В разделе рассматриваются простейшие приемы программирования («курс молодого бойца»), используемые при быстрой разработке в среде Delphi_7 приложений с графическим интерфейсом. Основные положения иллюстрируются работающими примерами с подробным описанием и комментариями. Раздел может быть использован как методическое пособие при программировании наукоемких задач. Лазурик В.М.

Раздел 2. Базовые приемы программирования в среде Delphi

2.1. Визуальная модель Delphi

2.1.1. Объектно-ориентированное программирование

В объектно-ориентированном программировании (ООП) разработчик оперирует объектами. Объект представляет собой совокупность свойств, методов и событий. Объекты позволяют строить программу не из чудовищных по сложности процедур и функций, а из кирпичиков-объектов, заранее наделенных нужными свойствами. В общем случае каждый объект «помнит» необходимую информацию, «умеет» выполнять некоторый набор действий и характеризуется набором свойств. То, что объект «помнит», хранится в его полях. То, что объект «умеет делать», реализуется в виде его внутренних процедур и функций, называемых методами. Свойства объектов аналогичны свойствам, которые мы наблюдаем у обычных предметов. Значения свойств можно устанавливать и читать. Программно свойства реализуются через поля и методы.

Объекты – это представители класса. Объекты объявляются в программе в разделе `var`, например: `Var  Student, Professor : TPerson;`

Класс – это обобщенное (абстрактное) описание множества однотипных объектов. Объекты являются конкретными представителями своего класса, их принято называть экземплярами класса. В приведенном примере объекты `Student` и `Professor` являются экземплярами класса `TPerson`. Вот пример объявления простого класса:

```
TPerson = class
private
fname: string[15];
faddress: string[35];
public
procedure show;
end;
```

В приведенном примере `TPerson` – это имя класса, `Fname` и `Faddress` – имена полей, `Show` – имя метода. Объявления свойств, методов и событий класса могут быть сделаны в разделах private (описания этого раздела доступны только этому классу), protected (описания этого раздела доступны только этому классу и его наследникам), public (описания этого раздела доступны всем), published (описания этого раздела отображаются в Object Inspector).

ООП держится на трех китах: инкапсуляции, наследовании и полиморфизме. Под *инкапсуляцией* понимается скрытие полей объекта с целью обеспечения доступа к ним только посредством методов класса. В концепции ООП *наследование* определяется, как возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам. Новый, порожденный класс (потомок), наследует свойства и методы своего базового, родительского класса. *Полиморфизм* означает, что в производных классах можно изменять работу уже существующих в базовом классе методов. При этом весь программный код, управляющий объектами родительского класса, пригоден для управления объектами дочернего класса без всякой модификации.

2.1.2. Компонентная модель Delphi

Компоненты – это более совершенные объекты. С ними можно работать визуально,

для этого у них есть необходимые свойства и методы. При создании технологии ООП о визуальности еще никто не думал. Когда фирма Borland создавала свою первую визуальную оболочку для Windows, пришлось немного доработать концепцию ООП, чтобы с объектами можно было работать визуально. Библиотека визуальных компонентов (Visual Component Library, VCL) Delphi содержит множество предопределенных типов компонентов, из которых пользователь может строить свою прикладную программу. Витрину библиотеки, палитру компонентов, можно увидеть, расположенной справа в полосе инструментальных панелей интегрированной среды разработки Delphi. До появления шестой версии в Delphi существовала только одна компонентная модель VCL. В шестой версии появилась новая библиотека CLX (Borland Component Library for Cross Platform).

Одним из основных компонентов является форма. Форма представляет собой стандартное окно приложения. Когда в визуальной среде разрабатывается новая форма, она создается как наследник класса `TForm`. Формы могут представлять главное окно приложения, диалоговые окна, многодокументные окна. Форма может служить контейнером для других объектов. В этом окне могут быть размещены визуальные и не визуальные компоненты в любом виде и порядке. Размещение компонентов управления на форме во время разработки осуществляется визуально. Все компоненты, расположенные на форме, а также обработчики событий, как для самой формы, так и для компонентов, являются членами класса формы.

2.1.3. Визуальная среда разработки

Интегрированная среда разработки IDE (Integrated Development Environment) – это среда, в которой есть все необходимое для проектирования, запуска и тестирования приложений. IDE интегрирует в себе редактор кодов, отладчик, инструментальные панели, редактор изображений, инструментарий баз данных и т.д. Основное окно интегрированной среды разработки Delphi 7 представлено на рис. 2.1.

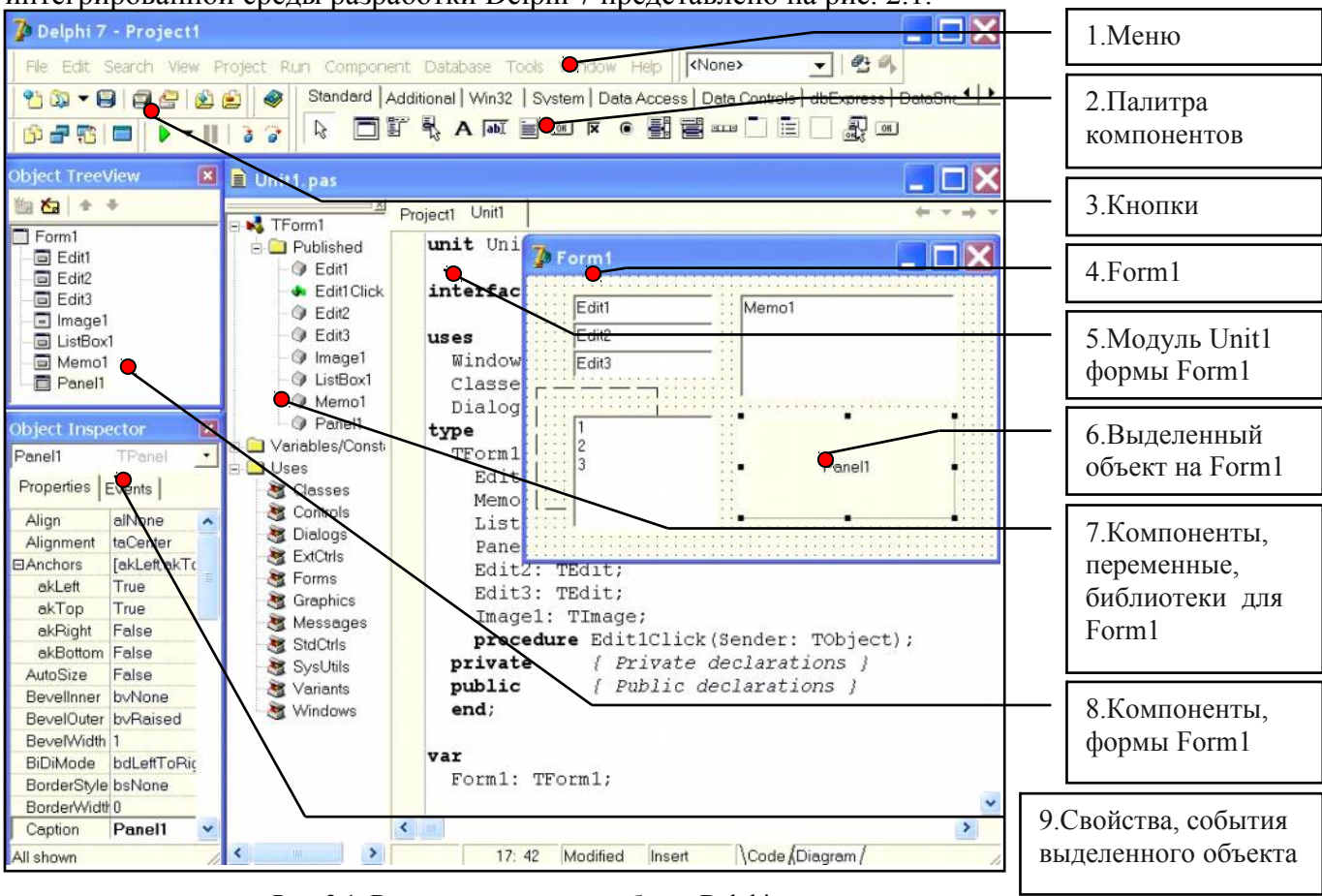

Рис. 2.1. Визуальная среда разработки Delphi

В верхней части окна расположена полоса главного меню (1). Ниже полосы главного меню расположены две инструментальные панели. Левая панель содержит два ряда быстрых кнопок (3), дублирующих некоторые наиболее часто используемые команды меню. Правая панель содержит палитру компонентов (2) – витрину библиотеки визуальных компонентов VCL.

