

Глава 7. Палитра компонентов Standard.....	99
7.1 Кнопка (TButton).....	99
7.2 Играем со свойствами кнопки (логические операции).	102
7.3 Надписи (TLabel).....	105
7.4 Строки ввода (TEdit).....	107
7.5 Многострочное поле ввода (TMemo).....	109
7.6 Объект TStrings.....	113
7.7 CheckBox.....	114
7.8 Панели (TPanel).....	116
7.9 Кнопки выбора TRadioButton.	118
7.10 Списки выбора (TListBox).....	119
7.11 Выпадающие списки (TComboBox).....	121
7.12 Полосы прокрутки (TScrollBar).....	122
7.13 Группировка объектов (GroupBox).....	123
7.14 Группа компонентов RadioButton (TRadioGroup).....	124
7.15 Ответы на вопросы.....	126

Глава 7. Палитра компонентов Standard.

В этой главе моей книги я буду рассказывать о компонентах, которые находятся на закладке Standard палитры компонентов. Одновременно с этим мы будем писать программы с использованием этих компонентов, и изучать язык программирования Delphi.

Здесь я не буду просто перечислять компоненты и их свойства. Мы будем писать вполне работающее приложение. Функциональность их пока будет очень слабая, но всё же программы будут вполне рабочими. Я постараюсь писать примеры как можно интереснее, но это уже больше зависит от самих компонентов.

Несмотря на то, что мне ещё надо рассказать немного про основы языка программирования Delphi, я решил сначала рассказать про компоненты, чтобы мы хоть немного попрактиковались в программировании. Ну а потом я расскажу уже последнюю часть теории самого языка Delphi. После этого нам останется только изучать компоненты и различные технологии.

Я ещё раз прошу тебя, повторяй все действия за мной самостоятельно. Я понимаю, что на диске есть все исходники моих примеров, но это мои исходники. Ты сможешь чему-то научиться, только если сам попробуешь. Поэтому я стараюсь дать максимальное количество практики.




Рис 7.1 Палитра компонентов, закладка «Standard».

Итак, переходим к рассмотрению компонента кнопка. Хотя он находится в середине закладки, я решил начать с него, потому что он проще и при рассмотрении других компонентов мы будем постоянно использовать эту кнопку.

7.1 Кнопка (TButton).

Кнопка в Delphi происходит от объекта *TButton*. Когда ты устанавливаешь на форму новую кнопку, то ей даётся имя по умолчанию *Button1*.

Давай напишем маленькую программу с использованием кнопки. Для этого создай новое приложение. Теперь щёлкни по изображению кнопки  на палитре компонентов. После этого щёлкни по форме в любом месте. На форме сразу же появится кнопка с заголовком *Button1*.

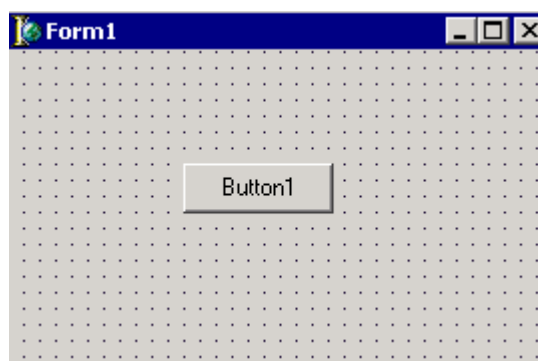


Рис 7.1.1 Форма с кнопкой.

Есть ещё один способ установить кнопку на форму – дважды щёлкнуть по изображению кнопки. Но в этом случае, кнопка окажется в центре формы, а не там, где мы хотим.

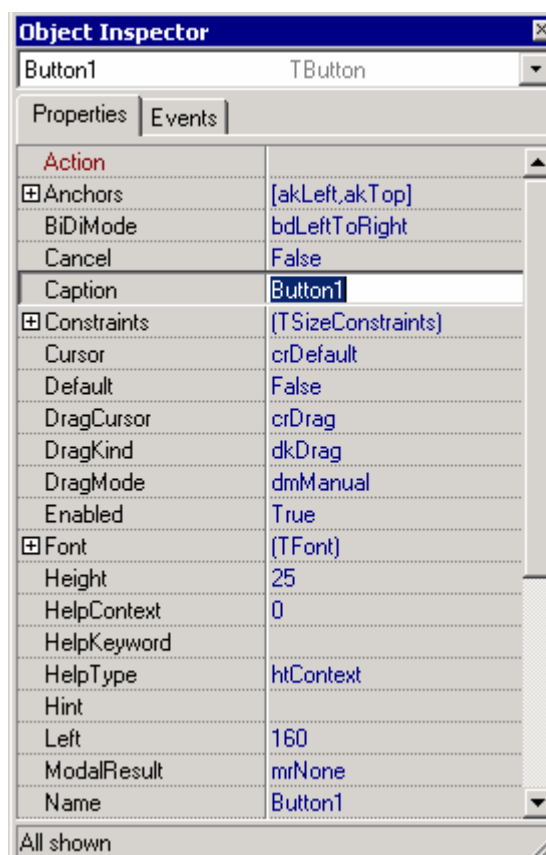


Рис 7.1.2 Объектный инспектор со свойствами кнопки.

Выдели кнопку и перейди в объектный инспектор. Сейчас здесь показаны свойства кнопки. Оглянись. Как видишь, большинство свойств нам уже знакомо, по свойствам формы, поэтому я не буду их расписывать. Есть только одно новое свойство – *ModalResult*, но мы с ним познакомимся позже.

Давай изменим заголовок кнопки. За заголовок формы у нас отвечало свойство *Caption*. Здесь то же самое. Найди свойство *Caption* и измени содержащийся там текст на «Нажми меня».

Давай сразу изменим свойство *Name* у кнопки. Я говорил, что любым компонентам и формам лучше давать понятные имена, чтобы не было бардака. Так давай с самого начала будем привыкать к нормальной жизни. Найди свойство *Name* и измени его на ***MyFirstButton***. Пускай имя кнопки пока не отражает никакого смысла, ведь она ещё ничего не делает.

Давай также изменим имя формы. Для этого сними выделение с кнопки (щёлкни в любом месте формы). Вверху окна, должна загореться надпись *Form1 TForm1*, как на рис 7.1.3.

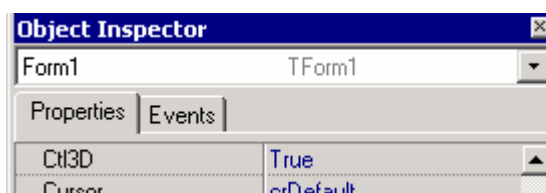


Рис 7.1.3 Объектный инспектор со свойствами кнопки.

Теперь найди здесь свойство *Name* (оно должно быть равно *Form1*) и измени его значение на *MainForm* (это переводится как - главная форма).

Попробуй запустить программу (нажми F9). Ты можешь спокойно нажимать на кнопку, только ничего не происходит.

Давай усложним наш пример и поймем событие, когда нажимается кнопка. Для этого перейди на закладку Events. Когда мы рассматривали события формы, я говорил, что за клик мышкой отвечает событие *OnClick*. Для кнопки есть такое же событие. Найди его и щёлкни по нему дважды. Delphi должен создать в редакторе кода процедуру - обработчик события *OnClick*. По умолчанию ей даётся имя в виде имени компонента (нашей кнопки) плюс имя события без приставки *On*. В нашем случае получается, что имя процедуры обработчика получается *MyFirstButtonClick*.

