

Глава 8. Учимся программировать.	127
8.1 Циклы (for..to..do).....	128
8.2 Циклы (while).....	130
8.3 Циклы (Repeat).	132
8.4 Управление циклами.....	133
8.5 Логические операторы.....	137
8.6 Работа со строками.....	140
8.7 Исключительные ситуации.	143
Глава 9. Создание рабочих приложений.	146
9.1 Создание главного меню программы.	147
9.2 Создание дочерних окон.....	150
9.3 Модальные и не модальные окна.	154
9.4 Обмен данными между формами.	156
9.5 Многодокументные MDI окна.	157
9.6 Инициализация окон.....	161

Глава 8. Учимся программировать.

В этой главе я буду описывать основные принципы программирования. Мы познакомимся с циклами на практике, увидим различные приёмы кодинга, и всё это будет снабжено громадным количеством полезных примеров.

Я всегда говорил, что все знания приходят с практикой. Ты сможешь по настоящему что-то освоить только когда попробуешь что-нибудь сам. Поэтому я ещё раз прошу тебя повторять все описываемые мной действия самостоятельно. Все исходники, которые идут на диске, я привёл только для того случая, когда у тебя что-то не получается. Ты можешь посмотреть их содержимое, но после этого ты просто обязан повторить все действия самостоятельно. Иначе тебе никакая книга не поможет.

Итак, я приступаю к последней главе, в которой я буду описывать чисто теорию. После этого мы будем знакомиться уже только с разными приёмами и технологиями кодинга. И это последняя глава, в которой я закладываю основы кодинга.



8.1 Циклы (for..to..do).

Циклы – это основа любого кодирования. Мы будем использовать их практически в каждой программе. Мы уже познакомились с ними на практике. Напомню тебе, как выглядит логика цикла:

От 1 до 5 выполнять
Начало цикла
*R:=R*INDEX;*
INDEX:=INDEX+1;
Конец цикла

Это логика расчёта факториала. Давай эту логику перенесём на язык программирования Delphi.

Цикл в Delphi оформляется как:

for переменная:=начальное значение to конечное значение do
Действие;

После слова for нужно присвоить какой-нибудь переменной начальное значение. Эта переменная будет использоваться в качестве счётчика. После каждого выполнения действия этот счётчик будет увеличиваться на единицу, пока переменная не превысит конечного значения.

В общем виде цикл выглядит так:

for..to..do
Действие1

Рассмотрим пример:

```
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
  Sum:=0;
  for index:=0 to 5 do
    Sum:=Sum+index;
  end;
```

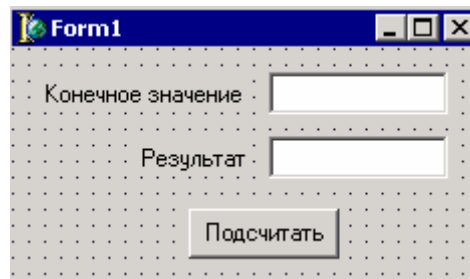
В этом примере я объявляю две переменные index и sum типа целое число. Сначала я присваиваю переменной Sum значение 0. После этого запускаю цикл в котором переменная index будет изменяться от 0 до 5.

Теперь посмотрим поэтапно, что здесь происходит:

1. На первом этапе переменная *index* равна нулю. *Sum* тоже равна нулю, значит выполнится операция *Sum:=0+0*. Результат *Sum=0*;
2. На втором этапе *index* увеличена на 1 значит, выполнится действие *Sum:=0+1*. Результат *Sum=1*.
3. Здесь *index* увеличена на 1 и уже равна 2, а *Sum=1*. Значит, выполнится действие *Sum:=1+2*. Результат *Sum=3*.
4. Здесь *index* увеличена на 1 и уже равна 3, а *Sum=3*. Значит, выполнится действие *Sum:=3+3*. Результат *Sum=6*.
5. Здесь *index* увеличена на 1 и уже равна 4, а *Sum=6*. Значит, выполнится действие *Sum:=4+6*. Результат *Sum=10*.
6. Здесь *index* увеличена на 1 и уже равна 5, а *Sum=10*. Значит, выполнится действие *Sum:=5+10*. Результат *Sum=15*.

Заметь, что мы не увеличиваем переменную *index*, она увеличивается автоматически.

Давай перенесём этот код в программу, чтобы мы могли убедиться в этом на реальном примере. Создай новое приложение. Брось на форму два компонента *TLabel*, два компонента *TEdit* и одну кнопку.