Палитра компонентов представлена в виде нескольких тематических закладок, на каждой из которых расположены разработанные и готовые к использованию компоненты.

Для разработки графического интерфейса приложения используются формы. Каждая форма имеет свое графическое представление (4) и соответствующий ей модуль (5), содержащий свойства и методы класса формы. Скалярные и векторные значения свойств выделенного на форме объекта (6), а также список событий для этого объекта визуализированы в окне Object Inspector (9). Окно Object TreeView (8) содержит список компонентов, расположенных на форме. Окно Exploring (7) содержит список объектов интерфейсной части выделенного модуля (Unit). Каждый разработчик имеет возможность настроить визуальную среду, как ему будет удобно для работы.

Чтобы перенести компонент на форму, надо открыть соответствующую закладку палитры компонентов, указать щелчком мыши необходимый компонент, после чего щелкнуть мышью в нужном месте формы. Можно сделать двойной щелчок на пиктограмме компонента, в этом случае он автоматически разместится в центре формы. Добавление компонента на форму приводит к появлению в модуле формы объектной переменной и ссылки на модуль, содержащий класс компонента в том случае, если он еще не подключен.

2.2. Основы языка программирования Delphi

2.2.1. Переменные, константы

Все переменные в Delphi должны быть явно описаны. Типы переменных представлены в таблице 2.1.

Типы переменных

Таблица 2.1

Целые		Вещественные		Текстовые		Логические	
Тип	Формат (бит)	Тип	Мантисса (байты)	Тип	Формат (байты)	Тип	Байты
Shortint	8 (знак)	Real48	11-12 (6)	Char (1 символ)	1	Boolean	1
Smallint	16 (знак)	Single	7-8 (4)	ShortString	2-256	ByteBool	1
Longint	32 (знак)	Double	15-16 (8)	AnsiString (String)	4-2Gb	WordBool	2
Int64	64 (знак)	Real	15-16 (8)	WideString	4-2Gb (Unicode)	LongBool	4
Byte	8 (б з)	Extended	19-20 (10)				
Word	16 (б з)	Comp	19-20 (8)				
Longword	32 (б з)	Currency	19-20 (8)				
Integer	32 (знак)						
Cardinal	32 (б з)						

Описание переменных и констант, например:

```
Var    X, Y, Z: Double;
      I, J, K: Integer;
```

```
      Ok: Boolean;
const  Max: Integer = 100;
```

Строковая константа в Delphi – это последовательность символов, заключенная в одиночные кавычки, например: 'Это строка'. Строки в Delphi индексируются, т. е. можно обратиться к символу строки по его индексу (начиная с 1). Пример:

```
Var MyStr:string;
Begin
 MyStr:='Batton';
 MyStr[2]:='u';
 ShowMessage(MyStr);  // Button
End;
```

По умолчанию Delphi использует десятичное представление для числовых констант. Если в программе необходимо оперировать шестнадцатеричными константами, в этом случае перед числом ставится символ $. В Delphi для задания значения цвета существуют встроенные константы. Например, clRed – красный, clBlue – голубой. Но цвет можно задать и в виде 4-байтной 16-ричной константы: $00XXXXXX, где первые два нуля – это неиспользуемый байт, третий и четвертый знак – значение Blue для RGB представления цвета, пятый и шестой знаки – Green, седьмой и восьмой – Red. Например: $00FF0000 – голубой, $0000FF00 – зеленый, $000000FF – красный, $00E4E4E4 – светло-серый.

Все символы и ключи, которыми можно оперировать, представлены в Windows в виде таблицы. В Delphi используется 8-битная расширенная таблица символов и ключей, ANSI таблица, которая берется из Windows. Каждому ключу или символу в этой таблице поставлено в соответствие определенное число. Например, Enter=13, Tab=9, BackSpace=8, 'A'=65.

В некоторых случаях бывает необходимо использовать символьное представление ANSI кода. Для этого необходимо разместить символ # перед значением ANSI кода или использовать функцию Chr: #7 – звонок, #27 – ESC, #13 – Enter, #0 – Пусто или Chr (7) – звонок, Chr (27) – ESC, Chr (13) – Enter. При необходимости, используя функцию Ord, можно получить значение ANSI кода символа, заданного в качестве аргумента, например:

```
Var   S: string;
 begin
  S :='The ASCII code for "c" is ' +
      IntToStr(Ord('c')) +  ' decimal';
  ShowMessage(S); // вывод сообщения
end;
```

В шаблоне каждого модуля, который формирует Delphi, в секции uses интерфейсной
части автоматически подключается модуль Windows. В этом модуле описаны встроенные константы целого типа (виртуальные ключи), которые определяют функциональные клавиши (VK_INSERT = 45, VK_ESCAPE = 27, VK_HOME = 36, …), цифры (VK_0 ..VK_9), латинские символы (VK_A .. VK_Z) и т. д.

Существует еще один тип данных, с которым необходимо познакомиться, это массивы. Массив – это последовательность переменных одного типа, имеющая общее имя. Синтаксис объявления массива выглядит так:

```
Имя массива : array [Min_индекс .. Max_индекс] of Тип переменных
```

Например:

```
Var   a : array [0..99] of string;
 begin
```

```
  a[0] :='Start';  // первый элемент массива
  a[99]:='Finish'; // сотый элемент массива
 end;
```

2.2.2. Операторы (циклы, условия)

Присваивание – `a : = b ;`

Условный оператор – существуют две формы оператора: `if...then` и `if...then...else`. Синтаксис оператора `if...then`:

```
if выражение then оператор
```

Выражение возвращает логическое значение. Если оно равно `True` – оператор выполняется, в противном случае – нет.
Например: `if J <> 0 then Result := I/J;`
Синтаксис оператора `if...then...else`:

```
if выражение then оператор1 else оператор2.
```

Если логическое значение выражения равно `True` – выполняется оператор1, иначе оператор2. Например:

```
if J = 0 then
  Exit
else
  Result := I/J;
```

Пример, когда в условном операторе используется составной оператор, ограниченный скобками `begin . . . end`:

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
Else   Exit;
```

Оператор цикла `for`.
В операторе `for` задается количество итераций цикла. Синтаксис :

```
for counter := initialValue to finalValue do statement; или
for counter := initialValue downto finalValue do statement;
```

Локальная переменная `counter` – порядкового типа; `initialValue` и `finalValue` –
выражения, совместимые по типу с `counter; statement` – простой или составной оператор, который не изменяет значения `counter`. На первом шаге работы оператора `for` значение `initialValue` назначается `counter`, после чего выполняется оператор тела цикла `statement`, причем после каждой итерации значение `counter` инкрементируется (для оператора цикла `for...to`) или декрементируется (для оператора цикла `for... downto`). Когда `counter` принимает значение, равное `finalValue`, `statement` выполняется еще один раз, и работа цикла прекращается.

Оператор цикла `repeat`. Синтаксис:

repeat statement$_1$; ...; statement$_n$; until expression;

expression – выражение, которое возвращает Boolean величину. Оператор repeat повторяет выполнение всей последовательности операторов statements, проверяя значение выражения expression после каждой итерации. Когда значение expression становится True, оператор repeat прекращает работу. Вся последовательность операторов statements выполняется, по крайней мере, один раз, поскольку проверка значения expression осуществляется в конце первой итерации. Пример:

```
repeat
  Write('Enter a value (1..9): ');
  Readln(I);
until (I = 9);
```

Оператор цикла while. Оператор while подобен оператору repeat за исключением того, что значение выражения expression проверяется перед первым выполнением последовательности операторов тела цикла. Тело цикла выполняется до тех пор, пока при проверке значения expression в начале каждой итерации будет получено значение false, в этом случае выполнение цикла прекращается. Цикл while может не выполниться ни одного раза, если expression принимает значение false в начале первой итерации. Синтаксис: while expression do statement;

2.2.3. Процедуры и функции

Участок кода, который выделен в отдельный именованный блок, называется процедурой или фунццией. Любая процедура начинается с ключевого слова Procedure, а функция с ключевого слова Function. После ключевого слова идет имя и в скобках – список параметров, если они есть. В случае Function после перечня параметров указывается тип функции, который определяет тип возвращаемого функцией значения.

Пример 2.2.3.-1. *Процедура без параметров*

```
Procedure Hello;
 Var s : string;
 Begin
   S := 'Hello, new user!';
 End;

Function Test (x:integer):boolean;
// функция с параметром, возвращает true или false
 Begin
   If x <>0 then result:=true  // или Test:=true
   Else result:=false;         // или Test:= false;
 End;
```

Пример 2.2.3.-2. *Функция с заданным по умолчанию параметром*

```
Function TestX (x:integer; y:integer=2):boolean;
  Begin
   If (x mod y) = 0 then result:=true
   Else result:=false;
 End;

Procedure Examine;
Var x:integer;
Begin
 If TestX (x) then ... //вызов 1:остаток от деления на 2 равен 0
 Else ... ; //вызов 1:остаток от деления на 2 не равен 0
```

```
 If TestX (x, 3) then ...//вызов 2:остаток от деления на 3 равен
0
  Else ... ; // вызов 2:остаток от деления на 3 не равен 0
End;
```

Функция `TestX` объявлена с двумя параметрами, второй `(y)` имеет значение по умолчанию = 2. В вызывающей процедуре `вызов 1` использует значение по умолчанию второго параметра функции, `вызов 2` задает явно значение второго параметра.

Передавать параметры в процедуру или функцию можно по значению или по адресу. Это касается простых переменных. Массивы и объекты передаются только по адресу. В приведенных примерах параметры передавались функции по значению. Параметр, переданный в процедуру или функцию по значению и измененный внутри тела процедуры или функции, в вызывающей процедуре свое значение не меняет, он является локальной переменной. Параметр, переданный процедуре или функции по адресу и измененный внутри тела процедуры или функции, сохраняет свое значение в вызывающей процедуре, в этом случае такая переменная считается глобальной. При передаче параметра по адресу можно говорить о возврате значения процедурой через этот параметр.

Пример 2.2.3.-3. *Передача параметра в процедуру по адресу*

```
Procedure Vizov;
 Var Iv, Iw:integer;
 Begin
  Iv:=5; Iw:=20;
  Subr(Iv,Iw);
  // значение Iv = 5; значение Iw = 25
 End;

Procedure Subr (n:integer; var m:integer);
 Begin
  n:=n+5; m:=m+5;
 End;
```

Разработка встроенных процедур, рекурсивный вызов процедур и перегрузка в «курсе молодого бойца» не обсуждаются. В этом случае полезно обратиться к литературе [2].

2.3. Общие моменты при разработке приложений

2.3.1. Файлы проекта

При создании нового приложения Delphi открывает новую пустую форму и создает заготовку будущего модуля (рис. 2.2). Комментарии (однострочные начинаются с пары символов «//», многострочные берутся в фигурные скобки) и объяснения кода шаблона, сгенерированного Delphi, представлены ниже. По умолчанию имя модуля `Unit1`.

```
unit Unit1;// Имя модуля
interface  // Описательная часть
uses       // Секция перечисления подключаемых модулей
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs;
type       // Объявление типов
  TForm1 = class(TForm) // Новый класс TForm1 – наследник TForm
  Private  // Закрытые члены класса формы
    { Private declarations }
  Public   // Открытые члены класса формы
```

```
    { Public declarations }
  end;
var                 // Объявление глобальных переменных
  Form1: TForm1;    // Объектная переменная Form1 типа TForm1
Implementation      // Исполнимая часть
{$R  *.dfm}          //   Директива компилятора, подключение  *.dfm
файла
end.              // Конец модуля
```

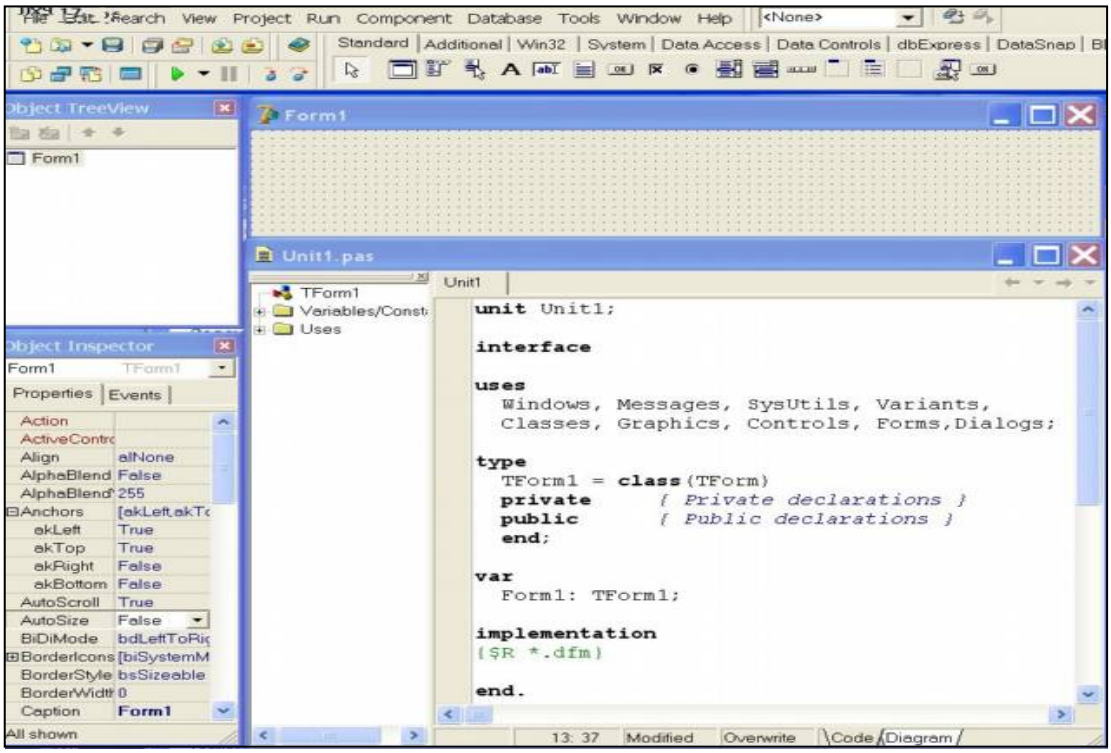

Рис. 2.2. Создание нового приложения

Разработаем приложение-приветствие, не программируя. В окне `Object Inspector` изменим свойство `Caption` формы с «`Form1`» на «`Привет!`» и значение цвета в свойстве `Color` с «`clBtnFace`» на «`clGray`». Сохраним разработку. Для этого необходимо нажать быструю кнопку `Save All` или выбрать соответствующий пункт главного меню. При сохранении файла `*.pas` можно задать оригинальное имя модулю, а при сохранении файла проекта `*.dpr` – оригинальное название проекта. Откомпилируем и запустим приложение (`Run`). Приветствие готово. Вернемся в среду разработки (`Program reset`), после чего откроем директорию, куда сохранили проект. Там появились файлы:

`Project1.dpr` – сам проект, в нем находится описание используемых модулей; `Project1.cfg` – конфигурация проекта; `Project1.dof` – опции проекта; `Project1.res` – ресурсы проекта; `Project1.exe` – исполнимый файл; `Unit1.pas` – исходный код модуля; `Unit1.dfm` – визуальная информация о форме, описанной в `Unit1`; `Unit1.dcu` – откомпилированный модуль `Unit1` проекта `Project1`;

2.3.2. Форма

Компонент `Form` реализован классом `TForm`, код которого находится в системном модуле `Forms`. Delphi автоматически подключает этот модуль к приложению на уровне файла проекта и в интерфейсной части каждого модуля формы:

```
Uses Windows, Messages, SysUtils, Classes,… Forms;
```

Когда последовательно создаются формы проекта, каждая форма имеет свойство `Visible:= false`. Первая по порядку создаваемая в проекте форма считается главной формой проекта (главным окном приложения) и для нее одной автоматически устанавливается свойство `Visible:= true`. Применение метода `Show` для какой-либо из форм тоже устанавливает свойство `Visible` для этой формы. Т. е. для того, чтобы визуализировать форму достаточно сделать для нее `Visible: = true.` Почему же применяют метод `Show`? Вызов этого метода инициирует событие `Show`, для которого можно написать обработчик. Как правило, так и делают в программах.

Последовательность событий от создания до визуализации формы:
1. `OnCreate` – создание формы (вызов конструктора);
2. `OnShow` – показ (визуализация);
3. `OnPaint` – отрисовка формы;
4. `OnActivate` – форма становится активной (получает фокус).

События формы `OnPaint` и `OnActivate` возникают, когда форма создана, и ее свойство `Visible` имеет значение `true`.

При закрытии формы (щелчок на системном меню формы или применен метод `Close`) происходят события `CloseQuery` и `Close.` Событие `CloseQuery` возникает перед закрытием формы и определяет условия, при которых форма может закрыться. Логический параметр `CanClose` по умолчанию имеет значение `true`, т. е. форме позволено закрыться. Событие `Close` возникает, когда форма закрывается. Параметр `Action` по умолчанию имеет значение `caHide`, т. е. форма становится скрытой (не видимой), приложение все равно имеет к ней доступ.

2.3.3. Обработчики событий

Первая форма проекта, представленного на рис. 2.2, является контейнером для нескольких компонентов. На ней расположены три компонента `Label`, три компонента `Edit` для ввода значений составляющих цвета и один компонент `PopupMenu` всплывающего меню. При размещении компонента на форме в коде модуля появляется описание соответствующей объектной переменной (полное описание примера будет приведено в разделе с описанием компонентов меню). Для хранения строковых значений цвета используется свойство `Edit.Text`, для формирования надписи свойство `Label.Caption.`

Пример 2.3.3.-1. *Обработчик `FormShow` события `Show` главной формы*

```
unit Unit1;
interface
uses
   Windows, Messages, SysUtils, Variants, Classes, Graphics,
   Dialogs, Menus, ExtCtrls, StdCtrls, Controls, Forms;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Label1: TLabel;
    PopupMenu1: TPopupMenu;
    . . .
    procedure FormShow(Sender: TObject);//описание обработчика
    private    { Private declarations }
    public     { Public declarations }
  end;
var
```

```
  Form1: TForm1;
implementation
uses Unit2, Unit3;
{$R *.dfm}
procedure TForm1.FormShow(Sender: TObject);
begin // код обработчика события Show как метод класса TForm1
Form2.Show;
Form3.Show;
end;
end.
```

Логика данного приложения диктует необходимость визуализации всех форм в момент запуска приложения. Первая форма `Form1`, главная форма приложения, становится видимой автоматически в момент запуска приложения на выполнение. Сделать видимыми `Form2` и `Form3` необходимо в коде вручную. Для этих целей подойдет событие `Show` первой формы. Все обработчики событий в Delphi реализуются в виде процедур. Для создания шаблона процедуры обработчика события `Show` необходимо дважды щелкнуть мышкой в строке `OnShow` закладки `Events` инспектора объектов для объекта `Form1`. В инспекторе объектов появится имя процедуры `FormShow` (рис. 2.3), а шаблон модуля будет дополнен описанием процедуры `FormShow` как метода формы в интерфейсной части и заготовкой для тела процедуры в исполнимой части модуля формы.

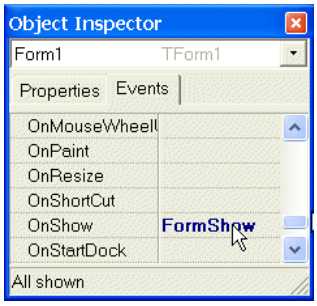

Рис. 2.3. Создание обработчика события

В теле процедуры необходимо разместить два оператора `Form2.Show; Form3.Show;` (вызов метода `Show` для `Form2` и `Form3`), выполнение которых сделает вторую и третью форму видимыми. Для ссылки на методы `Form2` и `Form3` из `Unit1` необходимо в `Unit1` добавить ссылку на модули, содержащие описания классов этих форм: `uses Unit2, Unit3.`

2.3.4. Операторы IS и AS

Оператор `IS` осуществляет проверку объекта на принадлежность указанному классу. Оператор `AS` приводит класс к указанному, если он совместим. Если после проверки объекта оператором `IS` определена принадлежность объекта классу, то применение оператора `AS` позволяет обращаться к свойствам этого класса. Такой синтаксис позволяет не использовать имя объекта. В заголовке процедур обработчиков событий всегда присутствует параметр `Sender: Tobject.` Этот параметр содержит класс объекта, инициировавшего данное событие. Используя операторы `IS` и `AS`, можно работать непосредственно с `Sender`, например:

```
If Sender is TEdit then
(Sender as TEdit).Text:='Univer';
```

Если установлена принадлежность объекта определенному классу, то приведение можно

сделать еще и так:

```
TEdit(Sender).Text:='Univer';
```

2.3.5. Модальные и немодальные окна

В примере 2.3.3.-1. был приведен код обработчика `FormShow` события `Show` формы `Form1`. В этом обработчике вызывался метод `Show` для второй и третьей формы, чтобы сделать их видимыми. Эти дочерние окна визуализировались в виде немодальных окон. При выполнении оператора `Form2.Show` управление передается обработчику события `Show` формы `Form2`, если он есть, выполняется код этого обработчика, форма визуализируется и продолжается выполнение процедуры `TForm1.FormShow` главной формы приложения. Это значит, что будет выполнен оператор `Form3.Show`, визуализирована третья форма и управление передано главной форме приложения. После выполнения обработчика на экране присутствуют три окна, причем активно (яркий заголовок окна) первое, принадлежащее главной форме приложения. Изменим код обработчика:

Пример2.3.5.-1. *Визуализация Form2 и Form3 как модальных окон*

```
procedure TForm1.FormShow(Sender: TObject);
begin
Form2.ShowModal;
Form3.ShowModal;
end;
```

Теперь для второй и третьей форм применяется метод `ShowModal`, который визуализирует формы как модальные окна. При выполнении оператора `Form2.ShowModal` окно второй формы становится видимым, управление передается `Form2`, выполнение процедуры `TForm1.FormShow` приостанавливается и может быть продолжено только, когда модальное окно формы `Form2` будет закрыто. При запуске приложения на выполнение с вызовом модальных форм на экране появляется одно активное окно второй формы. Когда пользователь закрывает его, – на экране активное окно третьей формы, и только при его закрытии на экране появляется активная главная форма приложения.

2.3.6. Обмен информацией между формами и модулями

Для того чтобы обратиться к объекту в пределах одного модуля, указывать имя владельца не нужно. Если объект `Edit1` является членом класса формы `Form1`, то обратиться к свойству `Edit1.Text` в пределах этого модуля можно одним из следующих способов:

```
Edit1.Text|Form1.Edit1.Text|self.Edit1.Text
```

Здесь зарезервированное слово `self` обозначает владельца объекта, к свойству которого идет обращение. Владельцем `Edit1` является форма `Form1`.

Для того чтобы обратиться к объекту `Form1` из другого модуля (например, из `Unit2`), прежде всего, нужно установить связь между модулями. Для этого можно в исполнимой (`implementation`) части `Unit2` разместить оператор `Uses Unit1`. Тогда Delphi «видит» в `Unit2` объекты формы `Form1` и в коде при написании `Form1` с точкой после имени, открывает список имен объектов в виде подсказки. Обращение к свойству объекта формы `Form1` из другого модуля должно представлять собой полное квалификационное имя: `Form1.Edit1.Text`.

Наличие оператора `Uses Unit1`, устанавливающего связь между двумя модулями позволяет также использовать в `Unit2` переменные и объекты `Unit1`, не принадлежащие `Form1`. В этом случае полное квалификационное имя выглядит: `Unit1.VariableName`. Delphi «видит» и такие объекты и переменные. Часто для

связи между модулями приложения используются глобальные переменные. Такие переменные можно, например, описать в модуле главной формы, а в остальных модулях в интерфейсной части при перечислении подключаемых системных модулей указать ссылку на этот модуль:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, . . ., Unit1;
```

Однако такой метод загромождает модуль главной формы, делая его плохо читаемым. Лучше для размещения описания глобальных переменных создать новый `Unit` без формы (`File→New→Unit`) и в каждом модуле приложения установить связь с ним. В такой модуль удобно поместить часто используемые процедуры и функции, не принадлежащие никакому классу формы.