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

end;
```

В объектном инспекторе, напротив строки *OnClick* тоже должно появиться имя процедуры обработчика (смотри рисунок 7.1.4).

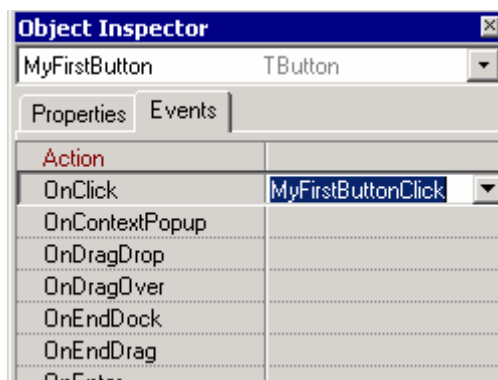


Рис 7.1.4 Закладка Events

Давай вернёмся в редактор кода и посмотрим, что там создал для нас Delphi? Это процедура *MyFirstButtonClick*. Ей передаётся один параметр *Sender* объектного типа *TObject*. При начале выполнения процедуры, в переменной *Sender* будет находиться указатель на объект, который вызвал этот обработчик. Это очень важно, потому что одна процедура обработчик может обрабатывать нажатия сразу нескольких кнопок. Или вообще, компоненты разного типа. По содержимому этой переменной можно узнать, какой именно компонент сгенерировал событие. Дальше мы будем использовать эту возможность, когда это понадобится.

Давай напишем внутри процедуры (между *begin* и *end*) команду *Close*. Эта команда закрывает окно. Теперь наша процедура должна выглядеть так:

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

Попробуй запустить программу и нажать на кнопку. Программа закроется.

 На компакт диске, в директории \Примеры\Глава 7\Button ты можешь увидеть пример программы.

Откуда я взял, что команда **Close** закрывает нашу программу? А оттуда, что это метод формы. Мы могли просто написать так:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Form1.Close;
end;
```

Это то же самое. Разницы абсолютно никакой нет. Так почему же я написал просто *Close*. Да потому что процедура относится к объекту *Form1* и внутри неё я имею право использовать его свойства и метода без указания владельца. По умолчанию будет браться *Form1*. Но если я захочу из этой процедуры закрыть другую форму, например *Form2*, то мне придётся указать, что я хочу именно *Form2*.

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Form2.Close;
end;
```

Если я просто напишу *Close*, то закроется *Form1*, а не *Form2*.

7.2 Играем со свойствами кнопки (логические операции).

Этот пример я уже описывал в журнале «Хакер». Он достаточно простой и удобный для обучения. Сейчас я снова напишу программу, у которой будет только одна кнопка. При наведении на неё мышкой, кнопка будет убегать.

Воспользуемся предыдущим примером и улучшим его. Для начала выдели форму и измени свойство *AutoScroll* на *False*, чтобы на форме не появлялись автоматически полосы прокрутки.

Теперь создай для кнопки обработчик события *OnMouseMove*. Для этого выдели кнопку и перейди в объектном инспекторе на закладку *Events*. Здесь ты уже создавал обработчик *OnClick*, теперь щёлкни дважды напротив строки *OnMouseMove*, чтобы создать соответствующий обработчик.

Если ты всё сделал правильно, то Delphi должен создать следующую процедуру для обработки сообщения *OnMouseMove*.

```
procedure TForm1.MyFirstButtonMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
end;
```

Напиши здесь следующее:

```
procedure TForm1.MyFirstButtonMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  index: integer;
begin
  index:=random(4);
  case index of
    0: MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width;
    1: MyFirstButton.Left:=MyFirstButton.Left-MyFirstButton.Width;
    2: MyFirstButton.Top:=MyFirstButton.Top+MyFirstButton.Height;
    3: MyFirstButton.Top:=MyFirstButton.Top-MyFirstButton.Height;
  end;


  if MyFirstButton.Left<0 then
    MyFirstButton.Left:=0;

  if (MyFirstButton.Left+MyFirstButton.Width)>Form1.Width then
    MyFirstButton.Left:=Form1.Width-MyFirstButton.Width;

  if MyFirstButton.Top<0 then
    MyFirstButton.Top:=0;

  if (MyFirstButton.Top+MyFirstButton.Height)>Form1.Height then
    MyFirstButton.Top:=Form1.Height-MyFirstButton.Height;
  end;
```

Пока просто перепиши содержимое этого листинга. Скоро мы подробно рассмотрим, что тут написано. Запусти программу и попробуй нажать на кнопку. Как только ты попытаешь навести на неё мышкой, кнопка будет убегать от тебя.

 На компакт диске, в директории \Примеры\Глава 7\Button1 ты можешь увидеть пример этой программы.

Обязательно сначала посмотри, как работает пример. Когда наиграешься, возвращаясь к чтению книги, и мы рассмотрим исходник.

В разделе **Var** я объявил одну переменную *index* типа целое число. В первой строчке я присваиваю этой переменной случайное число с помощью функции *random*:

```
index:=random(4);
```

Функция *random* возвращает случайное число. В качестве единственного параметра ей нужно передать число, которое будет означать максимально возможное случайное число. Я передаю цифру 4. Это значит, что функция вернёт мне число от нуля до четырёх ($0 \leq X < 4$). Само число 4 в диапазон возможных значений не входит, все случайные числа будут меньше него.

После этого я проверяю, какое число мне сгенерировала функция *random* с помощью конструкции:

```
case Переменной of
```

```
Значение1: Действие1;  
Значение2: Действие2;  
...  
...  
end;
```

Конструкция **Case** сравнивает переменную с перечисленными между ключевыми словами **of** и **end** значениями и если одно из них совпадает, то выполняет соответствующее действие. Например, допустим, что наша переменная равна числу «Значение2». В этом случае будет выполнено «Действие2». При этом «Действие1» и другие выполняться не будут.

Если тебе нужно, чтобы при равенстве значений выполнялось несколько действий, то необходимо заключить их в логические кавычки **Begin ... end**. Например:

```
case Переменной of  
Значение1:  
  Begin  
    Действие1_1;  
    Действие1_2;  
    ...  
  End;  
  
Значение2:  
  Begin  
    Действие2_1;  
    Действие2_2;  
    ...  
  End;  
  ...  
  ...  
end;
```

Теперь вернёмся к нашему примеру. В нём используется следующий **Case**:

```
case index of  
0: MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width;  
1: MyFirstButton.Left:=MyFirstButton.Left-MyFirstButton.Width;  
2: MyFirstButton.Top:=MyFirstButton.Top+MyFirstButton.Height;  
3: MyFirstButton.Top:=MyFirstButton.Top-MyFirstButton.Height;  
end;
```

Если переменная *Index* равна 0, то выполнится следующее действие:

MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width

Что это? Здесь я присваиваю свойству *Left* (левая позиция) кнопки *MyFirstButton* значение левая позиция этой же кнопки плюс её ширина. Это значит, если ты попытался навести мышкой на кнопку, и функция *Random* сгенерировала 0, то левое значение кнопки будет увеличено на ширину кнопки. А это значит, что кнопка сдвинется от тебя вправо. Если ты уже запускал пример, то ты уже понял меня.

Если значение переменной *Index* равно 1, то я наоборот, уменьшаю левую позицию кнопки на её ширину. А это значит, что кнопка убежит влево. Если значение переменной *Index* равно 2, то я увеличиваю верхнюю позицию кнопки на её высоту. А это значит, что кнопка убежит вниз. Надеюсь, что смысл понятен. Давай двигаться дальше.

После конструкции *case .. of .. end* идёт проверка: «Не убежала ли кнопка за пределы окна. Сначала я проверяю левую позицию кнопки:

```
if MyFirstButton.Left<0 then  
  MyFirstButton.Left:=0;
```

Здесь идёт проверка, если левая позиция кнопки (**MyFirstButton.Left**) меньше нуля, то установить её в ноль.

В следующей строке я проверяю, если левая позиция кнопки плюс её ширина больше ширины окна, то левой позиции присвоить значение «ширина окна» минус «ширина кнопки»:

```
if (MyFirstButton.Left+MyFirstButton.Width)>Form1.Width then  
  MyFirstButton.Left:=Form1.Width-MyFirstButton.Width;
```

Точно так же я проверяю и верхнюю позицию, чтобы она не вылезла за пределы окна.

В этой проверке я использую конструкцию *if .. then*. Я уже говорил о ней в теории, а теперь поговорим на практике. Я вообще буду иногда повторяться, потому что в кодинге это полезно.

Итак, конструкция *if .. then* выглядит так:

```
If Условие then  
  Выполнить действие;
```

Если условие выполнено, то будет выполнено одно следующее действие. Если ты хочешь выполнить два действия, то должен заключить их в логические скобки **Begin ... end**. Например:

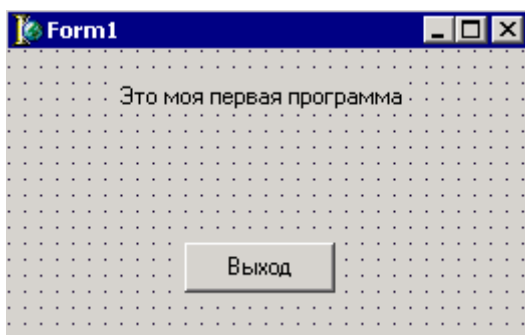
```
If Условие then  
  Begin  
    Действие1;  
    Действие2;  
    Действие3;  
    ...  
  End;
```

В этом случае все действия между **Begin** и **end** будут выполнены, если условие верно. Таким образом, **Begin ... end** группирует последовательность действий в одно.

7.3 Надписи (TLabel).

Этим компонентом мы будем пользоваться практически в каждом примере, выводя надписи для других компонентов.

Создай новое приложение. Брось на форму один компонент TLabel **A** и измени у него свойство **Caption** на «Это моя первая программа». У тебя должно получиться нечто подобное.



 На компакт диске, в директории \Примеры\Глава 7\Label ты можешь увидеть пример этой программы.

Но это слишком просто. Давай я тебе покажу, как сделать с помощью этого компонента очень красивый визуальный эффект:

Щёлкни дважды по свойству *Font* компонента *Label1*. Перед тобой появится окно свойств шрифта. Сделай шрифт побольше и измени цвет на белый.

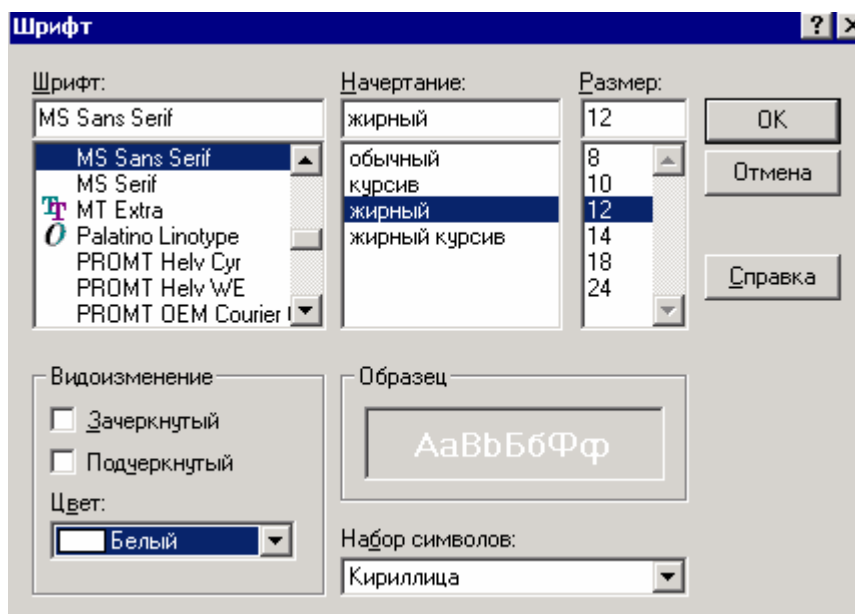


Рис 7.3.1 Изменение свойств шрифта

Теперь запомни значения свойств *Left* и *Top* компонента. У меня это 16 и 16. Теперь скопируй этот компонент в буфер обмена (меню Edit->Copy). Щёлкни по форме, чтобы снять выделение с компонента *Label1*. Теперь вставь копию компонента (меню Edit->Paste). Delphi создаст новый компонент *Label2*, который будет копией первого. Выдели новый компонент и измени свойства *Left* и *Top* на 18 и 18, чтобы второй компонент был как бы немного сдвинут вверх первого. Щёлкни дважды по свойству *Font* и измени цвет на синий. И наконец измени свойство *Transparent* (прозрачный) на *true*.

Если ты всё сделал правильно, то у тебя должно получиться нечто похожее на рисунок 7.3.2

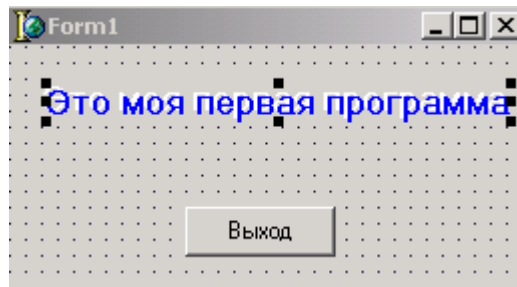



Рис 7.3.2 Результат

Получается как бы надпись с тенью. Симпатно? Я думаю, что да.

7.4 Строки ввода (TEdit).

С помощью строк ввода мы постоянно будем вводить различную информацию в наши программы. Давай попробуем написать несколько программ с использованием этого компонента, чтобы понять все тонкости работы с ним.

Создай новый проект. Брось на него два компонента *TEdit*  с палитры компонентов *Standard* и одну кнопку. У тебя должно получиться нечто подобное:

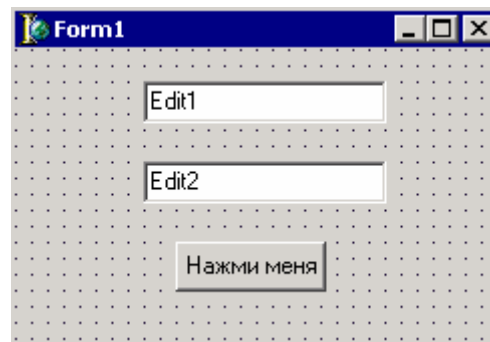


Рис 7.4.1 Форма

Давай очистим у обоих компонентов *TEdit* свойство *Text*. Это свойство отвечает за содержимое строки ввода. Мы просто очистим, чтобы после старта программы обе строки ввода были пустыми.


Теперь создай для кнопки обработчик события *OnClick*. Мы это уже делали, поэтому у тебя не должно быть с этим проблем. Кстати, если дважды щёлкнуть по кнопке, то Delphi автоматически создаст этот обработчик события. Ну а если он уже создан, то просто перенесёт тебя в то место, где написан код этого обработчика.

Итак, создай обработчик и напиши в нём следующее:

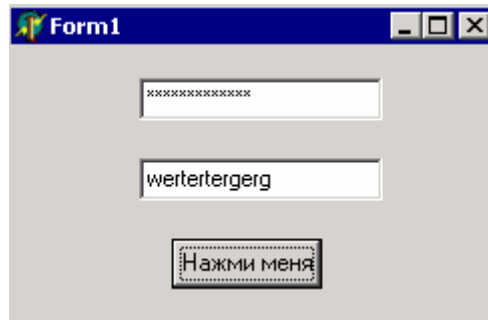
```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Edit2.Text:=Edit1.Text;
end;
```

По нажатию кнопки я копирую содержимое свойства *Text* компонента *Edit2* в свойство *Text* компонента *Edit1*.


Попробуй запустить программу. Теперь введи в первую строку ввода какой-нибудь текст, а потом нажми кнопку. Во второй строке ввода появится тот же текст.

 На компакт диске, в директории \Примеры\Глава 7\TEdit ты можешь увидеть пример этой программы.

Давай немного улучшим пример. Теперь измени у первой строки ввода свойство *PasswordChar* на звёздочку «*». Теперь запусти программу и попробуй ввести в эту строку текст. Вместо текста будут появляться звёздочки, как при вводе пароля в какой-нибудь программе:



Именно таким образом делаются строки ввода паролей. Попробуй нажать на кнопку и во вторую строку ввода перенесётся текст, который ты вводил.

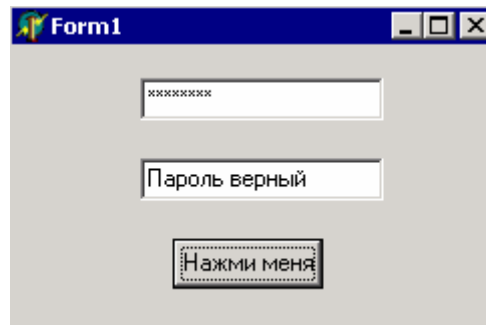
 На компакт диске, в директории \Примеры\Глава 7>PasswordChar ты можешь увидеть пример этой программы.

Давай теперь сделаем проверку на ввод пароля. Найди в процедуру обработчик события *OnClick* для кнопки и напиши там следующее:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  if Edit1.Text='password' then
    Edit2.Text:='Пароль верный'
  else
    Edit2.Text:='Неверно';
end;
```

Попробуй запустить программу. Если ты введёшь в первую строку ввода слово *password* и нажмёшь кнопку, то во второй строке появится надпись «Пароль верный», иначе будет надпись «Пароль неверный».


 На компакт диске, в директории \Примеры\Глава 7>PasswordChar1 ты можешь увидеть пример этой программы.

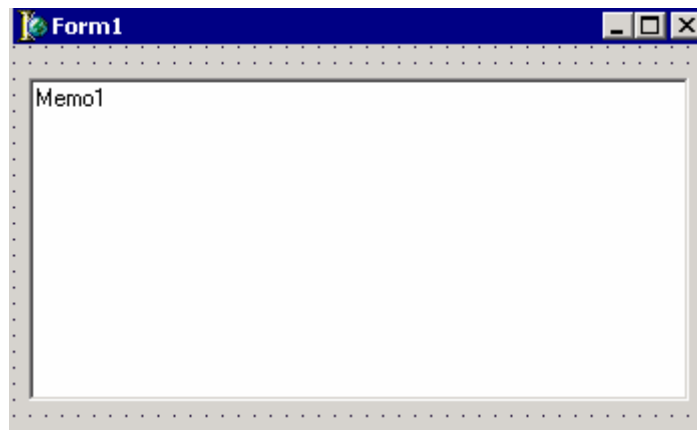


Этих знаний достаточно для написания практически любой программы со строкой ввода. Мы будем регулярно сталкиваться с этим компонентом, поэтому ты ещё успеешь познакомиться с ним поближе.