Компонент *Edit1* я переименовал в *EndEdit*, а *Edit2* я переименовал в *ResultEdit*. Теперь по нажатию кнопки я написал следующий код:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
  Sum:=0;

  EndCount:=StrToInt(EndEdit.Text);

  for index:=0 to EndCount do
    Sum:=Sum+index;

  ResultEdit.Text:=IntToStr(Sum);
end;
```

В принципе, текст тот же самый. Единственная разница – я запускаю цикл, начиная от 0 до числа введённого в компонент *EndEdit*. *EndEdit* содержит текст, а мне нужно превратить его в число, поэтому я использую функцию *StrToInt* для преобразования строки в число. Эта функция работает так же, как и *IntToStr*, которая наоборот преобразовывала число в строку.

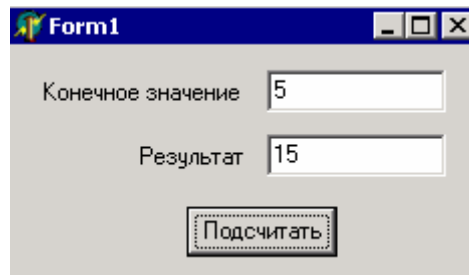
Результат преобразования я сохраняю в переменной *EndCount*:


```
EndCount:=StrToInt(EndEdit.Text);
```

После этого я уже запускаю цикл, в котором переменная *index* будет изменяться от 0 до значения *EndCount* (в котором находится число введённое в *EndEdit*).

```
for index:=0 to EndCount do
```

Запусти программу и введи в строку «*Конечное значение*» число 5. После этого нажми на кнопку и в строке результата должно появиться число 15.



 На компакт диске, в директории \Примеры\Глава 8\for..to..do ты можешь увидеть пример этой программы.

Хочу ещё отметить, что после цикла *for* будет выполняться только одно действие. Например, если ты захочешь выполнить два действия подряд, то ты должен заключить их в скобки **begin** и **end**:

```
for index:=0 to EndCount do
begin
    Sum:=Sum+Index;
    Sum:=Sum+1;
end;
```

В этом примере, на каждом шаге цикла я увеличиваю *Sum* ещё на единицу. Если ты попробуешь написать так:

```
for index:=0 to EndCount do
    Sum:=Sum+Index;
    Sum:=Sum+1;
```

то выполняться в цикле будет только строчка *Sum:=Sum+index*. Вторая строка *Sum:=Sum+1*; выполнится только по окончании расчёта цикла.

Если ты что-то не понял, вернись к главе, где я описывал блок-схемы. Там был описан пример, который работает как цикл *for..to..do*. Попробуй нарисовать блок схему для этого примера сам и мысленно пройти её шаги.

8.2 Циклы (while).

Ещё одной разновидностью цикла является цикл `while`. Это слово переводиться как «пока». Это значит, что цикл будет выполняться, пока выполняется условие.

В общем виде он выглядит следующим образом:

```
while условие do  
Действие
```

Сразу рассмотрим пример:

```
var  
index:integer;  
begin  
index:=0;  
  
while index<10 do  
index:=index+1;  
end;
```

В этом примере я объявляю переменную `index`. В первой строке кода я присваиваю ей 0. После этого я запускаю цикл. В условии я написал `index<10`, это значит, что будет выполняться следующее действие (`index:=index+1`) пока переменная `index` меньше 10.

Здесь так же выполняется только одно действие. Если ты захочешь выполнить в цикле сразу два действия, то ты должен заключить их в скобки *begin* и *end*.

Давай напишем предыдущий пример с использованием цикла `while`. Измени код по нажатию кнопки на следующий:


```
procedure TForm1.CalculateButtonClick(Sender: TObject);  
var  
index:Integer;  
sum:Integer;  
EndCount:Integer;  
begin  
Sum:=0;  
index:=0;  
  
EndCount:=StrToInt(EndEdit.Text);  
  
while index<EndCount do  
begin  
Sum:=Sum+index;  
index:=index+1;  
end;  
  