Пример 2.3.6.-1. *Размещение описания глобальных переменных*

```
unit MyVar;
interface
uses   Windows, Messages, SysUtils, Classes;

procedure FindSeparator(S1:string;var S2:string);

type
 Elem=record             //описание глобальной структуры
  AN, AV, AP :single;
end;

TMyArr = class(TObject) //класс как глобальный объект
  mas: array of double;
  constructor Create (ArrX: Integer);
  procedure Fin;
end;

var  Spectr,  Angle,  Space,  Cover:  double;  //  глобальные
переменные

implementation

constructor TMyArr.Create(ArrX: Integer);
begin                          // конструктор класса
 inherited Create;
 SetLength(mas,ArrX);
end;

procedure TMyArr.Fin;
begin
 Finalize(mas);  // освобождение памяти массива
end;

procedure FindSeparator(S1:string;var S2:string);
 begin . . . end; // тело процедуры


begin
 Spectr:=0.1; Angle:=180; Space:=1.0e-4; Cover:=1;
          // инициализация глобальных переменных
end.

unit Unit1;
interface
```

```
uses   Windows, Messages, SysUtils, . . ., MyVar;
 . . .
end.
```

```
unit Unit2;
interface
uses  Windows, Messages, SysUtils, Variants, . . ., MyVar;
 . . .
end.
```

2.4. Работа с компонентами TEdit, TMainMenu, TPopUpMenu

2.4.1. Строка ввода Edit

Компонент `Edit` реализован классом `TEdit` и расположен на закладке Standard палитры компонентов среды разработки Delphi.

Некоторые свойства и события:

`property Text: TCaption;` – используется для ввода или вывода текста (не более 255 символов), т. е. это компонент для отображения одной строки.

`property MaxLength: Integer;` – максимальное количество символов, которое пользователь может ввести в поле элемента `Edit`.

`property ReadOnly: Boolean;` установка этого свойства в `True` запрещает изменять содержимое поля, фонт текста – основной, цвет фона задан значением свойства `Color`, объект может реагировать на мышь и события таймера.

`property Enabled: Boolean`; если свойство имеет значение `False` – объект не реагирует на события мыши, клавиатуры, таймера, поле недоступно, в этом случае фонт текста имеет цвет, установленный в Windows как неактивный для текста, цвет поля как неактивный для фона.

`property SelStart: Integer;` специфицирует порядковый номер позиции, в которой находится курсор, если текст не выделен, и первого символа выделенной в поле порции текста.

`property PasswordChar: Char;` – в этом свойстве указывается символ, какой будет отображаться в поле, когда в него осуществляется ввод. Если значение свойства = #0, вводимые символы визуализируются как есть, без замены. Это свойство компонента используется при создании формы для ввода пароля.

`property OnChange` – событие `Change` для `Edit` возникает каждый раз, когда изменяется значение поля. Событие не годится для проверки конечного содержимого поля, т.к. возникает при введении или стирании каждого символа поля.

`property OnEnter` – событие `Enter` возникает, когда элемент управления `Edit` получает фокус ввода.

`property OnExit` – событие `Exit` возникает для активного элемента управления, когда фокус ввода покидает его и переходит к другому элементу.

Пример 2.4.1.-1*. Подсветка активного элемента управления*

Форма содержит несколько полей ввода Edit, переход от поля к полю может осуществляться при помощи клавиши табуляции. Всем элементам назначен один обработчик для события Enter и один для Exit. Цвет поля активного элемента – светло-желтый, неактивного – белый.

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
 Edit1.Color := $00B7FFFF;
```

```
end;

procedure TForm1.Edit1Exit(Sender: TObject);
begin
 Edit1.Color := clWhite;
end;
```

В приведенном примере для объекта Edit1 созданы обработчики Edit1Enter и Edit1Exit. Для всех других объектов класса TEdit в режиме дизайна в Object Inspector на закладке Events для обработчиков OnEnter и OnExit назначены Edit1Enter и Edit1Exit соответственно.

property OnKeyPress – событие KeyPress возникает при наборе одиночного символа в поле любого компонента, обладающего фокусом ввода (Edit, Memo, ComboBox, StringGrid). К одиночным символам относятся любые символы, цифры, ввод, BaskSpace, пробел, но не функциональные клавиши, не Ctrl, Shift,Alt.

property OnKeyDown – событие KeyDown возникает, когда пользователь нажимает любой ключ на клавиатуре (символы, цифры, ввод, BaskSpace, пробел, любое функциональное сочетание Ctrl, Shift,Alt и т.д.) в то время, когда элемент имеет фокус ввода.

property OnKeyUp – событие KeyUp возникает, когда пользователь отпускает любой ключ на клавиатуре в то время, когда элемент имеет фокус ввода.

При создании обработчика OnKeyPress Delphi предоставляет шаблон процедуры, где заголовок процедуры выглядит так:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
```

Поскольку значение введенного ключа передается в процедуру по адресу, то в этом обработчике можно не только проанализировать вводимый символ, но и запретить неугодный.

Пример 2.4.1.-2. *Обработчик события KeyPress*

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var key: Char);
begin
if Sender is TEdit then begin
  if not((key in ['0'..'9']) or (key = chr(vk_back)) or
          (key=DecimalSeparator)) then
       key := #0;
  if (key = DecimalSeparator) then
   if (pos(key, (Sender as TEdit).Text)>0) then
      key := #0;
 end;   end;
```

2.4.2. Меню MainMenu, PopUpMenu

Компонент MainMenu реализован классом TMainMenu и расположен на закладке Standard палитры компонентов среды разработки Delphi. Компонент MainMenu описывает элементы главного меню. Каждый пункт меню реализован классом TMenuItem. Свойство меню Items представляет собой индексированный массив элементов меню, каждый из которых описывает индивидуальный элемент меню. К элементу меню можно обратиться как FirstItem := Menu1.Items.Items[0] или FirstItem := Menu1.Items[0].

Главное меню может быть на форме только одно.

Компонент PopupMenu реализован классом TPopupMenu и расположен на закладке Standard палитры компонентов среды разработки Delphi. Компонент PopupMenu описывает элементы всплывающего меню. Элементы всплывающего меню также как и

главного меню реализованы классом TMenuItem. Свойство Items также представляет собой индексированный массив элементов меню. Всплывающих меню на форме может быть сколько угодно. Любому компоненту формы и самой форме может быть назначено всплывающее меню, если используется свойство компонента PopupMenu. Установленное свойство AutoPopup (AutoPopup:=true) определяет будет ли автоматически появляться всплывающее меню, если пользователь щелкнул правой кнопкой мышки. А свойство PopupComponent определяет класс компонента, вызвавшего всплывающее меню.

2.5. Текстовые файлы, компоненты OpenDialog, SaveDialog

2.5.1. Текстовые файлы

Текстовые файлы характерны построчной записью данных в виде строк текста. В Delphi различают логическую файловую переменную и физический текстовый файл.
Var MyF: TextFile; – задает описание логической файловой переменной.
AssignFile (MyF, 'C:\ MyDi r\ MyFile.txt'); – связывает логическую файловую переменную с физическим файлом. Reset (MyF); – открывает файл на чтение, курсор устанавливается в начало файла. Rewrite (MyF) – файл открывается на запись, если в нем есть данные, они уничтожаются, курсор устанавливается в начало файла. Append (MyF); – файл открывается на запись, файл не очищается, курсор устанавливается в конец файла. CloseFile (MyF) – закрывает открытый файл, разрывает связь логической файловой переменной с физическим файлом.

Пример 2.5.1 -1. *Чтение строки файла с форматированием.*
Строка файла имеет формат:

ЧИСЛО1	*N пробелов или Tab*	*ЧИСЛО2*	*N пробелов или Tab*	*ЧИСЛО3*

Если в качестве десятичного разделителя используется точка, то следующий код позволяет последовательно прочитать три числа и представить их как числа с плавающей точкой