7.5 Многострочное поле ввода (TMemo).

Теперь мы познакомимся с многострочными компонентами ввода инфы. Для этого на закладке *Standard* есть компонент *TMemo*.

Создай новый проект. Установи на форму компонент *TMemo* .

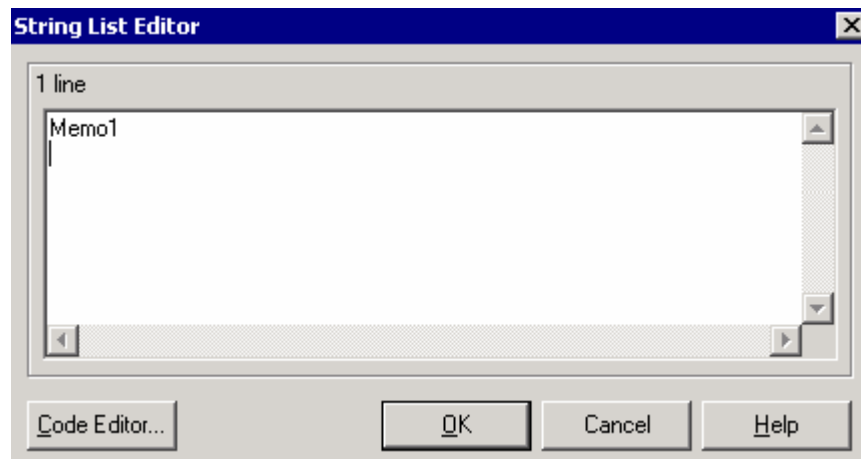


7.5.1 Форма с компонентом *TMemo*

По умолчанию, в нём уже присутствует одна строка текста равная имени компонента. За содержимое текста отвечает свойство *Lines*. Это свойство целый объект типа *TStrings*, и имеет свои свойства и методы. Немного позже мы познакомимся с некоторыми из них.

Давай сначала просто очистим содержимое компонента *Memo1*. Для этого дважды щёлкни по свойству *Lines*. Перед тобой откроется окно редактора строк (смотри рисунок 7.5.2). Это окно содержит простенький текстовый редактор, в котором можно набрать многострочный текст. Мы не будем этого делать, а просто удалим всё содержимое. Как только сделаешь это, нажми кнопку «ОК».

Теперь давай напишем небольшую программу, которая будет выполнять простые функции текстового редактора. Сложные вещи не получаться, потому что *TMemo* – это простой компонент. Для создания более сложных текстовых редакторов есть другой компонент.




7.5.2 Редактор строк.

Итак, добавь на форму пока только одну кнопку. Измени её свойство *Caption* на «Очистить» и имя на *ClearButton*. Кстати, давай изменим имя и компонента *Memo1* на *MainMemo*. Создай для кнопки обработчик события *OnClick*. В нём напиши следующее:

```
procedure TForm1.ClearButtonClick(Sender: TObject);
begin
  MainMemo.Lines.Clear;
end;
```

Здесь я вызываю метод *Clear* объекта *Lines*, который в свою очередь принадлежит объекту *MainMemo*. Немного запутано, но со временем ты поймёшь, что это очень даже удобно. У *MainMemo* есть свойство *Lines*, значит, чтобы получить к нему доступ мы должны написать *MainMemo.Lines*. У объекта *Lines* есть метод *Clear*, который очищает содержимое линий. Вот так и получается эта конструкция.

Попробуй запустить программу и написать внутри компонента *Мето* какой-нибудь текст. Потом нажми кнопку очистить, чтобы уничтожить всё, что ты ввёл.

 На компакт диске, в директории \Примеры\Глава 7\Мето ты можешь увидеть пример этой программы.

Давай теперь усложним наш пример, добавив возможность сохранения введённого текста и загрузки его обратно.

Создай обработчик события *OnShow* для формы и напиши там следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  MainMemo.Lines.LoadFromFile('memo.txt');
end;
```

Здесь я вызываю метод *LoadFromFile* объекта *Lines*. Ему нужно передать только один параметр – имя файла, откуда будет происходить загрузка данных. Есть только один недостаток – если ты сейчас попытаешься запустить программу, то во время загрузки произойдёт ошибка, потому что файла *memo.txt* нет. Тут есть два выхода:

1. Заранее создать этот файл.

2. При старте проверять, есть ли файл, и только если он существует производить загрузку текста.


С первым способом всё понятно. А вот для реализации второго способа нужно воспользоваться функцией *FileExists*, Ей нужно передать имя файла, которое надо проверить, и если такой файл существует, то она вернёт true. Так что давай изменим обработчик события *OnShow* следующим образом:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  if FileExists('memo.txt') then
    MainMemo.Lines.LoadFromFile('memo.txt');
end;
```

Теперь создадим обработчик события *OnClose*. В нём напомним процедуру сохранения содержимого *МемоМемо*.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  MainMemo.Lines.SaveToFile('memo.txt');
end;
```

Здесь используется метод *SaveToFile*, который работает так же, как и *LoadFromFile*, только этот сохраняет данные.

 На компакт диске, в директории \Примеры\Глава 7\Мемо1 ты можешь увидеть пример этой программы. Не запускай пример с привода CD-ROM, потому что при выходе из программы происходит попытка сохранить имеющиеся данные. А так как на CD-ROM записать невозможно, произойдёт ошибка.

Теперь нам осталось только научиться программно добавлять, удалять и изменять строки в компоненте *ТМемо* и можно считать, что мы досконально изучили этот компонент.

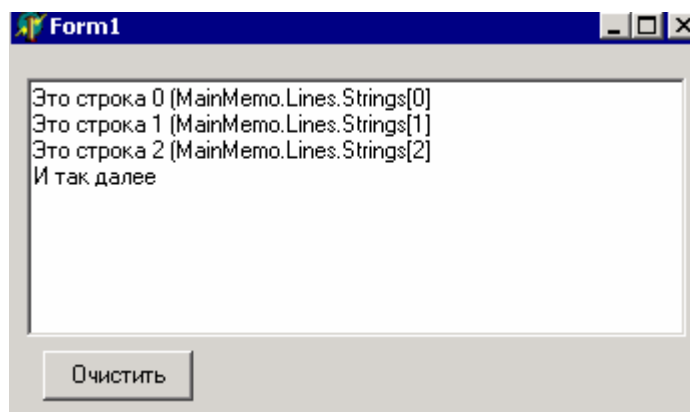


Рис 7.5.3. Хранение строк в *ТМемо*

В компоненте *ТМемо* строки хранятся как простая последовательность строк. Я набрал в прошлом примере текст, и после закрытия программы посмотрел, как он выглядит в файле. Вот содержимое *memo.txt*:

Библия для программиста в среде Delphi

Copyright Horrific
www.cydsoft.com/vr-online/

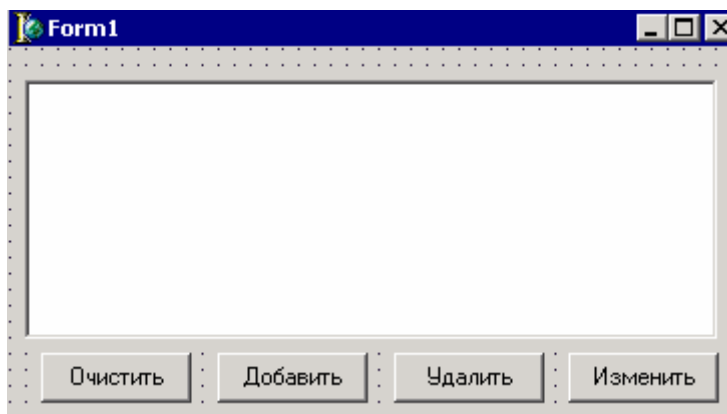
Как видишь, в файле четыре строчки (вторая пустая). Точно так же строки расположены и в памяти.

Для доступа к каждой строке можно воспользоваться свойством `Strings` объекта `Lines`. На рисунке 7.5.3 наглядно показано, как получить доступ к строкам. Чтобы получить нулевую строку, нужно написать `MainMemo.Lines.Strings[0]`, для первой строки `MainMemo.Lines.Strings[1]`, и так далее.

Давай напишем пример, который будет получать доступ к строкам, чтобы увидеть всё это на практике. Добавь к предыдущему примеру три кнопки:

1. Добавить. Я дал имя этой кнопке - `AddButton`.
2. Удалить. Я дал имя этой кнопке – `DelButton`.
3. Изменить. Я дал имя этой кнопке – `ChangeButton`.

Можешь их расположить следующим образом:



Теперь создадим обработчик события `OnClick` для кнопки «Добавить». Здесь мы будем программно добавлять новую строку в `MainMemo`. Для этого у объекта `Lines` есть метод `Add`, у которого есть только один параметр – текст новой строки:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  MainMemo.Lines.Add(' Новая строка');  
end;
```