ResultEdit.Text:=IntToStr(Sum);  
end;
```

Тут мне надо обнулять не только переменную `Sum`, но и `index`, чтобы начальное значение было равно нулю, и цикл шёл от этого нуля до введённого значения. Обрати так же внимание на то, что здесь мне нужно самостоятельно увеличивать переменную `index`

(`index:=index+1`), для этого я эту строчку добавил в цикл. Она вместе с расчётом суммы объединены с помощью *begin* и *end*.

Попробуй запустить программу и ввести число 5. Результатом расчёта будет 10. Если ты помнишь предыдущий пример, то там результат был 15. В чём проблема? Почему разные результаты? В прошлом примере мы выполняли цикл от 0 до 5 включительно. Здесь будет выполняться цикл от 0 и до того момента, пока выполняется условие `index<5`. Когда `index=5` условие не выполнится, и расчёт с цифрой 5 не будет производиться.

Для решения этой проблемы можно поменять условие цикла на `index<=5` (переменная `index` меньше или равна `EndCount`). В этом случае расчёт с цифрой 5 будет считаться. Или можно вводить цифру 5.

 На компакт диске, в директории \Примеры\Глава 8\while ты можешь увидеть пример этой программы.

8.3 Циклы (Repeat).

Теперь давай разберём ещё один тип циклов `repeat..until`. Этот тип цикла похож на `while`. Даже смысл цикла похож. Он означает, что выполнять действия, пока не наступит определённое событие. Только тут есть пару отличий:

1. В цикле `while` действия выполнялись, пока условие верно. В цикле `repeat` действия будут выполняться, пока условие не верно и прекращается, когда оно станет верным.
2. В цикле `while` верность условия проверяется перед началом действий. Это значит, что если условие заведомо неверно, то действия цикла не будут выполнены. В цикле `repeat` сначала выполняется действие, а потом происходит проверка. Это значит, что если условие заведомо не верно, действие всё равно будет выполнено один раз.

В общем виде цикл `repeat` выглядит так:

```
repeat
  действия
until Условие;
```

Заметь, что в этом случае, действий может быть несколько. Тут уже не надо объединять несколько действий в *begin..end*, потому что `repeat..until` уже действует как объединение нескольких действий.

Давай рассмотрим прошлый пример с использованием этого типа цикла:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
  Sum:=0;
  index:=0;

  EndCount:=StrToInt(EndEdit.Text);
```

```
repeat
  Sum:=Sum+index;
  index:=index+1;
until index>EndCount;

ResultEdit.Text:=IntToStr(Sum);
end;
```

В этом примере действия будут выполняться в цикле, пока переменная *index* не станет больше числа указанного в EndCount.

 На компакт диске, в директории \Примеры\Глава 8\repeat ты можешь увидеть пример этой программы.

8.4 Управление циклами.

Работой цикла можно ещё и управлять. В Delphi есть два оператора управления циклами – **break** и **continue**. Начнём мы рассмотрение с оператора **break**.

Допустим, что тебе надо разделить число 10 на числа начиная от –3 до 3 и вывести результат в TListBox.

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r: Integer;
begin
  for i:=-3 to 3 do
    begin
      r:=round(10/i);
      ListBox1.Items.Add('10/' + IntToStr(i) + '=' + IntToStr(r));
    end;
  end;
```

В этом примере я запускаю цикл, в котором переменная **i** будет изменяться от –3 до 3. На каждом шаге я делю 10 на значение **i** и сохраняю результат в переменную **r**.

При делении используется функция **round**, которая округляет переданное ей значение. В качестве значения мы передаём результат деления 10 на переменную **i** **round(10/i)**. Так что в переменную **r** будет записан округлённый до целого результат деления. Этого пока достаточно знать о функции **round**, потому что с ней мы познакомимся поближе немного позже.

После этого я добавляю результат в ListBox1, преобразовывая переменную **r** в строку.

Давай посмотрим, что произойдёт, когда переменная **i** будет равна 0. В этом случае число 10 будет делиться на 0, а значит, произойдёт ошибка, потому что на 0 делить нельзя. Как же тогда выйти из этой ситуации? Можно на каждом этапе цикла проверять значение **i**, и если оно равно 0, то не выполнять никаких действий. Вот два возможных решения:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r: Integer;
```



```
begin
for i:=-3 to 3 do
begin // Это начало для оператора for
if i<>0 then
begin // Это начало для оператора if
r:=round(10/i);
ListBox1.Items.Add('10/' + IntToStr(i) + '=' + IntToStr(r));
end; // Этот end для оператора if
end; // Этот end для оператора for
end;
```

В этом примере я на каждом этапе проверяю значение **i**, и если оно не равно 0, то только в этом случае произвожу вычисления.