```
Var x, y1, y2 : real;
 Read (MyF, x);
 Read (MyF, y1);
 Readln (MyF, y2);
 CloseFile (MyF)
```

Первый оператор чтения Read (MyF, x) осуществляет чтение символов до первого пробела или табуляции, перевод полученной строки во Float, размещение числа в переменной x. Курсор остается на следующем символе после прочитанного. Выполнение оператора Read(MyF, y1) вызывает пропуск пробелов или/и символов табуляции до первого значащего символа, последовательное чтение символов до первого пробела или табуляции, перевод полученной строки во Float, размещение числа в переменной y1. Курсор остается на следующем, после прочитанного символе. Действие оператора Readln(MyF, y2) аналогично действию Read(MyF, y1), но число размещается в переменной y2, а курсор устанавливается на первом символе следующей строки.

Если не известно, сколько строк в текстовом файле, используют функцию EOF, которая принимает значение true, если достигнут конец файла. Для просмотра всех строк файла можно использовать такую языковую конструкцию:

```
While Not  Eof (MyF) do //читать пока не достигнут конец файла
    begin  . . .  end;
```

После использования логический файл должен быть закрыт процедурой CloseFile.

2.5.2. Диалоги OpenDialog, SaveDialog

Компоненты OpenDialog (класс TOpenDialog) и SaveDialog (класс TSaveDialog) отображают модальное окно Windows для выбора и открытия (выбора и сохранения) файла. Диалоговое окно во время выполнения программы не появляется до тех пор, пока не будет вызван метод Execute. Когда пользователь нажимает кнопку Open в случае открытия файла или Save в случае сохранения файла, диалоговое окно закрывается, имя выбранного файла запоминается в свойстве компонента Files. Метод Execute – основной метод работы с OpenDialog или SaveDialog, он реализован функцией и возвращает true, если пользователь выбрал файл и нажал кнопку Open / Save. Если пользователь нажал кнопку Cancel, метод Execute возвращает false.

Пример 2.5.2.-1. *Открытие текстового файла с использованием OpenDialog*

```
var   F: TextFile;   S: string;
begin
  if not OpenDialog1.Execute then exit; // файл не выбран
  AssignFile(F, OpenDialog1.FileName); //установить связь между
    //логической переменной F и выбранным файлом
  Reset(F); // открыть файл на чтение
  Readln(F, S); // прочитать первую строку файла
  Edit1.Text := S; //отобразить строку в объекте Edit
  CloseFile(F);
end;
```

Свойство DefaultExt компонентов OpenDialog и SaveDialog определяет расширение файла, принятое по умолчанию. Указанное в этом свойстве расширение файла добавляется автоматически к выбранному имени файла кроме тех случаев, когда расширение задается явно. Если пользователь выбрал имя файла с новым расширением, то DefaultExt будет добавлено к имени файла с этим новым расширением. Расширение длиннее, чем в три символа не поддерживается и не включает в себя точку. Filter определяет маску для файлов, доступных в OpenDialog и SaveDialog. Для задания маски в режиме разработки применяется специальное диалоговое окно. Также можно задавать фильтр в коде.

Пример 2.5.2.-2. *Фильтр для файлов результатов расчета*

```
OpenDialog1.Filter  :=  'Text  files  (*.txt)|*.TXT|Result  files
(*.res, *.clc, *.don)|*.RES;*.CLC;*.DON';
```

FileName содержит полное имя выделенного файла (имя + путь); FilterIndex определяет какой из типов файлов, прописанных в Filter выделяется по умолчанию, когда открывается диалоговое окно выбора файла; Files возвращает имя выбранного файла, как элемент списка; HistoryList содержит в виде списка историю выбранных файлов; InitialDir определяет текущую директорию, в которой открывается диалоговое окно; Title текст заголовка в диалоговом окне; Options определяет появление и поведение диалогового окна.

Пример 2.5.2.-3. *Фрагмент кода для чтения текстового файла*

```
unit Main;
. . .
var
 MainForm              : TMainForm;
 MyF                   : TextFile;
 ConfigDir, NameFile   :string;
. . .
```

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
 ConfigDir:= GetCurrentDir +'\Data'; // текущая директория
end;

TMainForm.OpenClick(Sender: TObject);
var
ext, s                  :string;
begin
 OpenDialog1.Filter:= 'My program data file (*.dat)|*.dat|' +
         'Text file (*.txt)|*.txt';
 OpenDialog1.InitialDir:=ConfigDir;
 If Not(OpenDialog1.Execute) Then Exit
  AssignFile(MyF,OpenDialog1.FileName);
  reset(MyF);
   try
     ReadLn(MyF,s);
     except begin
       ShowMessage('Problem with data file');
       CloseFile(MyF); Exit;
     end; // except
   end;  // try
  if not (s = 'It is signature for my data file') then
    begin
     ShowMessage('It is not my program data file');
      CloseFile(MyF); Exit;
    end;
  NameFile:=OpenDialog1.FileName;
  ConfigDir:=ExtractFileDir(OpenDialog1.FileName);
  ext:=ExtractFileExt(OpenDialog1.FileName);
  . . . // processing
 CloseFile(MyF);
end;
```

В примере считывается текстовый файл с записанными данными программы. Стандартное расширение файла – `*.dat`. В первой строке файла находится служебная информация. Если текстовый файл сохранен на диске средствами самой программы – служебная информация формируется программно, если пользователь самостоятельно создал файл, он обязан сформировать служебную строку. Эта строка – подпись файла, при чтении файла осуществляется проверка на наличие этой служебной строки. Если не возникла исключительная ситуация при чтении файла и это файл для нашей программы, может быть осуществлена дальнейшая обработка. В этом случае полное имя файла, расширение файла и директория запоминаются в глобальных переменных. При последующих операциях чтения-записи диалоговое окно будет открыто в той директории, с файлом из которой осуществлялась работа в предыдущей операции чтения-записи.

2.6. Сетка StringGrid

Свойства: `ColCount` – количество столбцов таблицы. `RowCount` – количество строк таблицы. `FixedCols` – количество фиксированных, не скроллируемых столбцов таблицы. `FixedRows` – количество фиксированных, не скроллируемых строк таблицы. У `StringGrid` должен быть хотя бы один не фиксированный столбец и строка. `DefaultColWidth` – ширина столбика в пикселях по умолчанию. `DefaultRowHeight` – высота строк в пикселях по умолчанию. `ColWidths[Index: Longint]`– ширина

указанного столбика. RowHeights[Index: Longint]– высота указанной строки. Cells[ACol, ARow: Integer] – содержимое указанной ячейки, где ACol – номер столбца (от 0 до ColCount-1), ARow – номер строки (от 0 до RowCount-1). Col – номер столбика выделенной ячейки. Row – номер строки выделенной ячейки.
Options – опции таблицы. Установка опции goEditing в true допускает редактирование пользователем содержимого ячеек, goDrawFocusSelected – активная ячейка не выделяется контрастным цветом.
Свойство Selection реализовано классом TGridRect и указывает границы текущего выделения ячеек таблицы. Класс TGridRect наследник класса TRect:

```
type TGridRect = record
Left, Top, Right, Bottom: Longint;
End;
```

Пример 2.6.-1. *Программно выделить в таблице прямоугольник с 1 строки по 4 и со 2 столбика по 3.*

```
procedure TForm1.Button1Click(Sender: TObject);
var     myRect: TGridRect;
begin
  myRect.Left  := 2;        myRect.Top    := 1;
  myRect.Right := 3;        myRect.Bottom := 4;
  StringGrid1.Selection := myRect;
end;
```

