimpleNode;
var i: Integer;
begin
  i := 0;
  Result := nil;
  while (i < FDrawList.Count) and (Result = nil) do
    with TSimpleNode(FDrawList[i]) do
      if PtInRect(Rect(FX, FY,
        FX + FCharWidth*(1 + Length(Caption)),
        FY + FCharHeight), Point(X, Y)) then Result := FDrawList[i]
      else Inc(i);
  end;

function TSimpleTree.GetNodeCount: Integer;
begin
  Result := FNodes.Count;
end;

function TSimpleTree.GetTreeColor(Index: Integer): TColor;
begin
  Result := FTreeColors[Index];
end;

procedure TSimpleTree.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var Node: TSimpleNode;
begin
  inherited MouseDown(Button, Shift, X, Y);
  Node := NodeAt[X, Y];
  if Node <> nil then SelectedNode:=Node;
  if (Shift = [ssLeft, ssDouble]) and (FSelectedNode <> nil) then
    if FSelectedNode.FExpanded then CollapseNode(FSelectedNode)
    else ExpandNode(FSelectedNode);
end;

function TSimpleTree.NodeInView(Node: TSimpleNode): Boolean;
begin
  Result := FDrawList.IndexOf(Node) > -1;
end;

procedure TSimpleTree.Paint;

procedure DoDrawNodes;
var i: Integer;
begin
  for i:=0 to FDrawList.Count - 1 do
    TSimpleNode(FDrawList[i]).DrawAt(0, i * FCharHeight);
end;
```

```

procedure DoDrawLines;
var MaxLevel: Integer;
    i: Integer;
    j: Integer;
begin
    MaxLevel:=0;
    Canvas.Pen.Color:=LinesColor;
    for i:=0 to FDrawList.Count - 1 do
        with TSimpleNode(FDrawList[i]) do
            if FLevel > 0 then
                begin
                    Canvas.MoveTo(FX + FCharWidth, FY + FCharHeight div 2);
                    Canvas.LineTo(FX, FY + FCharHeight div 2);
                    if (FIndex > 0) and
                        (not NodeInView(FParent.Children[FIndex - 1])) then
                            Canvas.LineTo(FX, 0)
                    else
                        if FIndex = 0 then
                            Canvas.LineTo(FX, FY - FCharHeight div 2)
                        else Canvas.LineTo(FX, FParent.Children[FIndex - 1].FY);
                    if FIndex < FParent.ChildrenCount - 1 then
                        if not NodeInView(FParent.Children[FIndex + 1]) then
                            Canvas.LineTo(FX, ClientHeight);
                        if MaxLevel < FLevel then MaxLevel:=FLevel;
                    end;
                    for i:=1 to MaxLevel do
                        begin
                            j:=0;
                            while (j < FDrawList.Count) and
                                (TSimpleNode(FDrawList[j]).Level <> i) do Inc(j);
                            if j = FDrawList.Count then
                                begin
                                    Canvas.MoveTo((i + 1) * FCharWidth, 0);
                                    Canvas.LineTo((i + 1) * FCharWidth, ClientHeight);
                                end;
                            end;
                        end;
                    end;
                end;
            begin
                if FUpdateCount > 0 then exit;
                DoDrawNodes;
                if FDrawLines then DoDrawLines;
            end;

procedure TSimpleTree.SetBorder(const Value: TBorderStyle);
begin
    if FBorder <> Value then

```

```
begin
  FBorder := Value;
  RecreateWnd;
end;
end;

procedure TSimpleTree.SetDrawLines(const Value: Boolean);
begin
  if FDrawLines <> Value then
    begin
      FDrawLines := Value;
      Repaint;
    end;
end;

procedure TSimpleTree.SetSelectedNode(const Value: TSimpleNode);
var OldNode: TSimpleNode;
begin
  if FSelectedNode <> Value then
    begin
      OldNode:=FSelectedNode;
      FSelectedNode := Value;
      if (OldNode <> nil) and NodeInView(OldNode) then
        OldNode.Redraw;
      if NodeInView(FSelectedNode) then FSelectedNode.Redraw;
    end;
end;

procedure TSimpleTree.SetTreeColor(const Index: Integer;
  const Value: TColor);
begin
  FTreeColors[Index] := Value;
  Invalidate;
end;

procedure TSimpleTree.UpdateCharMetrics;
begin
  Canvas.Font := Self.Font;
  FCharHeight := Canvas.TextHeight('A') + 2;
  FCharWidth := Canvas.TextWidth('A');
end;

procedure TSimpleTree.UpdateDrawList;

function ListFull: Boolean;
begin
  Result := FDrawList.Count >= FMaxLinesInView;
end;
```

```
procedure FormDrawList(Node: TSimpleNode);  
var i: Integer;  
begin  
  if not ListFull then FDrawList.Add(Node);  
  Inc(FMaxLines);  
  if Node.FExpanded then  
    begin  
      for i := 0 to Node.ChildrenCount - 1 do  
        FormDrawList(Node.Children[i]);  
      Inc(FMaxLines, Node.ChildrenCount);  
    end;  
  end;  
  
var i, Min: Integer;  
begin  
  FMaxLinesInView := (ClientHeight div FCharHeight) + 1;  
  FDrawList.Clear;  
  FMaxLines:=0;  
  if FStartIndex + FMaxLinesInView > GetNodeCount then  
    Min := GetNodeCount - FStartIndex  
  else Min := FMaxLinesInView;  
  for i := FStartIndex to FStartIndex + Min - 1 do  
    FDrawList.Add(FNodes[i]);  
  for i := 0 to GetNodeCount - 1 do  
    with Nodes[i] do  
      if FParent = nil then Inc(FMaxLines)  
      else if FParent.FExpanded then Inc(FMaxLines);  
  UpdateScrollBar;  
end;  
  
procedure TSimpleTree.UpdateNodeList;  
  