Теперь создадим обработчик события `OnClick` для кнопки удалить. По этому событию мы должны удалить строку, в которой находится сейчас курсор. Для этого напиши в обработчике следующий текст:

```
procedure TForm1.DelButtonClick(Sender: TObject);  
begin  
  if MainMemo.Lines.Count <> 0 then  
    MainMemo.Lines.Delete(MainMemo.CaretPos.Y);  
end;
```

В первой строке я здесь проверяю, сколько строк в компоненте *MainMemo*. Для этого есть свойство *Count* объекта *Lines*. Если оно не равно нулю (*MainMemo.Lines.Count* <> 0), значит, строки есть, и мы можем удалить текущую строку. Знак <> означает «не равно».

Для удаления я использую метод *Delete* объекта *Lines*. В качестве единственного параметра нужно передать номер строки для удаления. Но как узнать, какая строка сейчас является текущей? Для этого у *MainMemo* есть свойство *CaretPos*, которое указывает на текущую позицию курсора.

CaretPos – это переменная типа *TPoint*. Этот тип переменной – запись. С таким типом мы подробно познакомимся немного позже. Единственное, что сейчас необходимо знать, так это то, что этот тип похож на объект, только у него нет методов, а только свойства. У *TPoint* есть два свойства «X» и «Y». X – указывает на текущую колонку, а Y указывает на текущую строку. Таким образом, я могу узнать текущую строку с помощью *MainMemo.CaretPos.Y*.


Итак *MainMemo.Lines.Delete* удаляет указанную строку. Я указываю текущую строчку с помощью *MainMemo.CaretPos.Y*. Я думаю, что задача выполнена.

Нам осталось только написать обработчик события *OnClick* для кнопки «Изменить». В нём напиши следующее:

```
procedure TForm1.ChangeButtonClick(Sender: TObject);
begin
  MainMemo.Lines.Strings[MainMemo.CaretPos.Y]:='Horrific';
  MainMemo.Lines.Strings[0]:='Текст изменён';
end;
```

В первой строке я изменяю текущую строку на 'Horrific'. Второй строкой кода я изменяю первую строчку *MainMemo* на 'Текст изменён'.

Здесь тебе уже всё должно быть знакомым. Так что можно считать, что мы изучили основные возможности компонента *TMemo*.

 На компакт диске, в директории \Примеры\Глава 7\Мемо2 ты можешь увидеть пример этой программы. Не запускай пример с привода CD-ROM, потому что при выходе из программы происходит попытка сохранить имеющиеся данные. А так как на CD-ROM записать невозможно, произойдёт ошибка.

7.6 Объект TStrings.

В предыдущей части я познакомил тебя на практике с объектом *TStrings*. Свойство *Lines* компонента *TMemo* имеет такой тип. Это очень сильный объект, с которым мы будем очень часто встречаться на протяжении всей книги, поэтому я решил здесь остановиться и рассказать о нём подробнее.

Объект *TStrings* это набор строк. Везде, где информация поделена на строки этот объект является мощнейшим средством для хранения и работы со строками. Представь себе простой текстовый файл. Как хорошо, когда с ним можно работать именно разбив содержимое на строки, а не со всей сплошной информацией. Когда я буду рассказывать о работе с файлами, то этот объект тоже будет присутствовать. Я думаю, что ты ещё не раз будешь возвращаться к этой главе, чтобы вспомнить, для чего предназначено то, или иное свойство или метод.

Здесь я дам только описание свойств и методов объекта и начнём мы конечно же со свойств.

Свойства объекта TStrings:

Count – это свойство, которое ты можешь только читать. Здесь храниться количество строк, содержащихся в объекте.

Strings – здесь храниться сам набор строк. К любой строке ты можешь получить доступ, написав такую конструкцию:

```
Переменная:=Имя Объекта.Strings[Номер строки];  
Имя Объекта.Strings[Номер строки]:= Переменная;
```

Первая строка кода запишет в переменную содержимое указанной строки. Вторая строка наоборот запишет содержимое переменной в указанную строку. Запомни, что строки в этом объекте нумеруются с нуля.

Text – в этом свойстве хранятся все строки в виде одной целой строки.

Методы объекта TStrings:

Add(Строка) – этот метод добавляет строку, указанную в качестве параметра в конец набора строк объекта. Этот метод возвращает номер, под которым добавлена эта строка.

Append(Строка) – этот метод тоже добавляет строку, указанную в качестве параметра в конец набора строк объекта. Метод ничего не возвращает.

AddStrings(Набор строк типа TStrings) – этот метод добавляет все строки из другого объекта типа *TStrings*.

Assign – этот метод присваивает вместо своего набора строк, указанный в качестве параметра новый набор.

Clear – удалить все строки из объекта.

Delete(номер строки) – удалить строку под указанным номером.

Equals(Набор строк типа TStrings) – сравнить собственный набор строк с указанным в качестве параметра. Если наборы равны, то метод вернёт *true* иначе вернёт *false*.

Exchange(Номер1, Номер2) – поменять местами строки указанных номеров.

Get(номер строки) – метод возвращает строку указанного номера.

IndexOf(Строка) – найти указанную в качестве параметра строку. Если такая строка существует в наборе, то метод вернёт её индекс, иначе –1.

Insert(Номер, Строка) – вставить в набор новую строку под указанным номером.


LoadFromFile(Имя файла) – загрузить набор строк из указанного текстового файла.

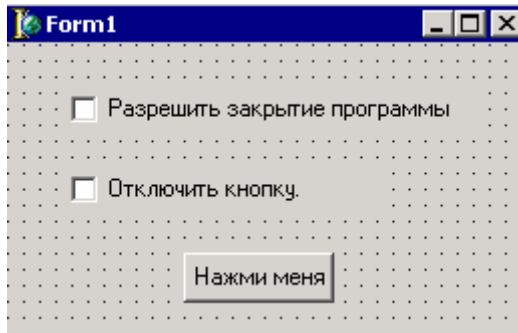
SaveToFile(Имя файла) – сохранить набор строк в указанный текстовый файл.

Move(Номер1, Номер2) – переместить строку под номером 1, на место номера 2.

Пока этих свойств и методов больше чем достаточно. В этой книге будут рассматриваться в основном эти методы. Я не описывал всё, и те возможности, которыми ты не будешь пользоваться (знаю по своему опыту) я описывать не стал.

7.7 CheckBox.

Теперь мы переходим к рассмотрению компонента *CheckBox*. Как всегда, давай создадим маленькую программу, которая будет использовать этот компонент. Создай новое приложение, брось на него одну кнопку и два компонента *TCheckBox* . Первому компоненту дай заголовок (Caption) равным «Разрешить закрытие программы» и имя (Name) равным «AllowCloseCheckBox». Второму компоненту дай заголовок (Caption) равным «Отключить кнопку» и имя (Name) равным «EnableButtonCheckBox». Кнопке я дал имя «MyFirstButton».



Создай обработчик события *OnClick* для компонента *EnableButtonCheckBox* (это второй *CheckBox*). В нём напиши следующее:

```
procedure TForm1.EnableButtonCheckBoxClick(Sender: TObject);
begin
  MyFirstButton.Enabled:=not EnableButtonCheckBox.Checked;
end;
```

Здесь я присваиваю свойству *Enabled* нашей кнопки значение **not** *EnableButtonCheckBox.Checked*. Что это значит? Свойство *Checked* компонента *EnableButtonCheckBox* показывает: стоит ли галочка на это *CheckBox*-е. Если да, то свойство *Checked* равно *True*, иначе *False*. Оператор *not* меняет это состояние на противоположное. Это значит, что если свойство *Checked* было равно *True*, то в *MyFirstButton.Enabled* будет присвоено противоположное (*False*).