Пример 2.6.-2. *В таблице, открытой для редактирования, у которой одна фиксированная строка, запретить изменение данных в первом столбце и разрешить ввод чисел с десятичным разделителем во втором столбце.*

```
 procedure TForm1.StringGrid1KeyPress(Sender: TObject; var Key:
                                      Char);
var Mcol,Mrow:integer;
begin
 Mcol:=(Sender as TStringGrid).Col;
 Mrow:=(Sender as TStringGrid).Row;
 if  Mcol=0 then
  Key:=#0
 else
  if  Not ((Key in ['0'..'9']) or  (Key=Chr(vk_Back))
   or (Key=DecimalSeparator) ) then
    Key:=#0
  else begin
   if Key = DecimalSeparator then
    if  Pos (DecimalSeparator,
    (Sender as TStringGrid).Cells[Mcol,Mrow])<>0 then
      Key:=#0;
  end;
end;
```

Пример 2.6.-3. *Вставить пустую строку после выделенной ячейки.*

```
procedure TForm1.Insert1Click(Sender: TObject);
var i,j: integer;
begin
 Tabl.RowCount:=Tabl.RowCount+1;//к-во строк на 1 больше
 for i:= Tabl.RowCount-1 downto   Tabl.Row+2 do
                          //сдвиг строк с конца
```

```
  Tabl.Rows[i]:=Tabl.Rows[i-1];
 for i:=0 to Tabl.ColCount do //очистка вставленной строки
  Tabl.Cells[i, Tabl.Row+1]:='';
end;
```

2.7. Компоненты-контейнеры TPanel , TGroupBox, TradioGroup

`property ControlCount: Integer;` – количество элементов управления, расположенных на родительском элементе управления.

`property Controls[Index: Integer]: TControl;` – доступ к элементу управления по индексу.

`property Parent: TWinControl;` – объект родитель для этого элемента управления.

Пример 2.7.-1. *Использование свойства Parent.*
Есть три панели с надписями, расположенные на форме, и есть отдельная Panel4 пустая панель. Щелчок по панелям 1, 2, 3 меняет их родителя. Визуально это выглядит так: панель, по которой щелкнули (1, 2, 3), перемещается на панель 4, если ранее была расположена на форме, или на форму, если она находилась на панели. Обработчик Panel1Click назначается как обработчик события OnClick для объектов Panel1, Panel2, Panel3.

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
 if Sender is TPanel then
 begin
  If (Sender as TPanel).Parent=Form1  then
  begin
   (Sender as TPanel).Parent:=Panel4;    Exit;
  end;
  If (Sender as TPanel).Parent=Panel4  then
  begin
   (Sender as TPanel).Parent:=Form1;  Exit;
  end;
 end;
end;
```

2.8. Компоненты CheckBox, RadioButton

Компонент `CheckBox` применяется для выбора опций. Если на компоненте родителе присутствуют несколько `CheckBox`, то может быть отмечено любое их количество, хоть все, или ни одного.

Основное свойство: `property Checked: Boolean`; позволяет определить находится ли `CheckBox` в состоянии `Checked`. Используется в случае, когда `AllowGrayed = false`. Если `AllowGrayed = true` – возможно три состояния: `checked, unchecked, grayed`.

`RadioButton` – радио кнопки характерны тем, что в группе кнопок, принадлежащей одному родителю, может быть отмечена только одна кнопка. Активизация другой радио кнопки делает ее отмеченной и снимает отметку с предыдущей. Групп радио кнопок может быть сколько угодно, но каждая группа должна принадлежать одному родителю (панели, форме, элементу `RadioGroup`). Свойство `Checked` позволяет определить отмечена ли кнопка.

2.9. Кнопки Button, BitBtn, SpeedButton

Чтобы использовать кнопку только с надписью – используют Button, чтобы использовать картинку bitmap вместо или вместе с надписью – используют BitBtn. Чтобы использовать кнопку, которая сохраняет нажатое состояние или поместить ее в ToolBar – используют SpeedButton.

Свойства, общие для кнопок Button и BitBtn: property Default: Boolean; – определяет, будет ли вызываться обработчик события Click, когда пользователь нажал клавишу Enter . Кнопка ВВОД ПО УМОЛЧАНИЮ может быть на форме только одна, если нескольким кнопкам установлено свойство Default=true, то кнопкой ВВОД ПО УМОЛЧАНИЮ будет только та видимая кнопка, у которой свойство TabOrder имеет меньшее значение. property Cancel: Boolean; – определяет, будет ли вызываться обработчик события Click, когда пользователь нажал клавишу Escape . Кнопка ОТМЕНА ПО УМОЛЧАНИЮ может быть на форме только одна, если нескольким кнопкам установлено свойство Cancel=true, то кнопкой ОТМЕНА ПО УМОЛЧАНИЮ будет только та видимая кнопка, у которой свойство TabOrder имеет меньшее значение.

Отличия. Кнопка Button не позволяет иметь цветную надпись (Font.Color= clBlack);

Кнопка BitBtn имеет предопределенные обработчики событий, которые можно задать в свойстве property Kind: TBitBtnKind. При установке этого свойства кнопка выполняет действия автоматически, по умолчанию установлен тип кнопки bkCustom (предопределенного обработчика нет).

Свойства, общие для кнопок BitBtn и SpeedButton: property Glyph: TBitmap – назначает рисунок, который будет появляться на поле кнопки. Глиф может содержать до 4 простых изображений, все изображения должны быть одного и того же размера и следовать друг за другом. Кнопка использует одно из этих изображений в зависимости от своего состояния.

Первое (Up) – изображение появляется, когда кнопка не нажата (если в глифе нет других изображений – отображается во всех состояниях).

Второе (Disabled) – изображение появляется, когда кнопка недоступна.

Третье (Clicked) – изображение появляется, когда нажали, но еще не отпустили кнопку.

Четвертое (Down) – изображение появляется, когда кнопка зафиксирована в нажатом состоянии.

Если bitmap для глифа содержит мультиизображения – их количество указывается в свойстве NumGlyphs.

Отличия. Кнопки SpeedButton могут функционировать как группа. Обычно они размещаются на панели инструментов приложения. Чтобы сделать быстрые кнопки работающими как группа необходимо свойству GroupIndex назначить любое не нулевое значение. GroupIndex =0 – кнопки работают независимо, не как группа. Кнопки со свойством, например, GroupIndex = 1 образуют одну группу, со свойством GroupIndex = 2 образуют другую группу и т. д. По умолчанию быстрые кнопки появляются в не нажатом состоянии. Если GroupIndex > 0 , установка свойства Down = true выделяет кнопку. Свойство AllowAllUp = True – в группе кнопок может отсутствовать нажатая кнопка, AllowAllUp = False – группа быстрых кнопок работает как радио группа, т. е. в один момент времени может быть нажата только одна кнопка. Свойство Flat определяет, будет ли автоматически прорисовываться 3D бордюр вокруг кнопки, который обеспечивает нажатый и отжатый вид кнопки. Свойство Transparent = true задает прозрачный фон для кнопки.

Замечание: главная особенность всех видов кнопок – фон кнопки всегда такой,

как принято в установках Windows для используемого компьютера. Цветную кнопку можно создать, не разрабатывая нового класса, а используя уже известные компоненты. Для этих целей может подойти Panel и ее свойства BevelInner и BevelOuter.

2.10. Списки ListBox, Memo, ComboBox

ListBox реализован классом TListBox и представляет собой не редактируемый список. Свойство Items: TStrings; содержит строки, которые появляются в списке. Свойство опубликовано, доступно как во время разработки, так и во время выполнения. Свойство реализовано классом TStrings и является векторным. Сам класс TStrings имеет свойство Count: Integer – количество строк списка и векторное свойство Strings[Index: Integer]: string, которое обеспечивает доступ к строке по индексу. Поэтому к содержимому первой строки списка (индексация от 0) можно обратиться так:
ListBox1.Items. Strings[0]; или ListBox2.Items[0].
property ItemIndex: Integer – порядковый номер выделенной строки списка. Если выделена первая строка, значение свойства = 0. Если не выделена ни одна строка, значение свойства = - 1. Свойство Selected[Index: Integer]: Boolean имеет значение true для указанного номера строки, если строка выделена и false, если нет выделения.

Пример 2.10.-1. *Поиск строки, выделенной в списке*