  procedure UpdateAbsoluteIndex(Node: TSimpleNode);  
  var i: Integer;  
  begin  
    Node.FAbsoluteIndex := FNodes.Add(Node); for i := 0 to  
    Node.ChildrenCount - 1 do  
      UpdateAbsoluteIndex(Node.Children[i]);  
  end;  
  
  begin  
    FNodes.Clear;  
    UpdateAbsoluteIndex(FMainNode);  
  end;  
  
  procedure TSimpleTree.UpdateScrollBar;  
  var ScrollInfo: TScrollInfo;
```

```

begin
  if FMaxLinesInView >= FMaxLines then
    ShowScrollBar(Handle, SB_VERT, False)
  else begin
    FillChar(ScrollInfo, SizeOf(TScrollInfo), 0);
    ScrollInfo.cbSize := SizeOf(TScrollInfo);
    ScrollInfo.fMask := SIF_ALL;
    ScrollInfo.nMax := FMaxLines;
    ScrollInfo.nPage := FMaxLinesInView;
    ScrollInfo.nPos := FStartIndex;
    ShowScrollBar(Handle, SB_VERT, True);
    SetScrollInfo(Handle, SB_VERT, ScrollInfo, True);
  end;
end;

procedure TSimpleTree.WMSize(var Msg: TWMSize);
begin
  inherited;
  UpdateCharMetrics;
  UpdateDrawList;
  UpdateScrollBar;
end;

procedure TSimpleTree.WMVScroll(var Msg: TWMVScroll);
begin
  case Msg.ScrollCode of
    SB_THUMBPOSITION:
      begin
        SetScrollPos(Handle, SB_VERT, Msg.Pos, True);
        FStartIndex := Msg.Pos;
      end;
    SB_LINEUP: if FStartIndex > 0 then Dec(FStartIndex) else exit;
    SB_LINEDOWN:
      if FStartIndex < FMaxLines - FMaxLinesInView + 1
        then Inc(FStartIndex) else exit;
      else exit;
  end;
  UpdateDrawList;
  Invalidate;
end;

{ TSimpleNode }
procedure TSimpleNode.ClearChildren;
var i: Integer;
begin
  for i := 0 to FChildren.Count - 1 do

```

```
begin
  Children[i].ClearChildren;
  Children[i].Free;
end;
FChildren.Clear;
end;

constructor TSimpleNode.Create(ATree: TSimpleTree);
begin
  inherited Create;
  FTree := ATree;
  FParent := nil;
  FChildren := TList.Create;
  FLevel := 0;
  FIndex := -1;
  FExpanded := False;
end;

destructor TSimpleNode.Destroy;
begin
  ClearChildren;
  FChildren.Free;
  inherited Destroy;
end;

procedure TSimpleNode.DrawAt(X, Y: Integer);
begin
  FY := Y;
  with FTree.Canvas do
    begin
      Brush.Color:=FTree.Color;
      FX:=X + (Level + 1) * FTree.FCharWidth;
      if Selected then
        begin
          Brush.Color:=FTree.SelBackColor;
          Font.Color:=FTree.SelTextColor;
        end
      else Font.Color:=FTree.TextColor;
      TextOut(FX + FTree.FCharWidth + 1, Y, Caption);
    end;
  end;

function TSimpleNode.GetChildren(Index: Integer):
  TSimpleNode;
begin
  Result:=nil;
  if (Index >= 0) and (Index < FChildren.Count) then
    Result:=FChildren[Index];
  end;
```

```
function TSimpleNode.GetChildrenCount: Integer;
begin
    Result:=FChildren.Count;
end;

function TSimpleNode.GetFullPath: String;
begin
    if FParent <> nil then
        Result:=FParent.GetFullPath + FTree.NodeSeparator + Caption
    else Result:=Caption;
end;

function TSimpleNode.GetSelected: Boolean;
begin
    Result:=(FTree.SelectedNode = Self);
end;

procedure TSimpleNode.Redraw;
begin
    DrawAt(FX - (Level + 1) * FTree.FCharWidth, FY);
end;

procedure TSimpleNode.SetCaption(const Value: String);
begin
    if FCaption <> Value then
        begin
            FCaption:=Value;
            FTree.Invalidate;
        end;
end;

procedure TSimpleNode.SetExpanded(const Value: Boolean);
begin
    if Value <> FExpanded then if Value then
        FTree.ExpandNode(Self)
    else FTree.CollapseNode(Self);
end;

procedure TSimpleNode.SetSelected(const Value: Boolean);
begin
    if Selected then
        begin
            if not Value then FTree.SelectedNode:=nil;
        end
    else
        if Value then FTree.SelectedNode:=Self;
    end;
end.
```

## Файл модуля приложения (main.pas)

```
unit Main;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, SimpleTree;
```

```
type
```

```
  TMainFrm = class (TForm)  
    procedure FormCreate(Sender: TObject);  
    procedure FormDestroy(Sender: TObject);  
  private  
    Tree: TSimpleTree;  
    procedure TreeCheckforChildren(Sender: TObject;  
ParentNode: TSimpleNode);  
  end;
```

```
var
```

```
  MainFrm: TMainFrm;
```

```
implementation
```

```
{ $R *.DFM }
```

```
const
```

```
  sMyComputer = 'My computer';
```

```
procedure TMainFrm.FormCreate(Sender: TObject);
```

```
begin
```

```
  Tree := TSimpleTree.Create(Self);  
  Tree.OnCheckforChildren := TreeCheckforChildren;  
  Tree.Color := clWindow;  
  Tree.Parent := Self;  
  Tree.Left := 10;  
  Tree.Font.Name := 'Courier New';  
  Tree.MainNode.Caption := sMyComputer;  
  Tree.Align := alClient;  
end;
```