Это очень простое решение для маленьких и простых программ. Но если твой цикл большой и выполняет много действий, то такое решение будет как минимум неудобно и может потеряться читабельность кода. В худшем случае, вообще может ничего не получиться. Вот тут на помощь приходит оператор **continue**:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
i, r: Integer;
begin
for i:=-3 to 3 do
begin // Это начало для оператора for

if i=0 then
begin // Это начало для оператора if
ListBox1.Items.Add('На ноль делить нельзя');
Continue;
end; // Этот end для оператора if

r:=round(10/i);
ListBox1.Items.Add('10/' + IntToStr(i) + '=' + IntToStr(r));
end; // Этот end для оператора for
end;
```

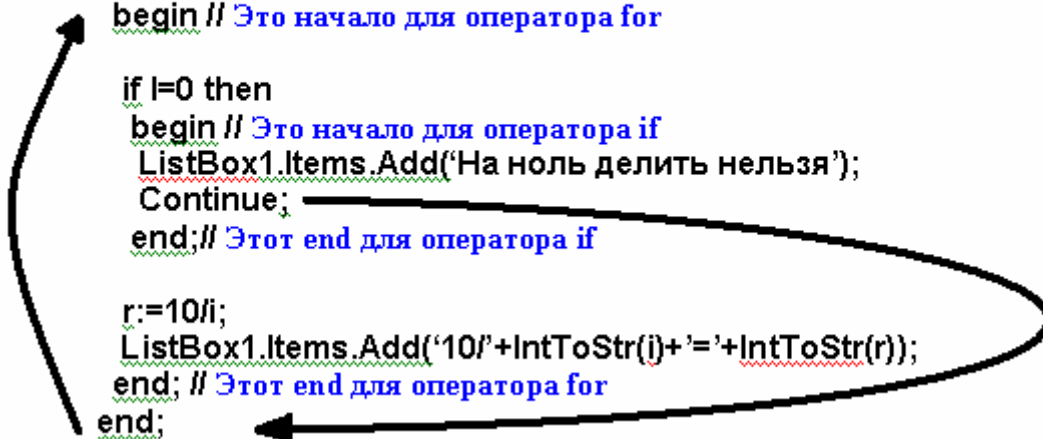
В этом примере я так же проверяю на каждом этапе значение переменной **i**. Если оно равно нулю, то я вывожу сообщение в *ListBox1* о том, что на ноль делить нельзя и выполняю оператор **continue**. Как только программа встречает такой оператор, то она сразу прерывает дальнейшее выполнение цикла и переходит на следующий шаг. Это то же самое, что выполнить команду: «Остановиться дальнейшее выполнение программы, увеличить значение переменной **i** и начать выполнение цикла со следующим значением».

Посмотри на следующий рисунок, где я показал стрелками процесс выполнения оператора **continue**:

```
var
  i, r: Integer;
begin
  for i := -3 to 3 do
    begin // Это начало для оператора for

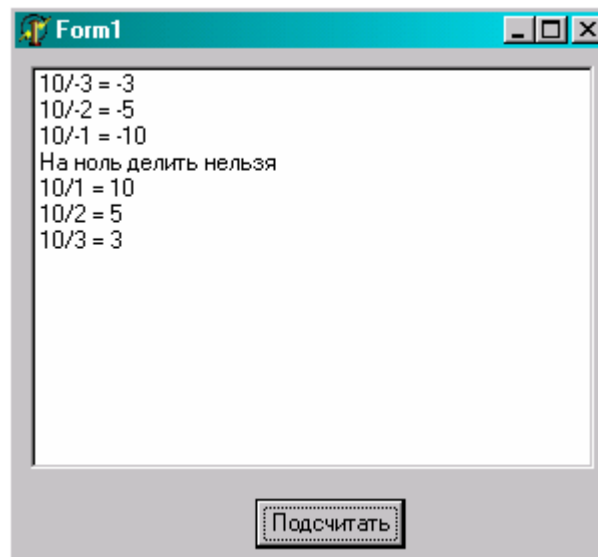
      if i = 0 then
        begin // Это начало для оператора if
          ListBox1.Items.Add('На ноль делить нельзя');
          Continue;
        end; // Этот end для оператора if