```
for i := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected[i] then
    . . .
```

Методы списков реализованы тоже классом TStrings:
ListBox1.Items.Add(String); – добавить строку с конец списка. ListBox2.Items.Delete(Index) – уничтожить строку с указанным индексом. ListBox2.Items.Insert(Index, String) – вставить строку в указанное место списка. ListBox1.Items.Move(CurIndex, NewIndex) – переместить строку с индексом CurIndex на новое место, определенное индексом NewIndex.

Пример 2.10.-2. *При щелчке мышью на выделенной строке списка ListBox2 – добавить эту строку к списку ListBox1, а в с писке ListBox2 – уничтожить.*

```
procedure TForm1.ListBox2Click(Sender: TObject);
begin
  ListBox1.Items.Add(ListBox2.Items[ListBox2.ItemIndex]);
  ListBox2.Items.Delete(ListBox2.ItemIndex);
end;
```

Компонент Memo реализован классом TMemo и представляет собой многострочный редактируемый список. По свойствам и методам он мало отличается от TListBox. Главное отличие – не имеет свойства Items, аналогичное по назначению свойство для TMemo называется Lines: TStrings. Поскольку это редактируемый список, то компонент имеет свойство SelStart – позиция курсора в невыделенном тексте или индекс первого символа в выделенном тексте, такое же, как и для Edit.

Компонент ComboBox реализован классом TComboBox и представляет собой список + поле ввода, т. е. компонент сочетает в себе свойства и методы TListBox и TEdit. Свойство AutoComplete (true/false) определяет возможность поиска в списке по первой, введенной пользователем, букве, а свойство Style определяет внешний вид и поведение компонента.

2.11. График Chart

Компонент Объект Chart реализован классом TChart. Это наиболее важный компонент из библиотеки TeeChart. TChart – наследник класса TPanel, он обладает всеми возможностями родителя плюс много новых возможностей, связанных с графикой.
Метод класса TChart: function ChartRegionRect : TRect; – прямоугольник Chart, ограниченный 4 осями.

Пример 2.11.-1: *Определить находится ли точка с координатами x,y внутри прямоугольника Chart*

```
if  PtInRect( ChartRegionRect, Point( x,y ) ) then …
```

В этом примере использована функция PtInRect (модуль Types), определяющая, лежит ли точка внутри указанного прямоугольника. Метод, реализованный процедурой CopyToClipboardBitmap, копирует всю область Chart в буфер обмена в Bitmap формате. Процедура SaveToBitmapFile(Const FileName : String) позволяет сохранить текущее изображение Chart в файл *. BMP. Свойство View3d : Boolean; определяет 3D/2D вид графика. Chart – родительский компонент для класса TChartSeries. Каждая диаграмма на графике реализуется в виде серии точек с использованием класса TChartSeries. Каждая серия имеет свое собственное имя, к серии в коде программы можно обратиться по ее имени или по индексу, используя индексное свойство Series[Index:Longint]:TChartSeries компонента Chart. Например, очистка первой серии Series1.Clear; или Chart1.Series[0].Clear. Перед каждой перерисовкой компонента Chart серии необходимо чистить.
На графике могут быть представлены различные виды диаграмм, для этих целей класс TChartSeries имеет различные типы, например; Line (TLineSeries), Area (TAreaSeries) и т. д. Свойства класса TChartSeries можно задать в режиме разработки в диалоговом окне редактора графика, используя закладку Series, или во время выполнения программы. Использование во время выполнения программы свойств XValues и YValues позволяет обратиться к каждому X, Y значению точек конкретной серии графика**.** Использование свойства Active позволяет визуализировать или скрыть серию. Даже если серия скрыта, ее точки не нуждаются в перезаполнении.
Метод класса TChartSeries, реализованный функцией:

```
function AddY(Const AYValue: Double; Const AXLabel: String;
AColor: TColor): Longint;
```

позволяет занести точку в серию. Метод AddY может быть использован, чтобы вставить новую точку в серию, когда значение X не известно. Новая точка будет иметь только Y значение. X значение вычисляется автоматически. AXLabel может быть пустым, AColor может быть пустым, задавать конкретное значение цвета или иметь значение clTeeColor (использовать значение цвета, заданное при разработке). Функция возвращает индекс новой точки в серии.
Метод класса TChartSeries, реализованный функцией:

```
AddXY(Const AXValue, AYValue: Double; Const AXLabel: String;
AColor: TColor)
```

позволяет занести в серию точку с координатами X, Y**.**

Пример 2.11.-2. *График визуализирует одну кривую. При щелчке правой кнопкой мыши на графике вызывается всплывающее меню. Пункты: 1) Изменить вид графика (2d / 3d) 2) Копировать изображение графика в буфер обмена 3) Копировать данные графика в буфер обмена*

```
unit Unit1;
interface
uses . . ., TeeProcs, Chart, ClipBrd;
. . .
procedure TForm1.Plot2d3dClick(Sender: TObject);
begin
 Chart1.View3D:=  not Chart1.View3D;
end;
procedure TForm1.ImageToClipClick(Sender: TObject);
 var   MyColor:TColor;
begin
 MyColor:= Chart1.Color;
 Chart1.Color:=clWhite;
 Chart1.CopyToClipboardBitmap;
 Chart1.Color:= MyColor;
end;
procedure TForm1.DataToClip(Sender:TObject);
var  i,j  :integer; s  :string;
begin
 s:='';
 s:=' X '+#9+' Y '+#13+#10;
 for i:=0 to Chart1.Series[0].Count-1 do
  begin
   s:=s+FloatToStrF(Chart1.Series[0].XValue[i],ffFixed,6,4)+#9
   +FloatToStrF(Chart1.Series[0].yValue[i],ffFixed,6,4)+#13+#10;
  end;
 Clipboard.Open;
 Clipboard.Clear;
 Clipboard.AsText:=s;
 Clipboard.Close;
end;
```

В примере перед тем, как отправить изображение графика в буфер обмена, для графика устанавливается белый цвет фона, после копирования в буфер обмена цвет графика восстанавливается. Чтобы передать данные в буфер обмена формируется текстовая строка, в которой каждая точка представлена координатами `X` и `Y`, разделенными символом табуляции**.** Точки между собой разделены символом перевода строки. После того как строка сформирована, открывается и очищается буфер обмена, после чего ему назначаются текстовые данные, и буфер обмена закрывается.

Вопросы для самоконтроля

1. Основные характеристики объектно-ориентированного программирования.
2. Что такое компоненты?
3. Какие типы переменных используются в Delphi?
4. Рассказать как работает оператор цикла `for`.Привести примеры.
5. Рассказать как работает операторы цикла `repeat, while`.Привести примеры.
6. Как осуществляется передача параметров в процедуры и функции?
7. Какая последовательность событий от создания до визуализации формы?
8. Примеры обработчиков событий.
9. Как работают операторы IS и AS?
10. Рассказать, чем отличаются модальные и немодальные окна.

11. Каким образом можно запретить ввод нецифровых символов в поле ввода?
12. Как вывести в компонент-сетку информацию?
13. Как прочитать текстовый файл?
14. Каким образом можно визуализировать данные на графике?
15. Что такое КНОПКА ПО УМОЛЧАНИЮ?
16. Чем отличаются CheckBox и RadioButton?

Литература к разделу 2

1. Сван Том. Delphi 4. Библия разработчика: Пер. с англ. – К.; М.; СПб. : Диалектика, 1998. – 672 с.
2. Флёнов М. Е. Библия Delphi. – СПб. : БХВ-Петербург, 2007. – 880 с.
3. Флёнов М. Е. Delphi в шутку и всерьёз: что умеют хакеры. – СПб. : Питер, 2006. – 270 с.
4. Флёнов М. Е. Delphi 2005. Секреты программирования. – СПб. : Питер, 2006. – 266 с.
5. Пашеку Хавьер. Программирование в Borland Delphi 2006 для профессионалов. : Пер. с англ. – М. : Издательский дом "Вильямс", 2006. – 944 с.
6. Хладни Иван. Внутренний мир Borland Delphi 2006. : Пер. с англ. – М. : Издательский дом "Вильямс", 2006. – 768 с.
7. Delphi Russian Knowledge Base, http://www.forum.sources.ru
8. Delphi Frequently Asked Question, http://www.bnd.borland.com
9. Программирование на Delphi, http://www.softera.ru