```
procedure TMainFrm.TreeCheckforChildren(Sender: TObject;  
    ParentNode: TSimpleNode);  
procedure AddDrives;  
var i: Char;  
begin  
    for i := 'a' to 'z' do  
        if GetDriveType(PChar(String(i + ':\'))) > 1 then  
            Tree.AddChild(ParentNode, i + ':');  
end;  
  
procedure AddFiles;  
  
    procedure AddNode(FileName: String);  
    begin  
        if (FileName <> '.') and (FileName <> '..') then  
            Tree.AddChild(ParentNode, FileName);  
    end;  
  
    var H: Integer;  
        R: TSearchRec;  
        S: String;  
    begin  
        S := ParentNode.FullPath;  
        Delete(S, 1, Length('My computer') + 1);  
        H:=FindFirst(S + '\*.*', faAnyFile, R);  
        if H = 0 then  
            begin  
                AddNode(R.Name);  
                while FindNext(R) = 0 do AddNode(R.Name);  
            end;  
    end;  
  
    begin  
        if ParentNode = Tree.MainNode then AddDrives else AddFiles;  
    end;  
  
    procedure TMainFrm.FormDestroy(Sender: TObject);  
    begin  
        Tree.Free;  
    end;  
  
    end.
```

## Приложение 3 «Поиск файлов»

### Исходный код файла формы Main.dfm

```
object MainForm: TMainFrm
  Left = 227
  Top = 136
  Width = 454
  Height = 259
  Caption = 'Поиск файлов'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 13
object Label1: TLabel
  Left = 8
  Top = 8
  Width = 25
  Height = 13
  Caption = 'Имя:'
end
object Label2: TLabel
  Left = 8
  Top = 32
  Width = 35
  Height = 13
  Caption = 'Папка:'
end
object EName: TEdit
  Left = 64
  Top = 0
  Width = 249
  Height = 21
  TabOrder = 0
  OnChange = ENameChange
end
```

```
object EPath: TEdit
    Left = 64
    Top = 32
    Width = 161
    Height = 21
    ParentShowHint = False
    ShowHint = True
    TabOrder = 1
end
object Button1: TButton
    Left = 232
    Top = 32
    Width = 81
    Height = 25
    Caption = 'Обзор'
    TabOrder = 2
    OnClick = Button1Click
end
object CheckBox1: TCheckBox
    Left = 8
    Top = 64
    Width = 305
    Height = 17
    Caption = 'Просмотреть вложенные папки'
    Checked = True
    State = cbChecked
    TabOrder = 3
end
object BFind: TButton
    Left = 328
    Top = 0
    Width = 113
    Height = 25
    Caption = 'Найти'
    TabOrder = 4
    OnClick = BFindClick
end
object BStop: TButton
    Left = 328
    Top = 32
    Width = 113
    Height = 25
    Caption = 'Стоп'
    TabOrder = 5
    OnClick = BStopClick
end
```

```
object BClear: TButton
    Left = 328
    Top = 64
    Width = 113
    Height = 25
    Caption = 'Сброс'
    TabOrder = 6
    OnClick = BClearClick
end
object lb: TListBox
    Left = 0
    Top = 104
    Width = 446
    Height = 128
    Align = alBottom
    ItemHeight = 13
    TabOrder = 7
    OnDblClick = lbDblClick
end
end
```

## Исходный код файла модуля Main.pas

```
unit Main;

interface

uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, Buttons,
    ExtCtrls, Menus, ComCtrls, thread, shellapi;

type
    TMainFrm = class (TForm)
        Label1: TLabel;
        Label2: TLabel;
        EName: TEdit;
        EPath: TEdit;
        Button1: TButton;
        CheckBox1: TCheckBox;
        BFind: TButton;
        BStop: TButton;
        BClear: TButton;
        lb: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure ENameChange(Sender: TObject);
```

```
procedure BFindClick(Sender: TObject);  
procedure Button1Click(Sender: TObject);  
procedure BStopClick(Sender: TObject);  
procedure ThreadDone(sender: TObject);  
procedure BClearClick(Sender: TObject);  
procedure lbDblClick(Sender: TObject);  
private  
  { Private declarations }  
public  
  { Public declarations }  
  HThread: SearchThread;  
end;  
  
var  
  MainFrm: TMainFrm;  
  
implementation  
  
uses OpenDir;  
  
{ $R *.DFM }  
  
procedure TMainfrm.ThreadDone(sender: TObject) ;  
begin  
  ShowMessage ('Поиск файлов завершен');  
  EName.Text := '';  
  BStop.Enabled := false;  
  BFind.Enabled := false;  
  HThread.Free;  
end;  
  
procedure TMainfrm.FormCreate(Sender: TObject);  
begin  
  EPath.Hint := EPath.Text;  
  BStop.Enabled := false;  
  BClear.Enabled := false;  
  BFind.Enabled := false;  
  EPath.Text := 'c:\';  
end;  
  
procedure TMainFrm.ENameChange(Sender: TObject);  
begin  
  BClear.Enabled := true;  
  BFind.Enabled := true;  
end;
```

```
procedure TMainFrm.BFindClick(Sender: TObject);  
begin  
    BStop.Enabled := true;  
    BFind.Enabled := false;  
    Hthread := SearchThread.create(ENAME.Text, epath.text,  
checkbox1.checked);  
    HThread.OnTerminate := threaddone;  
end;  
  
procedure TMainFrm.Button1Click(Sender: TObject);  
begin  
    OpenFrm.ShowModal;  
end;  
  
procedure TMainFrm.BStopClick(Sender: TObject);  
begin  
    hthread.terminate;  
    BFind.Enabled := false;  
    BStop.Enabled := false;  
end;  
  
procedure TMainFrm.BClearClick(Sender: TObject);  
begin  
    lb.Clear;  
    ENAME.text := '';  
    BFind.Enabled := false;  
    BStop.Enabled := false;  
end;  
  
procedure TMainFrm.lbDbClick(Sender: TObject);  
begin  
    ShellExecute(Handle, 'open', PChar(lb.Items[lb.itemindex]),  
nil, nil, SW_SHOWNORMAL);  
end;  
  
end.
```

## Исходный код файла формы OpenDir.dfm

```
object OpenFrm: TOpenFrm  
    Left = 260  
    Top = 140  
    Width = 210  
    Height = 251  
    Caption = 'Обзор папок'  
    Color = clBtnFace
```