      r := 10 / i;
      ListBox1.Items.Add('10/' + IntToStr(i) + '=' + IntToStr(r));
    end; // Этот end для оператора for
  end;
```



Как только программа встречает оператор **continue**, она перескакивает на конец цикла, где увеличивается значение счётчика (в данном случае переменная **i**) и продолжается выполнение уже со следующего шага.

На следующем экране показана форма с результатом работы нашего примера:



 На компакт диске, в директории \Примеры\Глава 8\continue ты можешь увидеть пример этой программы.

Теперь давай разберёмся с оператором **break**. Как только программа встречает такой оператор, цикл прерывается и выполнение передаётся следующему оператору после цикла. Давай возьмём наш предыдущий пример и заменим в нём **continue** на **break**:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r: Integer;
begin
```

```
for i:=-3 to 3 do
begin // Это начало для оператора for

    if i=0 then
    begin // Это начало для оператора if
        ListBox1.Items.Add('На ноль делить нельзя');
        break;
    end; // Этот end для оператора if

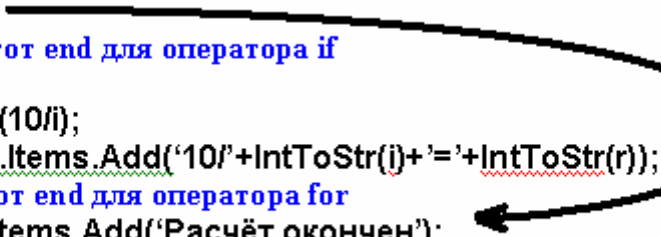
    r:=round(10/i);
    ListBox1.Items.Add('10/'+IntToStr(i)+'=' + IntToStr(r));
end; // Этот end для оператора for
ListBox1.Items.Add('Расчёт окончен');
end;
```

В этом случае, если *i* будет равно нулю, то выведется сообщение о том, что на ноль делить нельзя и цикл прервётся. После этого, управление передаётся следующему оператору после цикла:

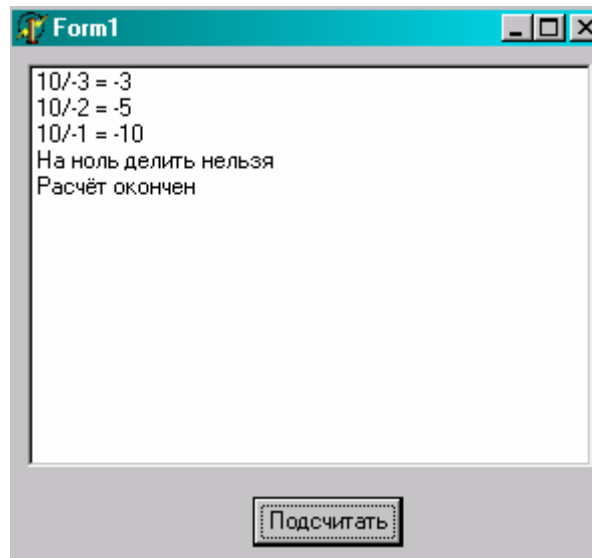
```
var
i, r:Integer;
begin
for i:=-3 to 3 do
begin // Это начало для оператора for

    if i=0 then
    begin // Это начало для оператора if
        ListBox1.Items.Add('На ноль делить нельзя');
        break;
    end; // Этот end для оператора if

    r:=round(10/i);
    ListBox1.Items.Add('10/'+IntToStr(i)+'=' + IntToStr(r));
end; // Этот end для оператора for
ListBox1.Items.Add('Расчёт окончен');
end;
```



На следующем рисунке показан экран программы нашего кода:



Как видишь, после встречи в нулём, в цикле больше ничего не выполняется.

 На компакт диске, в директории \Примеры\Глава 8\break ты можешь увидеть пример этой программы.

8.5 Логические операторы

С логическими операторами я тебя уже знакомил, и мы с ними уже достаточно много работали. Но здесь я дам более полную информацию по логическим операциям. Точнее сказать, по одной – **IF**.

Как ты уже знаешь, она выполняет проверку – « Если какое-то условие выполнено, то выполнить следующее за условием действие. Если нужно выполнить несколько действий, то их нужно объединить с помощью *begin...end*.

В общем виде логика выглядит так:

If Условие выполнено then
Действие1;

Если нужно выполнить два действия, то нужно написать так:

If Условие выполнено then
begin
 Действие1;
 Действие2;
end;

А если надо проверить сразу два условия? В этом случае можно поступить двумя способами:

If Условие1 выполнено then

**If Условие2 выполнено then
Действие1;**

Если условие один верно, то выполнится следующее за логикой действие, а это вторая проверка. Если вторая проверка условия верна, то выполнится действие. Если хотя бы одно из условий не выполнится, то цепочка прерывается, и действие не будет выполнено.