Можешь попробовать запустить пример, и посмотреть что происходит. Когда тыставишь галочку напротив «Отключить кнопку», свойство *Checked* этого компонента меняется на *True*. Срабатывает событие *OnClick* и в свойство *Enabled* кнопки присваивается значение свойства *Checked* компонента *CheckBox*, изменённое на противоположное, т.е. *False*. А когда свойство *Enabled* кнопки равно *False* она становится недоступной.

Чтобы окончательно разобраться с работой примера понажимай на *EnableButtonCheckBox* при запущенной программе. Потом попробуй убрать из исходного кода оператор **not** и снова запусти программу.

Теперь давай создадим обработчик *OnClick* для кнопки. В нём напиши следующее:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  if AllowCloseCheckBox.Checked then
    Close;
end;
```

Здесь я проверяю, если свойство *Checked* компонента *AllowCloseCheckBox* (первый *CheckBox* на форме) равно *True*, то закрыть программу (выполнить метод *Close*). Иначе ничего не произойдёт.

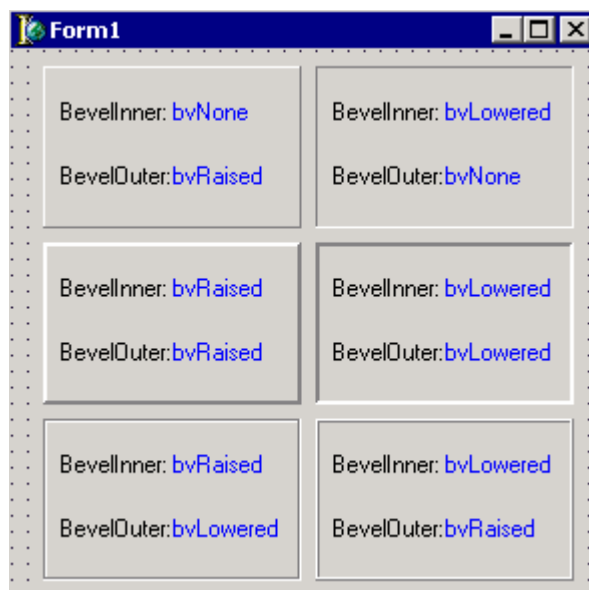
В принципе, это всё, что нужно знать и как можно использовать компонент *TConboBox*. Он достаточно простой, но в большинстве программ незаменим.


7.8 Панели (TPanel).


Вообще-то, если идти по порядку, то сейчас должен был быть *TRadioButton*. Но я решил немного перескочить сразу на *TPanel*. Потому что я начну её использовать уже в следующих примерах.

TPanel это компонент в виде панели. Он ведёт себя, так же как и форма. Ты можешь на нём располагать компоненты, и если ты передвинешь панель, то все компоненты установленные на ней тоже передвинутся.

Панель может выглядеть по разному. За внешний вид отвечают два свойства: *BevelInner* и *BevelOuter*. Я написал маленькую программу, которая ничего не делает, зато она показывает, как может выглядеть панель с различными вариантами установленных параметров. На панели синим шрифтом написаны установленные значения *BevelInner* и *BevelOuter*:



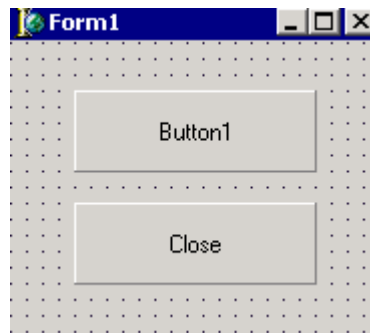
 На компакт диске, в директории \Примеры\Глава 7\Panel ты можешь увидеть пример этой программы.

Давай напишем пример, в котором будем программно менять внешний вид панели. Для этого создай новое приложение и установи на форму два компонента *TPanel* с палитры компонентов *Standard* .

В этом примере я не буду менять имена панелей и оставлю их по умолчанию *Panel1* и *Panel2*. Не знаю почему, но я так захотел.

Единственное, что мы поменяем – это свойства *Caption* обеих панелей. У первой я написал «*Button1*», а у второй – «*Close*».

На рисунке ниже ты можешь увидеть форму будущей программы:



Теперь создадим обработчик события *OnMouseDown* для первой панели.

```
procedure TForm1.Panel1MouseDown(Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
    Panel1.BevelOuter:=bvLowered;  
end;
```

Одна строчка кода меняет вид панели. Создадим ещё обработчик события *OnMouseUp* для первой панели. По этому событию мы меняем вид панели на исходный:

```
procedure TForm1.Panel1MouseUp(Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
    Panel1.BevelOuter:=bvRaised;  
end;
```

Попробуй запустить программу и нажать на первую панель. Когда ты нажмёшь мышь, панель изменит вид на вогнутый. При отпускании мыши панель возвращает вид на исходный. Таким образом, панель начинает работать как кнопка.

Раз так, давай создадим экзотичную кнопку. Для второй панели изменим свойства:

- ***BevelOuter*** на *bvRaised*
- ***BevelInner*** на *bvLowered*.

Теперь создадим обработчик события *OnMouseDown* для второй панели.

```
procedure TForm1.Panel2MouseDown(Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
    Panel2.BevelOuter:=bvLowered;  
    Panel2.BevelInner:=bvRaised;  
end;
```

Здесь меняется вид панели на вогнутый. Теперь создадим обработчик события *OnMouseUp* для второй панели. По этому событию мы меняем вид панели на исходный:


```
procedure TForm1.Panel2MouseUp(Sender: TObject;
```

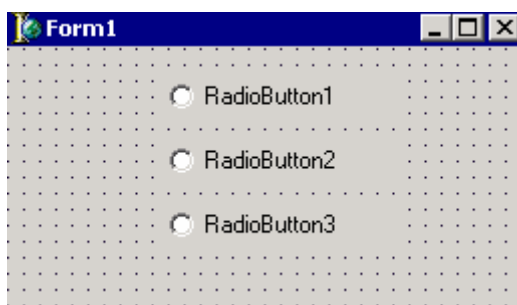
```
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
  Panel2.BevelOuter:=bvRaised;  
  Panel2.BevelInner:=bvLowered;  
  Close;  
end;
```


 На компакт диске, в директории \Примеры\Глава 7\Panel1 ты можешь увидеть пример этой программы.

7.9 Кнопки выбора TRadioButton.

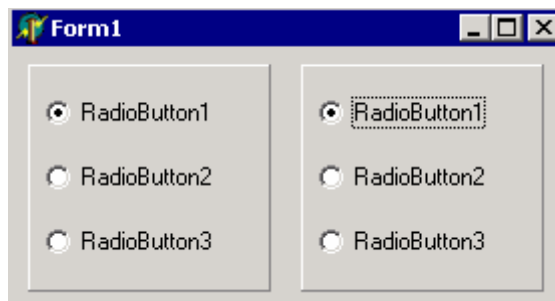
Эти кнопки очень похожи на *TCheckBox* даже по методу работы. У них так же есть свойство *Checked*, которое отображает её состояние. Если *RadioButton* выделен, то это свойство равно *True*, иначе равно *False*. Единственная разница – если у тебя на форме есть несколько таких компонентов, то одновременно может быть выделен только один.


Давай посмотрим это на практике. Брось на форму несколько компонентов *RadioButton* . Теперь запусти программу и попробуй пощёлкать.



 На компакт диске, в директории \Примеры\Глава 7\RadioButton ты можешь увидеть пример этой программы.

Как видишь, ты не можешь выделить сразу два компонента *RadioButton*. А как же тогда сделать возможность двойного выбора на форме? Для этого компоненты *RadioButton* можно убрать на панели:




 На компакт диске, в директории \Примеры\Глава 7\RadioButton1 ты можешь увидеть пример этой программы.

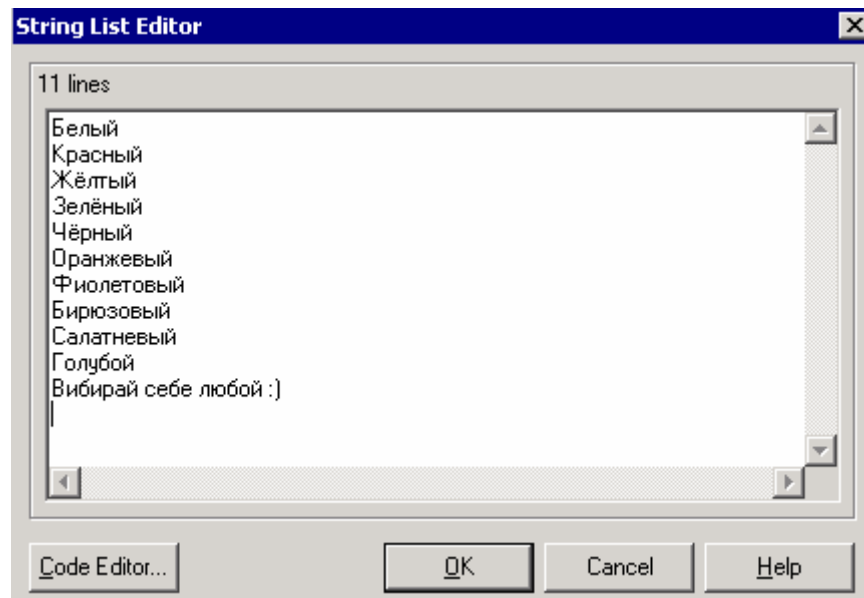
Больше ничего нового я не могу сказать. Как я уже сказал, работа с этим компонентам происходит так же, как и с *CheckBox*.

7.10 Списки выбора (TListBox).

Списки выбора хранят в себе какие-то списки (например, параметров) и дают пользователям возможность выбирать один или несколько параметров.

В работе списки достаточно просты. Чтобы получить доступ к строкам списка нужно воспользоваться свойством *Items* объекта *TListBox*. Это свойство имеет тип *TStrings*. Это ничего тебе не напоминает? Такой же тип у свойства *Lines* объекта *TMemo*. Значит работа со строками списка нам уже известна и не вызовет затруднений, потому что всё, что мы говорили про методы и свойства *Lines* объекта *TMemo* так же относится и к свойству *Items* объекта *TListBox*.

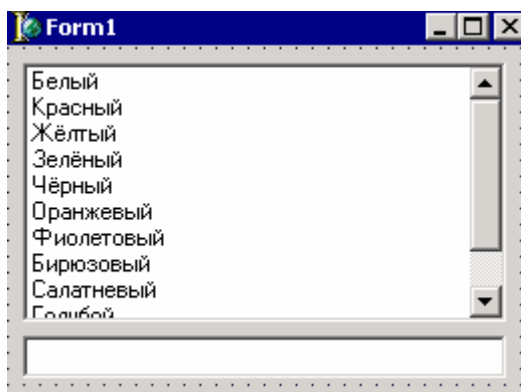
Давай напишем маленькое приложение с использованием этого компонента. Создай новый проект. Брось на форму один компонент *TListBox*  и один компонент *TEdit*. Теперь дважды щёлкни по свойству *Items* компонента *Listbox1*. Перед тобой откроется редактор строк:



Я набрал здесь названия всех основных цветов:

Белый
Красный
Жёлтый
Зелёный
Чёрный
Оранжевый
Фиолетовый
Бирюзовый
Салатневый
Голубой
Вибирай себе любой :)

После этого жми «ОК», чтобы сохранить введённые данные. У тебя должна получиться такая форма:



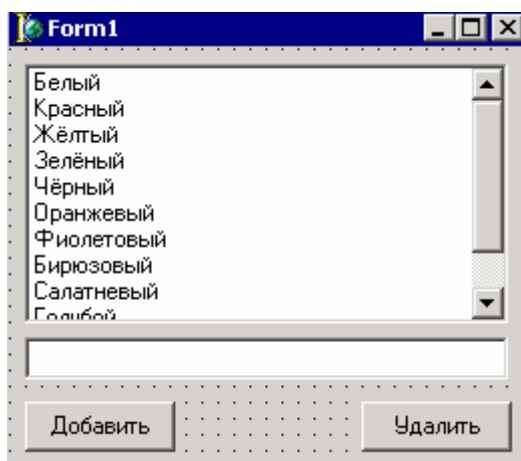
Давай теперь создадим обработчик события *OnClick* для списка выбора. В нём напишем следующее:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Edit1.Text:=ListBox1.Items.Strings[ListBox1.ItemIndex];
end;
```

Свойство *ItemIndex* объекта *ListBox1* указывает на выделенную строку списка выбора. С помощью *ListBox1.Items.Strings* мы можем получить доступ ко всем строкам списка. В результате получается, что я присваиваю в *Edit1* текст выделенной строки в списке выбора.

 На компакт диске, в директории \Примеры\Глава 7\ListBox ты можешь увидеть пример этой программы.

Всё очень похоже на работу с *ТМето*. Для большей ясности, давай добавим к нашему приложению ещё кнопку «Добавить» и «Удалить» строку.



Для кнопки добавить, в обработчике события *OnClick* напишем следующий код:

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

```
ListBox1.Items.Add('Новая строка')  
end;
```

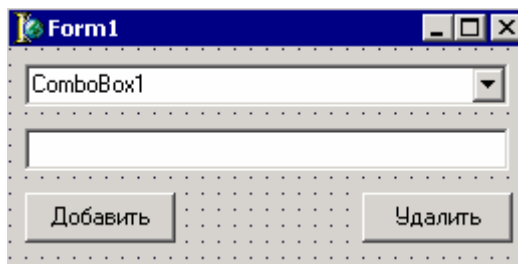
Для кнопки удалить напишем такой текст:

```
procedure TForm1.DelButtonClick(Sender: TObject);  
begin  
  ListBox1.Items.Delete(ListBox1.ItemIndex);  
end;
```

Я думаю, что комментарии излишни. Полная аналогия с *ТМето*.

7.11 Выпадающие списки (TComboBox).

Выпадающие списки по своей работе, свойствам и методам похожи на списки выбора. Я бы сказал, что это полная копия. Давай создадим приложение похожее на предыдущее, только вместо *ListBox* будет *ComboBox*.



Теперь создадим обработчик события *OnChange* для выпадающего списка *ComboBox1*. Это событие происходит, когда пользователь выбрал какой-нибудь элемент списка. По нему мы напишем следующий текст:

```
procedure TForm1.ComboBox1Change(Sender: TObject);  
begin  
  Edit1.Text:=ComboBox1.Items.Strings[ComboBox1.ItemIndex];  
end;
```


Как видишь, это вообще полная копия кода из предыдущего примера, только используется другое имя компонента.

Теперь напишем код для кнопки «Добавить»:

```
procedure TForm1.AddButtonClick(Sender: TObject);  
begin  
  ComboBox1.Items.Add('Новая строка')  
end;
```

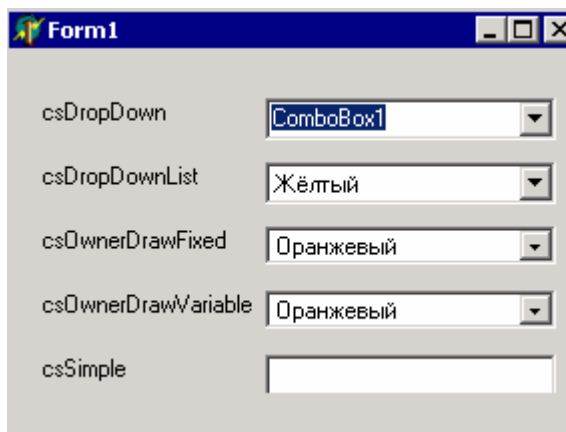
Для кнопки удалить код будет таким:

```
procedure TForm1.DelButtonClick(Sender: TObject);
begin
  ComboBox1.Items.Delete(ComboBox1.ItemIndex);
end;
```

 На компакт диске, в директории \Примеры\Глава 7\ComboBox ты можешь увидеть пример этой программы.

Как видишь, наблюдается полная аналогия со списком выбора и *TMemo*. Содержимое списков можно сохранять с помощью *ComboBox1.Items.SaveToFile('Имя файла')*, а загружать с помощью *ComboBox1.Items.LoadFromFile('Имя файла')*.

Существует несколько типов выпадающих списков. За тип списка отвечает свойство *Style*. Я написал маленькую программу, которую ты можешь найти на диске в \Примеры\Глава 7\ComboBox1, которая показывает все типы списков в действии.



Этот пример не может показать все различия стилей, поэтому я дам ещё небольшое описание:

CsDropDown – основной стиль. При нём ты можешь не только выбирать значения из списка, но и вводить в строку свои.

CsDropDownList – при этом стиле можно только выбирать из списка.

CsOwnerDrawFixed – при этом стиле ты можешь рисовать элементы сам. Высота элементов фиксированная.

CsOwnerDrawVariable – при этом стиле ты можешь рисовать элементы сам. Отличается от предыдущего тем, что высота элементов не фиксированная.


CsSimple – только строка ввода.