```
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = True
OnCreate = FormCreate
PixelsPerInch = 96
TextHeight = 13
object DirectoryListBox1: TDirectoryListBox
  Left = 8
  Top = 24
  Width = 185
  Height = 169
  ItemHeight = 16
  TabOrder = 0
end
object DriveComboBox1: TDriveComboBox
  Left = 8
  Top = 0
  Width = 185
  Height = 19
  TabOrder = 1
  OnChange = DriveComboBox1Change
end
object BitBtn1: TBitBtn
  Left = 8
  Top = 192
  Width = 185
  Height = 25
  TabOrder = 2
  OnClick = BitBtn1Click
  Kind = bkOK
end
end
```

## Исходный код файла модуля OpenDir.pas

```
unit OpenDir;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, FileCtrl, Buttons;
```

**type**

```
TOpenFrm = class (TForm)
  DirectoryListBox1: TDirectoryListBox;
  DriveComboBox1: TDriveComboBox;
  BitBtn1: TBitBtn;
  procedure DriveComboBox1Change(Sender: TObject);
  procedure BitBtn1Click(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;
```

**var**

```
OpenFrm: TOpenFrm;
```

**implementation**

```
uses Main;
```

```
{ $R *.DFM }
```

```
procedure TOpenFrm.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
end;
```

```
procedure TOpenFrm.BitBtn1Click(Sender: TObject);
begin
  MainFrm.Epath.text:=DirectoryListBox1.Directory+'\';
  close;
end;
```

```
procedure TOpenFrm.FormCreate(Sender: TObject);
begin
  DriveComboBox1.Drive := 'c';
  DirectoryListBox1.Directory :=MainFrm.epath.text;
end;

end.
```

## Исходный код файла модуля потока Thread.pas

```
unit Thread;

interface

uses
  Classes, stdctrls, sysUtils;

type
  SearchThread = class(TThread)
  private
    { Private declarations }
    fileName, Path: string;
    SubDir : boolean;
    procedure AddFileFound;
    procedure FindFiles(FN, p : string);
    procedure SearchDirs(FN, p : string);
  protected
    procedure Execute; override;
  public
    constructor create(FFileName: string; FPath: string;
      FSubdir: Boolean);
    end;

implementation

uses main;

{ Important: Methods and properties of objects in VCL can only
be used in a method called using Synchronize, for example,

  Synchronize(UpdateCaption);

and UpdateCaption could look like,

  procedure SearchThread.UpdateCaption;
  begin
    Form1.Caption := 'Updated in a thread';
    end; }

{ SearchThread }
procedure searchThread.AddFileFound ;
begin
  MainFrm.lb.Items.Add(path+fileName);
end;
```

```

procedure SearchThread.FindFiles(FN, p : string);
  var SR: TSearchRec; found: integer;
begin
  Path:=p;
  Found := FindFirst(P+FN, faArchive , SR);
  while (Found = 0) and not terminated do begin
    FileName := SR.Name;
    Synchronize (AddFileFound);
    Found := FindNext(SR);
  end;
  FindClose(SR);
end;

procedure SearchThread.SearchDirs(FN, p : string);
  var sr: TSearchRec; found: integer;
begin
  found:= FindFirst(P + '.*', faDirectory, SR);
  while (found=0) and not terminated do
  begin
    if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1] <> '.')
    then
      begin
        FindFiles(FN, P + SR.Name + '\\');
        SearchDirs(FN,P + SR.Name + '\\');
      end;
      found := FindNext(SR);
    end;
  end;
  FindClose(SR);
end;

procedure SearchThread.Execute;
begin
  { Place thread code here }
  if not SubDir then FindFiles (Filename, Path)
  else SearchDirs(FileName, Path);
end;

constructor SearchThread.Create( FFileName: string; FPath:
string; FSubDir: Boolean);
begin
  filename := FFileName;
  Path := FPath ;
  SubDir := FSubdir;
  FreeOnTerminate := True;
  inherited Create(False);
end;

end.

```

## Приложение 4 «Chat» для локальной сети

### Исходный код файла модуля ChatMsgs.pas

```
unit ChatMsgs;

interface

const
    MSG_USERCONNECT = 1;
    MSG_USERSENDTEXT = 2;
    MSG_USERDISCONNECT = 3;

type
    TUserConnectMsg = record
        MsgNumber: Integer;
        UserName: ShortString;
    end;
    PUserConnectMsg = ^TUserConnectMsg;

    TUserSendTextMsg = record
        MsgNumber: Integer;
        Text: ShortString;
    end;
    PUserSendTextMsg = ^TUserSendTextMsg;

    TUserDisconnectMsg = record
        MsgNumber: Integer;
        UserName: ShortString;
    end;
    PUserDisconnectMsg = ^TUserDisconnectMsg;

implementation

end.
```

### Исходный код файла формы ClientUnit.dfm

```
object ClientForm: TClientForm
    Left = 192
    Top = 103
```

```
Width = 511
Height = 286
Caption = 'Чат'
Color = clBtnFace
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
OnCloseQuery = FormCloseQuery
OnCreate = FormCreate
PixelsPerInch = 96
TextHeight = 13
object Label1: TLabel
    Left = 8
    Top = 0
    Width = 61
    Height = 13
    Caption = 'Сообщения:'
end
object Label2: TLabel
    Left = 347
    Top = 0
    Width = 76
    Height = 13
    Anchors = [akTop, akRight]
    Caption = 'Пользователи:'
end
object Label3: TLabel
    Left = 6
    Top = 232
    Width = 93
    Height = 13
    Anchors = [akLeft, akBottom]
    Caption = 'Текст сообщения:'
end
object RichEdit: TRichEdit
    Left = 8
    Top = 16
    Width = 329
    Height = 207
    Anchors = [akLeft, akTop, akRight]
    ReadOnly = True
    TabOrder = 0
end
```