Есть ещё один способ:

**If (Условие1 выполнено) and (Условие2 выполнено) then
Действие1;**

В этом примере я объединил две проверки в одну. Если **Условие1** и **Условие2** верны, то выполнится действие.

А если тебе нужно выполнить действие, если хотя бы одно из условий верно? Не обязательно, чтобы оба сразу, а хотя бы одно. В этом случае можно для объединения использовать не **and**, а **or**. Это будет выглядеть так:

**If (Условие1 выполнено) or (Условие2 выполнено) then
Действие1;**

Если ты объединяешь два условия в один **if**, то их обязательно нужно оградить скобками. Если ты их не поставишь, то будет ошибка. Вот пример неправильного оформления:

**If Условие1 выполнено or Условие2 выполнено then
Действие1;**

В качестве условий можно применять следующие операторы:

Оператор	Описание	Пример использования
<	Меньше	Index < 10
>	Больше	Index > 10
==	Равно	Index == 10
<=	Меньше либо равно	Index <= 10
>=	Больше либо равно	Index >= 10

Существует одно исключение, при котором оператор может отсутствовать. Если ты проверяешь булеву переменную, то оператор можно опустить. Например:

```
var  
b:Boolean;  
  
begin  
b:=true;
```

```
if b then
  Выполнить действие;

end;
```

В этом примере, происходит проверка булевой переменной **b**. Но с чем её сравнивают, не указано. Как ты знаешь, булевы переменные могут принимать одно из двух значений: *true* или *false* (истина или ложь). Так вот, в таком случае происходит проверка на истину. Если булева переменная равна *true*, то действие будет выполнено.

До сих пор мы рассматривали сокращённый вид логики **if**. В полном виде она выглядит так:

```
If Условие выполнено then
  Действие1
else
  Действие2
```

В этом виде, если условие выполнено, то выполнится действие1, иначе выполнится действие 2.



*Я много раз говорил, что каждый оператор должен заканчиваться точкой с запятой. Запомни, что после любого оператора перед **else** точка с запятой не ставится.*


Давай рассмотрим пару примеров:

```
var
  i:integer;

begin
  if i>0 then
    i:=i+10 //Заметь, что точки с запятой нет.
  else
    i:=i+20

    if i<0 then
      begin
        i:=10;
        i:=i-2;
      end //Заметь, что точки с запятой нет.
    else
      begin
        i:=20;
        i:=i-2;
      end;
    end;
```

Вот теперь я рассказал всё необходимое о логических операциях и как с ними работать. Так что можно двигаться дальше и постигать что-то новое.

 На компакт диске, в директории \Примеры\Глава 8\if ты можешь увидеть пример программы, в которой используются различные типы логических операций.

При использовании логических операций – результат сравнения всегда должен быть логическим. Нельзя написать так:

```
if i:=0 then
```

Здесь в операторе используется присваивание, а у него нет результата, а значит, произойдёт ошибка. Чтобы не делать таких ошибок, просто запомни, что при использовании оператора **if** обязательно должны использоваться только логические операции равенства, больше, меньше и так далее. Единственный случай, когда можно опускать операторы сравнения – когда проверяется логическая переменная.

```
var
perem:Boolean;
begin
perem:=true;
if perem then
perem:=false;
end;
```

В этом примере нет никаких операторов типа равенства, больше или меньше, потому что переменная **param** – логическая и проверяется она. Если она равна true, то следующее действие выполниться, если нет, то пропуститься.

8.6 Работа со строками.

В этой части расскажу, как можно работать со строками. Мы поговорим об основных функциях работы со строками и применим их в действии. Для этого я напишу несколько полезных примеров, и мы разберём их по косточкам. Для начала, давай рассмотрим основные функции для работы со строками.

[Length](#)

Эта функция возвращает длину строки. У неё есть только один параметр – строка, длину которой надо вернуть. Функция **Length** выглядит так:

```
function Length(S): Integer;
```

Пример использования функции:

```
var
Str:string;
```

```
Index: Integer;  
begin  
  Str:='Привет с большого будуна';  
  index:= Length(Str);  
end;
```

В этом примере я объявил две переменные Str (строка) и index (целое число). В первой строчке кода я присваиваю в переменную Str строку «Привет с большого будуна». После этого я присваиваю переменной index длину строки Str. Результат, в переменной index будет число 24 – длина строки (если, конечно, я правильно подсчитал 😊).