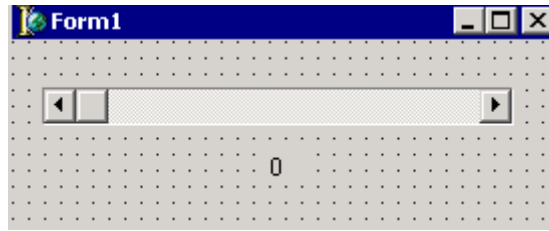
Немного позже в этой книге я покажу тебе как рисовать внутри списков выбора и выпадающих списков.

7.12 Полосы прокрутки (TScrollBar).

Полосы прокрутки очень часто используются для прокручивания какого-то действия. Например, когда ты слушаешь музыку, ты можешь прокрутить её в любое место с помощью простой полосы прокрутки. Или если информация не помещается в окно, её так же прокручивают с помощью таких полосок, но в

большинстве случаев это делается автоматически, и тут ты вмешиваться будешь очень редко.

Давай посмотрим на полосу прокрутки в действии. Создай новое приложение. Брось на форму один компонент *TLabel* и одну полосу прокрутки *TScrollBar* . У тебя должна получится форма следующего вида:




У компонента *Label1* измени свойство *Caption* на «0». Теперь создай обработчик события *OnChange* для полосы прокрутки и напиши там следующее:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Label1.Caption:=IntToStr(ScrollBar1.Position);
end;
```

В этом коде я присваиваю свойству *Caption* компонента *Label1* значение текущей позиции ползунка полосы прокрутки. Текущее значение ползунка можно получить с помощью свойства *Position* объекта *ScrollBar1*. Только тут есть одно «НО». Это свойство имеет тип целое число, а свойство *Caption* компонента *Label1* – это строка. Поэтому нам надо превратить целое число в строку. Для этого есть функция *IntToStr*. Ей нужно передать число, а она нам вернёт строку. Поэтому если вызвать эту функцию с параметром текущей позиции ползунка (*IntToStr(ScrollBar1.Position)*), результат её работы можно присвоить свойству *Caption* компонента *Label1*.

Попробуй запустить программу и подвигать ползунок. Значение позиции будет отображаться в компоненте *Label1*.

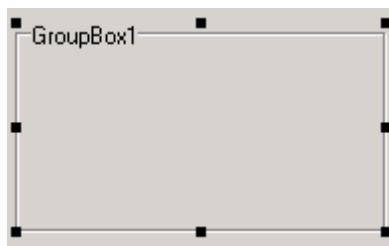
 На компакт диске, в директории \Примеры\Глава 7\ScrollBar ты можешь увидеть пример этой программы.

В этом примере мы написали пример горизонтальной полосы прокрутки. Чтобы сделать её вертикальной, нужно свойство *Kind* поменять на *sbVertical*. И ещё, значение ползунка изменяется от 0 до 100. Чтобы изменить эти значения есть свойства *Min* (по умолчанию равно нулю) и *Max* (по умолчанию равно 100).

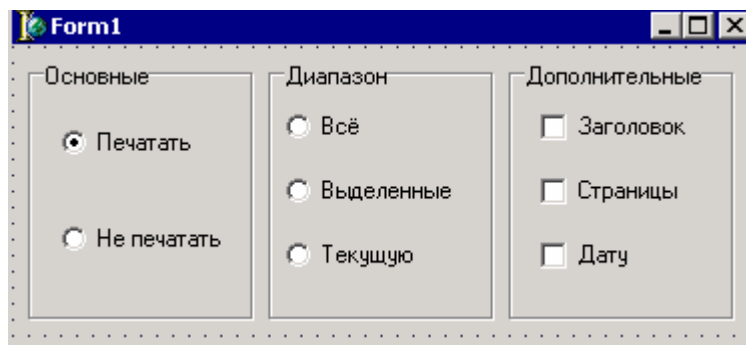
Больше ничего особенного в работе полос прокрутки нет. Поэтому давай двигаться дальше.


7.13 Группировка объектов (GroupBox).

Компонент *GroupBox* очень удобно использовать для группировки каких-то компонентов. На вид это простая панель только с заголовком наверху:



За текст отображаемый в заголовке отвечает свойство *Caption*. Больше ничего особенного эта панель делать не умеет. Я написал маленькую программу, чтобы показать, как можно использовать компонент *TGroupBox*.



 На компакт диске, в директории \Примеры\Глава 7\GroupBox ты можешь увидеть пример этой программы.

Панель *TGroupBox* в основном используют для группировки компонентов *TRadioButton*.

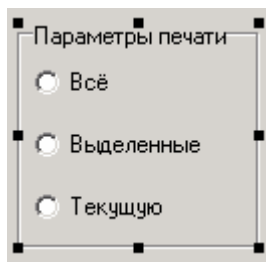
7.14 Группа компонентов RadioButton (TRadioGroup).

Если установить этот компонент на форму, то на первый взгляд он выглядит, как простой *TGroupBox*.

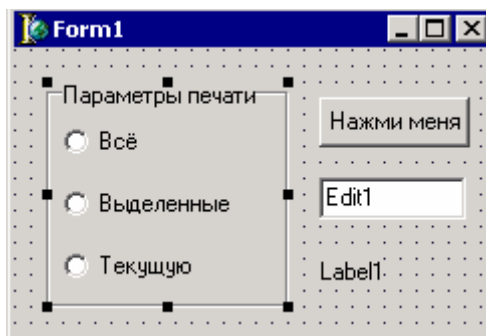
Создай новый проект и прось на него один компонент *TRadioBox*. Но это только на первый взгляд. Щёлкни по свойству *Items* и перед тобой появится уже знакомый редактор строк. Введи туда три строчки:

- Всё
- Выделенные
- Текущую

Нажми «ОК» и твой ты увидишь *GropBox* содержащий в себе три компонента *TRadioBox*. Зачем же нужен такой гибрид? Потерпи немного, пока мы не напишем пример до конца. Тогда ты сможешь оценить все прелести этого гибрида и возможно полюбишь его.



Брось на форму ещё одну кнопку, одну строку ввода и один *TLabel*. Расположи их так, как показано на скрине ниже:



Теперь создадим обработчик события *OnClick* для компонента *RadioGroup1*. В нём напишем следующее:

```
Label1.Caption:=IntToStr(RadioGroup1.ItemIndex);
```

Свойство *ItemIndex* компонента *RadioGroup1* показывает, какой компонент сейчас выделен. Компоненты пронумерованы в таком же порядке, как записаны их имена в списке. Это свойство имеет тип целое число, поэтому его приходится превращать в строку с помощью *IntToStr*.

По нажатию кнопки напишем следующий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:=IntToStr(RadioGroup1.ItemIndex);
end;
```

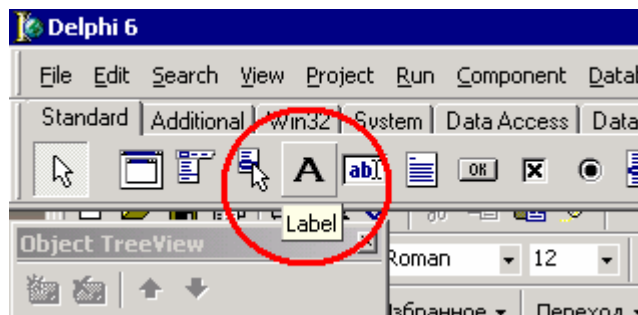
По нажатию кнопки я делаю то же самое, только номер выделенного компонента помещаю в *Edit1*.

А теперь посмотрим на преимущества этого компонента. Представим, что у нас просто стоит три компонента *TRadioButton*. Чтобы узнать, какой из них сейчас выделен, нужно проверить свойство *Checked* всех этих компонентов. А при использовании группы *TRadioGroup* ничего этого не надо. Нужно только проверить свойство *ItemIndex*, компонента *TRadioGroup* и никаких проблем.

7.15 Ответы на вопросы.

Здесь я буду отвечать на вопросы, которые могут возникнуть у тебя при чтении этой главы.

Вопрос: Почему, когда я навожу мышкой на компонент, например *TLabel* выскакивает подсказка, в которой написано *Label*, а не *TLabel*? Куда девается буква «Т»?



Ответ: действительно, в подсказках всегда отсутствует буква «Т». Просто компонент называется *Label*, а объект этого компонента называется *TLabel*. Так принято, что имена всех объектов всегда начинаются с буквы «Т». Это не значит, что так обязательно. Это значит, что так желательно. Просто взглянул на имя и видишь, что это имя объекта. А в подсказках показывают имя компонента, к которому нет такого соглашения, поэтому там нет никаких букв вначале.