```
object ListBox: TListBox
    Left = 344
    Top = 16
    Width = 154
    Height = 207
    Anchors = [akLeft, akTop, akRight]
    ItemHeight = 13
    TabOrder = 1
end
object Edit: TEdit
    Left = 105
    Top = 232
    Width = 305
    Height = 21
    Anchors = [akLeft, akBottom]
    MaxLength = 255
    TabOrder = 2
end
object Button: TButton
    Left = 419
    Top = 228
    Width = 75
    Height = 27
    Anchors = [akLeft, akBottom]
    Caption = 'Отправить'
    TabOrder = 3
    OnClick = ButtonClick
end
object ClientSocket: TClientSocket
    Active = False
    ClientType = ctNonBlocking
    Port = 1024
    OnConnect = ClientSocketConnect
    OnDisconnect = ClientSocketDisconnect
    OnRead = ClientSocketRead
    Left = 28
    Top = 36
end
end
```

## Исходный код файла модуля ClientUnit.pas

```
unit ClientUnit;
```

```
interface
```

**uses**

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ScktComp, ChatMsgs, StdCtrls, ComCtrls;

**type**

```
TClientForm = class(TForm)
  ClientSocket: TClientSocket;
  RichEdit: TRichEdit;
  Label1: TLabel;
  ListBox: TListBox;
  Label2: TLabel;
  Label3: TLabel;
  Edit: TEdit;
  Button: TButton;
procedure FormCreate(Sender: TObject);
procedure ClientSocketConnect(Sender: TObject; Socket:
TCustomWinSocket);
procedure ClientSocketRead(Sender: TObject; Socket:
TCustomWinSocket);
procedure ClientSocketDisconnect(Sender: TObject;
Socket: TCustomWinSocket);
procedure ButtonClick(Sender: TObject);
procedure FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
private
  FUserName: String;
procedure HandleMessage(Msg: Pointer);
procedure ConnectUser(Msg: PUserConnectMsg);
procedure SendText(Msg: PUserSendTextMsg);
procedure DisconnectUser(Msg: PUserDisconnectMsg);
end;
```

**var**

```
ClientForm: TClientForm;
```

**implementation**

```
{ $R *.DFM }
```

```
procedure TClientForm.FormCreate(Sender: TObject);
var ServerName: String;
begin
  ServerName := '';
  FUserName := '';
```

```
if
  InputQuery('Введите имя сервера', 'Имя сервера:',
  ServerName) and
  InputQuery('Введите имя пользователя', 'Имя
  пользователя:', FUserName)
and (ServerName <> '') and (FUserName <> '') then
begin
  ClientSocket.Host := ServerName;
  ClientSocket.Active := True;
end
else Application.Terminate;
end;

procedure TClientForm.ClientSocketConnect(Sender: TObject;
Socket: TCustomWinSocket);
var Msg: TUserConnectMsg;
begin
  Msg.MsgNumber := MSG_USERCONNNNECT;
  Msg.UserName := FUserName;
  Socket.SendBuf(Msg, SizeOf(TUserConnectMsg));
end;

procedure TClientForm.ClientSocketRead(Sender: TObject;
Socket: TCustomWinSocket);
var Msg: Pointer;
    Size: Integer;
begin
  Size := Socket.ReceiveLength;
  GetMem(Msg, Size);
  Socket.ReceiveBuf(Msg^, Size);
  HandleMessage(Msg);
  FreeMem(Msg);
end;

procedure TClientForm.HandleMessage(Msg: Pointer);
begin
  case Integer(Msg^) of
    MSG_USERCONNNNECT: ConnectUser(Msg);
    MSG_USERSENDTEXT: SendText(Msg);
    MSG_USERDISCONNECT: DisconnectUser(Msg);
  end;
end;

procedure TClientForm.ConnectUser(Msg: PUserConnectMsg);
begin
  ListBox.Items.Add(Msg.UserName);
end;
```

```
procedure TClientForm.DisconnectUser (Msg: PUserDisconnectMsg);  
var Index: Integer;  
begin  
    Index := ListBox.Items.IndexOf (Msg.UserName);  
    ListBox.Items.Delete (Index);  
end;  
  
procedure TClientForm.SendText (Msg: PUserSendTextMsg);  
begin  
    RichEdit.Lines.Add (Msg.Text);  
end;  
  
procedure TClientForm.ClientSocketDisconnect (Sender:  
TObject; Socket: TCustomWinSocket);  
var Msg: TUserConnectMsg;  
begin  
    Msg.MsgNumber := MSG_USERDISCONNECT;  
    Msg.UserName := FUserName;  
    Socket.SendBuf (Msg, SizeOf (TUserConnectMsg));  
end;  
  
procedure TClientForm.ButtonClick (Sender: TObject);  
var Msg: TUserSendTextMsg;  
begin  
    Msg.MsgNumber := MSG_USERSENDTEXT;  
    Msg.Text := Edit.Text;  
    ClientSocket.Socket.SendBuf (Msg,  
SizeOf (TUserSendTextMsg));  
end;  
  
procedure TClientForm.FormCloseQuery (Sender: TObject;  
var CanClose: Boolean);  
begin  
    ClientSocket.Socket.Close;  
end;  
  
end.
```

## Исходный код файла формы ServerUnit.dfm

```
object ServerForm: TServerForm  
    Left = 196  
    Top = 107  
    Width = 543  
    Height = 138  
    Caption = 'Север'
```

```
Color = clBtnFace
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 13
object List: TListView
    Left = 0
    Top = 0
    Width = 535
    Height = 111
    Align = alClient
    Columns = <
        item
            Caption = 'Имя пользователя'
            Width = 200
        end
        item
            Caption = 'Дата / Время подключения'
            Width = 200
        end
    FlatScrollBars = True
    GridLines = True
    ReadOnly = True
    RowSelect = True
    TabOrder = 0
    ViewStyle = vsReport
end
object ServerSocket: TServerSocket
    Active = True
    Port = 1024
    ServerType = stNonBlocking
    OnClientRead = ServerSocketClientRead
    Left = 492
    Top = 272
end
end
```

## Исходный код файла модуля ServerUnit.pas