Copy

Эта функция возвращает указанный отрывок строки. Например, тебе нужно получить из строки «Привет с большого будуна» символы начиная с 7-го по 10-й. Это легко сделать с помощью функции *copy*. У неё есть три параметра:

1. Строка, из которой нужно получить отрывок текста.
2. Начальный символ.
3. Количество нужных символов.

function Copy(S; Index, Count: Integer): string;

Рассмотрим пример:

```
var  
  Str1:string;  
  Str2:Integer;  
begin  
  Str1:='Привет с большого будуна';  
  Str2:= copy(Str1, 7, 10);  
end;
```

Здесь я объявил две строковых переменных: **Str1** и **Str2**. В первой строчке кода я присваиваю в переменную Str1 строку «Привет с большого будуна». В следующей строке, я копирую в переменную Str2 десять символов из Str1, начиная с седьмого символа. Результатом будет в **Str2** строка: «с большого».

Delete

Эта функция удаляет кусок текста из указанной строки. У неё есть три параметра:

1. Строка, из которой нужно удалить отрывок текста.
2. Начальный символ, начиная с которого будут удаляться символы.
3. Количество символов для удаления.

В общем виде функция выглядит так:

procedure Delete(var S: string; Index, Count: Integer);

Рассмотрим пример:

```
var  
  Str1:string;  
begin  
  Str1:='Привет с большого будуна';  
  Delete(Str1, 7, 10);  
end;
```

В этом примере я удаляю из строки *Str1* символы, начиная с 7-го по 10-й. В результате в переменной *Str1* останется только строка «Привет будуна».

Pos

Эта функция ищет указанные символы в строке. Если эти символы найдены, то она вернёт порядковый номер, начиная с которого найдена нужная строка. У функции два параметра:

1. Строка, которую надо искать.
 2. Строка, в которой надо искать.
- Если подстрока не найдена, то функция вернёт ноль.

function Pos(Substr: string; S: string): Integer;

Рассмотрим пример:

```
var  
  Str1:string;  
  index: integer;  
begin  
  Str1:='Привет с большого будуна';  
  index:=Pos('большого', Str1);  
end;
```

В этом примере я запускаю поиск строки «большого» в строке *Str1*. В данном случае, строка «большого» есть в строке переменной *Str1*, и начинается с 9-го символа. Результат, в переменной *index* будет число 9.

Insert

Эта процедура вставляет одну строку в другую, начиная с указанного символа. У неё есть три параметра:

1. Строка, которую надо вставить.
2. Строка, в которую надо вставить.
3. Позиция, куда надо вставить.

В общем виде функция выглядит так:

procedure Insert(Source: string; var S: string; Index: Integer)

Рассмотрим пример:

```
var
  Str1:string;
  index: integer;
begin
  Str1:='Привет с будуна';
  Insert('большого', Str1, 9);
end;
```

Здесь я вставляю в строку *Str1* текст «большого» начиная с девятого символа. Результатом будет строка «Привет с большого будуна».

На этом и закончим обзор функция для работы со строками и двинемся дальше по ступенькам к знаниям ☺.

8.7 Исключительные ситуации.

Пора нам уже познакомиться с исключительными ситуациями. Для чего они нужны? Допустим, что у тебя есть участок кода, где может произойти ошибка. Как сделать так, чтобы программа не вылетала при её возникновении? Очень просто. Надо заключить этот код в блок проверки исключений и тогда твоя программа выдержит даже цунами :).

И так. Простейший блок исключений выглядит как:

```
TRY
  //Здесь ты пишешь код, в котором может произойти ошибка
EXCEPT
  //Если ошибка произошла, то выполнится этот код
END;
```

Рассмотрим простейший пример:

```
TRY
  x:=y/0;
EXCEPT
  //Здесь можно вывести сообщение об ошибке.
END;
x:=0;
```

Между TRY и EXCEPT я вставил маленькое действие - деление на ноль. Компьютер не умеет делать такие вещи, поэтому произойдёт ошибка и выполнится код между EXCEPT и END. После обработки ошибки процедура заканчивает выполнение и все остальные операторы не будут выполнены, как, например, в нашем случае - **x:=0**;

Если бы мы поменяли 0 на любое другое число, то ошибки бы не было, и код между EXCEPT и END никогда не выполнялся бы.