```
unit ServerUnit;

interface
```

**uses**

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ScktComp, StdCtrls, ComCtrls, ChatMsgs;

**type**

```
TServerForm = class (TForm)
  ServerSocket: TServerSocket;
  List: TListView;
  procedure ServerSocketClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
private
  procedure HandleMessage(Socket: TCustomWinSocket; Msg:
    Pointer; Size: Integer);
  procedure ConnectUser(Socket: TCustomWinSocket; Msg:
    PUserConnectMsg; Size: Integer);
  procedure SendText(Msg: PUserSendTextMsg; Size: Integer);
  procedure DisconnectUser(Socket: TCustomWinSocket; Msg:
    PUserDisconnectMsg; Size: Integer);
  function IndexOf(Caption: String): Integer;
end;
```

**var**

ServerForm: TServerForm;

**implementation**

{ \$R \*.DFM }

```
procedure TServerForm.ConnectUser(Socket:
  TCustomWinSocket; Msg: PUserConnectMsg; Size: Integer);
var i: Integer;
      M: TUserConnectMsg;
begin
  M.MsgNumber := MSG_USERCONNECT;
  for I := 0 to ServerSocket.Socket.ActiveConnections - 2 do
begin
    ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);
    M.UserName := List.Items[i].Caption;
    Socket.SendBuf(M, SizeOf(TUserConnectMsg));
  end;
  with List.Items.Add do
begin
    Caption := Msg.UserName;
    SubItems.Add(DateTimeToStr(Now));
    Data := Socket;
  end;
end;
```

```
procedure TServerForm.DisconnectUser(Socket:
TCustomWinSocket; Msg: PUserDisconnectMsg; Size: Integer);
var i: Integer;
    Index: Integer;
begin
    for i := 0 to ServerSocket.Socket.ActiveConnections - 1 do
        ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);
        Index := IndexOf(Msg.UserName);
        List.Items.Delete(Index);
end;

procedure TServerForm.HandleMessage(Socket:
TCustomWinSocket; Msg: Pointer; Size: Integer);
begin
    case Integer(Msg^) of
        MSG_USERCONNECT: ConnectUser(Socket, Msg, Size);
        MSG_USERSENDTEXT: SendText(Msg, Size);
        MSG_USERDISCONNECT: DisconnectUser(Socket, Msg, Size);
    end;
end;

function TServerForm.IndexOf(Caption: String): Integer;
begin
    for Result := 0 to List.Items.Count - 1 do
        if AnsiSameText(Caption, List.Items[Result].Caption) then
            Exit;
    Result:=-1;
end;

procedure TServerForm.SendText(Msg: PUserSendTextMsg;
Size: Integer);
var i: Integer;
begin
    for i := 0 to ServerSocket.Socket.ActiveConnections - 1 do
        ServerSocket.Socket.Connections[i].SendBuf(Msg^, Size);
end;

procedure TServerForm.ServerSocketClientRead(Sender: TObject;
Socket: TCustomWinSocket);
var Msg: Pointer;
    Size: Integer;
begin
    Size:=Socket.ReceiveLength;
    GetMem(Msg, Size);
    Socket.ReceiveBuf(Msg^, Size);
    HandleMessage(Socket, Msg, Size);
    FreeMem(Msg);
end;

end.
```

## Литература

1. Климов Ю. С. и др. Программирование в среде Turbo-Pascal 6.0. — Минск: Высш. шк., 1992.
2. Буч Г. Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.
3. Александровский А. Д. Delphi 4. Шаг в будущее — М.: ДМК, 1999.
4. Иванова Г. С., Ничушкина Т. Н., Пугачёва Е. К. Объектно-ориентированное программирование: Учеб. для вузов — М.: Изд-во МГТУ им. Н. Э. Баумана, 2001.
5. Тейксейра С., Пачеко К. Delphi 5. Руководство разработчика. Т. 1. Основные методы и технологии программирования. — М.: ИД «Вильямс», 2000.
6. Тейксейра С., Пачеко К. Delphi 5. Руководство разработчика. Т. 2. Разработка компонентов и программирование баз данных. — М.: ИД «Вильямс», 2000.
7. Соломон Д., Русинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. — СПб.: Питер; М.: ИТД «Русская редакция», 2001.
8. Окулов С. М., Торгашова Н. Э., Кедров А. В. Объектно-ориентированное программирование. — Киров, 1994.
9. Снейдер Й. Эффективное программирование TCP/IP. Библиотека программиста. — СПб.: Питер, 2002.
10. Мамаев Е., Вишневский А. Microsoft SQL Server 7 для профессионалов. — СПб.: Питер, 2001.
11. Окулов С. М. Программирование в алгоритмах. — М.: БИНОМ. Лаборатория знаний, 2004.
12. Окулов С. М. Основы программирования. — М.: БИНОМ. Лаборатория знаний, 2004.
13. Желонкин А. В. Основы программирования в интегрированной среде DELPHI. — М.: БИНОМ. Лаборатория знаний, 2004.

# Предметный указатель

## A

Abstract 22

## B

BDE 247

## C

Class 12

Constructor 13

## D

DataBase DeskTop 265

Destructor 13

Dynamic 22

## E

Except 86

## F

Finally 86

## I

Implementation 54

Interface 53

IP-адрес 289

## M

MDI 161

## O

Object inspector 29

Override 20

## P

Private 12

Property 181

Protected 12

Public 12

Published 12

## R

RAD 61

## S

SDF 77

SDI 161

SQL 269

## T

Try 86

## V

VCL 60

Virtual 20

## A

Абстрагирование 11

## Б

Библиотека визуальных  
компонентов 60

## В

Виртуальное адресное  
пространство 221

Возвратный адрес 289

Вычисляемые поля 267

## Г

Группа проектов 302

## Д

Деструктор 11, 13

Дизайнер форм 29

Директивы видимости 12

**И**

- Иерархия 16
- Инкапсуляция 11
- Инспектор объектов 29
  - комбинированная панель выбора объекта 29
  - страница свойств 29
  - страница событий 29
- Интерфейс
  - объекта 11
  - пользователя 161
- Исключительная ситуация 86

**К**

- Класс 11
  - TButton 62
  - TCalendar 136
  - TClientSocket 290
  - TColorDialog 157
  - TCombobox 102
  - TComponent 173
  - TControl 175
  - TCustomControl 177
  - TDataSource 251
  - TDBEdit 253
  - TDBGrid 251
  - TDBImage 253
  - TDBMemo 253
  - TDBNavigator 255
  - TDirectoryListBox 235
  - TDirList 236
  - TDrawGrid 114
  - TDriveCombobox 236
  - TEdit 63
  - TFindDialog 158
  - TFontDialog 155
  - TGraphicControl 176
  - TImageList 126
  - TIniFile 100
  - TLabel 64
  - TList 95
  - TListBox 82
  - TListView 305
  - TMainMenu 160
  - TMaskEdit 96

- TNoteBook 80
- TOpenDialog 153
- TOpenPictureDialog 159
- TPopupMenu 160
- TPrinterSetupDialog 157
- TPrintDialog 157
- TQuery 273
- TRadioGroup 102
- TReplaceDialog 158
- TRichEdit 162
- TSaveDialog 153
- TSavePictureDialog 160
- TScrollBar 102
- TServerSocket 297
- TShape 102
- TSpinEdit 102
- TStringGrid 122
- TStringList 100
- TStrings 77
- TTTable 250
- TTabs 83
- TThread 232
- TTimer 128
- TToolBar 125
- TWinControl 177
- Компонент 60
  - визуальный 60
  - графический 61
  - диалоговый 151
  - невизуальный 60
  - оконный 61
- Конструктор 11, 13
- М**
- Метод 12
  - абстрактный 22
  - виртуальный 20
  - динамический 21
  - статически полиморфный 18
- Многозадачность 220
  - вытесняющая 220
  - кооперативная 220
- Модульность 14
- Н**
- Наследование 16

**О**

- Объект 7
  - Application 51
  - Полиморфный 18
- Объектная декомпозиция 7
  - композиция 24
  - наполнение 24
- Объектно-ориентированное программирование 7
- Окно редактирования модуля 29

**П**

- Палитра компонентов 29
- Панель быстрого доступа 28
- Параллелизм 23
- Поведение объекта 10
- Поле, объектное 24
- Полиморфизм 18
- Простой 18
- Сложный 19
- Порт 289
- Поток 221
  - приоритет 222
- Протокол 289
- Процесс 23, 221

**Р**

- Реализация объекта 12
- Редактор полей таблицы базы данных 260
- Реляционная база данных 247

**С**

- Свойства
  - массив свойств 190
  - массив 188
  - объектного типа 186
  - перечислимого типа 184
  - простые 183
  - типа множества 185
- Событие 36, 192
- События
  - клавиатуры 41
  - мыши 33
  - системные 45
- Сокет 288
- Состояние объекта 10
- Сохраняемость 23

**Т**

- Таблица виртуальных методов ТВМ 20
- Таблица динамических методов ТДМ 21
- Тип 18
  - Файловый 92
- Типизация 18

**Ф**

- Файл модуля 53
- Файл проекта 51
- Файл формы 56, 182

# Оглавление

Введение .....	3
<b>1. Объектно-ориентированное программирование. ....</b>	<b>6</b>
1.1. Из истории развития языков программирования .....	6
1.2. Объектная декомпозиция .....	7
1.3. Основные элементы ООП. ....	11
1.4. Композиция и наполнение .....	24
<i>Задания для самостоятельного выполнения</i> .....	26
<i>Вопросы для повторения</i> .....	26
<b>2. Введение в среду программирования Delphi. ....</b>	<b>27</b>
2.1. Историческая справка. ....	27
2.2. Основные элементы среды программирования Delphi ...	28
2.3. Создание приложения. ....	29
2.3.1. Сохранение приложения .....	30
2.3.2. Запуск приложения. ....	30
2.3.3. Изменение свойств .....	31
2.3.4. Обработка событий .....	32
2.3.4.1. События мыши. ....	33
<i>Задание для самостоятельного выполнения</i> .....	41
2.3.4.2. События клавиатуры .....	41
<i>Задания для самостоятельного выполнения</i> .....	45
2.3.4.3. Системные события. Отладка приложения: точки прерывания .....	45
<i>Задания для самостоятельного выполнения</i> .....	50
<i>Вопросы для повторения</i> .....	50
<b>3. Файлы, составляющие приложения Delphi ....</b>	<b>51</b>
3.1. Файл проекта (.DPR). ....	51
<i>Задания для самостоятельного выполнения</i> .....	53
3.2. Файл модуля (.PAS) .....	53
<i>Задания для самостоятельного выполнения</i> .....	56
3.3. Файл формы (.DFM) .....	56
3.4. Дополнительные файлы приложения Delphi .....	57
<i>Вопросы для повторения</i> .....	59

<b>4. Введение в визуальное проектирование</b>	<b>60</b>
4.1. Визуальное проектирование	60
4.2. Компоненты Label, Edit, Button	62
<i>Задания для самостоятельного выполнения</i>	64
<i>Задания для самостоятельного выполнения</i>	74
<i>Вопросы для повторения</i>	76
<b>5. Списки строк. Обработка исключительных ситуаций</b>	<b>77</b>
5.1. Класс TStrings	77
<i>Задания для самостоятельного выполнения</i>	86
5.2. Исключительные ситуации	86
<i>Задания для самостоятельного выполнения</i>	91
5.3. Класс TList	95
<i>Задание для самостоятельного выполнения</i>	99
5.4. Классы TStringList и TIniFile. Динамическое помещение компонент на форму	100
<i>Задания для самостоятельного выполнения</i>	112
<i>Вопросы для повторения</i>	113
<b>6. Сетки строк</b>	<b>114</b>
6.1. Класс TDrawGrid	114
6.2. Класс TStringGrid	122
<i>Задания для самостоятельного выполнения</i>	133
<i>Вопросы для повторения</i>	139
<b>7. Интерфейс Drag&amp;Drop</b>	<b>140</b>
<i>Задания для самостоятельного выполнения</i>	148
<i>Вопросы для повторения</i>	150
<b>8. Невизуальные компоненты Delphi</b>	<b>151</b>
8.1. Диалоговые компоненты	151
8.1.1. Основные правила использования диалоговых панелей	152
8.1.2. Компоненты OpenFileDialog и SaveDialog	153
8.1.3. Компонент FontDialog	155
8.1.4. Компоненты PrintDialog и PrinterSetupDialog	157
8.1.5. Компонент ColorDialog	157

8.1.6. Компоненты FindDialog и ReplaceDialog . . . . .	158
8.1.7. Компонент OpenPictureDialog . . . . .	159
8.1.8. Компонент SavePictureDialog . . . . .	160
8.2. Компоненты-меню. . . . .	160
<i>Задание для самостоятельного выполнения. . . . .</i>	<i>166</i>
8.2.1. Форматирование абзаца. . . . .	166
<i>Задания для самостоятельного выполнения . . . . .</i>	<i>167</i>
8.2.2. Форматирование текста. . . . .	168
<i>Задания для самостоятельного выполнения . . . . .</i>	<i>169</i>
8.2.3. Элементы меню Правка. . . . .	169
<i>Задания для самостоятельного выполнения . . . . .</i>	<i>171</i>
<i>Вопросы для повторения. . . . .</i>	<i>172</i>
<b>9. Разработка компонентов в среде Delphi . . . . .</b>	<b>173</b>
9.1. Выбор класса-предка. . . . .	174
9.1.1. Класс TControl. . . . .	175
<i>Задание для самостоятельного выполнения. . . . .</i>	<i>176</i>
9.1.2. Класс TGraphicControl. . . . .	176
9.1.3. Класс TWinControl . . . . .	177
9.1.4. Класс TCustomControl . . . . .	177
9.2. Создание модуля компонента и тестового приложения. . . . .	177
9.3. Добавление свойств, методов и событий . . . . .	180
9.3.1. Простые свойства. . . . .	183
9.3.2. Свойства перечислимого типа . . . . .	184
9.3.3. Свойства типа множества . . . . .	185
9.3.4. Свойство-объект . . . . .	186
9.3.5. Свойство-массив . . . . .	188
9.3.6. Массив свойств . . . . .	190
9.3.7. Перекрытие и переопределение свойств. . . . .	191
9.3.8. Создание событий . . . . .	192
9.3.9. Создание методов. . . . .	194
9.4. Регистрация компонента в среде Delphi . . . . .	197
<i>Задания для самостоятельного выполнения . . . . .</i>	<i>207</i>
<i>Задания для самостоятельного выполнения . . . . .</i>	<i>218</i>
<i>Вопросы для повторения. . . . .</i>	<i>219</i>

<b>10. Создание многопоточных приложений</b>	<b>220</b>
10.1. Многозадачность и многопоточность	220
10.2. Функция CreateThread	222
<i>Задания для самостоятельного выполнения</i>	231
10.3. Класс TThread	232
<i>Задания для самостоятельного выполнения</i>	244
<i>Вопросы для повторения</i>	245
<b>11. Базы данных в Delphi</b>	<b>246</b>
11.1. Принципы построения баз данных	246
11.2. Компоненты, используемые для связи с базами данных	248
11.2.1. Компонент Table	250
11.2.2. Компонент DataSource и компоненты отображения данных	251
<i>Задания для самостоятельного выполнения</i>	260
11.3. Создание баз данных с помощью Database desktop	260
<i>Задания для самостоятельного выполнения</i>	269
11.4. Основы языка SQL	269
11.4.1. Оператор выбора	269
11.4.2. Объединение таблиц	271
11.4.3. Операции с записями	271
11.4.4. Операции над таблицей	271
11.4.5. Изменение структуры таблицы	273
11.4.6. Удаление таблиц	273
11.5. Компонент Query	273
<i>Задание для самостоятельного выполнения</i>	277
<i>Задания для самостоятельного выполнения</i>	284
<i>Вопросы для повторения</i>	287
<b>12. Программирование сокетов</b>	<b>288</b>
12.1. Компонент ClientSocket	290
<i>Задания для самостоятельного выполнения</i>	297
12.2. Компонент TServerSocket	297
<i>Задания для самостоятельного выполнения</i>	300
12.3. Посылка/прием сложных данных	300
<i>Задания для самостоятельного выполнения</i>	312

12.4. Посылка файлов через сокет . . . . .	313
<i>Задания для самостоятельного выполнения</i> . . . . .	314
<i>Вопросы для повторения</i> . . . . .	319
Приложение 1 . . . . .	320
Приложение 2 . . . . .	325
Приложение 3 «Поиск файлов» . . . . .	340
Приложение 4 «Chat» для локальной сети. . . . .	349
Литература . . . . .	358
Предметный указатель. . . . .	359

*Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 11-й для платформ Windows, Mac OS, Android, iOS, Windows Phone и BlackBerry; экран 10"*

*Учебное электронное издание*

**Бабушкина Ирина Анатольевна**  
**Окулов Станислав Михайлович**

**ПРАКТИКУМ ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ  
ПРОГРАММИРОВАНИЮ**

Подписано к использованию 20.02.15. Формат 125×200 мм

Издательство «БИНОМ. Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3

Телефон: (499) 157-5272

е-mail: [binom@Lbz.ru](mailto:binom@Lbz.ru)

<http://www.Lbz.ru>, <http://e-umk.Lbz.ru>, <http://metodist.Lbz.ru>