Давай посмотрим ещё пример:

```
var
  b:Tbitmap
begin
  b:= TBitmap.Create;

  TRY
    b.Canvas.Rectangle(1,1,100,100);

  EXCEPT
    //Здесь можно вывести сообщение об ошибке.
    b.free;
    exit;
  END;
  b.free;
end;
```

В этом примере мы создаём объект *b* типа *Tbitmap* (картинка). Потом начинаем блок *TRY*. В этом блоке мы пытаемся начать рисование. Если во время рисования произошла ошибка, то можно сообщить об этом пользователю и освобождается память *b.free*. После этого происходит выход из процедуры. Если ошибок не было, то просто освобождается память *b.free*.

Теперь давай разберёмся с ещё одним типом исключительных ситуаций - **TRY ... FINALLY**:

```
TRY
  //Здесь ты пишешь код, в котором может произойти ошибка
FINALLY
  //Этот код выполнится в любом случае
END;
```

Между *TRY* и *FINALLY* ты пишешь свой сомнительный код, в котором может произойти ошибка. А между *FINALLY* и *END* ты пишешь код, который должен выполниться в не зависимости от результата кода. В этом случае мы не можем информировать пользователя об ошибке, потому что в разделе *FINALLY* мы не знаем, произошла ошибка или нет. Зато вот такой пример будет уместен:

```
var
  b:TBitmap
begin
  b:= TBitmap.Create;
  TRY
    b.Canvas.Rectangle(1,1,100,100);
  FINALLY
    b.free;
  END;
end;
```

Подобный пример, мы уже рассматривали. В этом случае мы создаем объект *b* и пытаемся нарисовать. В разделе *FINALLY* мы удаляем созданный объект *b*. Теперь мы уверены, что *b* всегда будет удалён корректно, и мы освободим память. Потому что код

между *FINALLY* и *END*, будет выполняться всегда, вне зависимости от произошедших ошибок.

Если бы все программы в Windows были написаны корректно и сомнительные участки кода заключались бы в блоки исключительных ситуаций, пользователь забыл бы, что такое синий экран смерти. Если ты собираешься писать коммерческое программное обеспечение, то ошибки в нём непростительны. Никто не будет покупать дырявые программы, которые будут вылетать через каждые пять минут. Это я тебе говорю из своего опыта.

Будь внимателен. Ошибки могут появиться даже в самых неожиданных местах. Пользователь очень часто может тыкать не туда, куда надо и сбивать твоё творение с пути праведного ☺. Конечно же, можно говорить, что пользователь безрукий, но этот безрукий тебе платит за то, чтобы программа работала. Поэтому от его безмозглых действий нужно предохраняться. Исключительные ситуации тут один из лучших способов.

Глава 9. Создание рабочих приложений.

В этой главе я уже начну рассказывать о том, как создавать самые настоящие приложения. До этого момента я писал какие-то примеры, но они были примитивны и отображали только самое необходимое.

Начнём мы рассмотрение главы с создания меню. Это единственное, что опустил при рассмотрении палитры компонентов «**Standard**». Сейчас уже настало время восполнить этот пробел.


После этого мы научимся создавать панели кнопок и научимся с ними работать. Я буду описывать примеры, и показывать, как их создавать с точки зрения эргономики и дизайна. Хотя художник из меня никакой, но хоть какую-то красоту в наших программах мы наведём.

Эта глава будет насыщена интересными примерами и их описаниями. Я не люблю особо грузить сухой теорией и стараюсь все свои слова подкреплять практикой. Получается как бы я «отвечаю за базар» ☺.




9.1 Создание главного меню программы.

Эту часть я буду описывать чисто на практике. Теорию будем поглощать по мере продвижения нашего примера. Давай создадим маленькую программу, которая будет содержать какое-то меню. Какое именно не особо важно, лишь бы научится с ним работать.

Создай новое приложение. Брось на форму один компонент **MainMenu** . Теперь посмотрим какие есть свойства у этого компонента:

- *AutoHotkeys* – будут ли создаваться автоматически клавиши быстрого вызова. Если ты выберешь *maAutomatic*, то Delphi будет автоматически создавать клавиши. При *maManual* придётся это делать вручную.
- *AutoMegre* – автоматическое слияние с дочерними окнами.
- *Images* – сюда можно подключать списки картинок, которые смогут отображаться на пунктах меню.
- *Items* – здесь описываются пункты меню.

Давай сразу подключим список картинок. Брось на форму компонент **ImageList** с закладки **Win32** . Теперь дважды щёлкни по нему и перед тобой откроется окно:

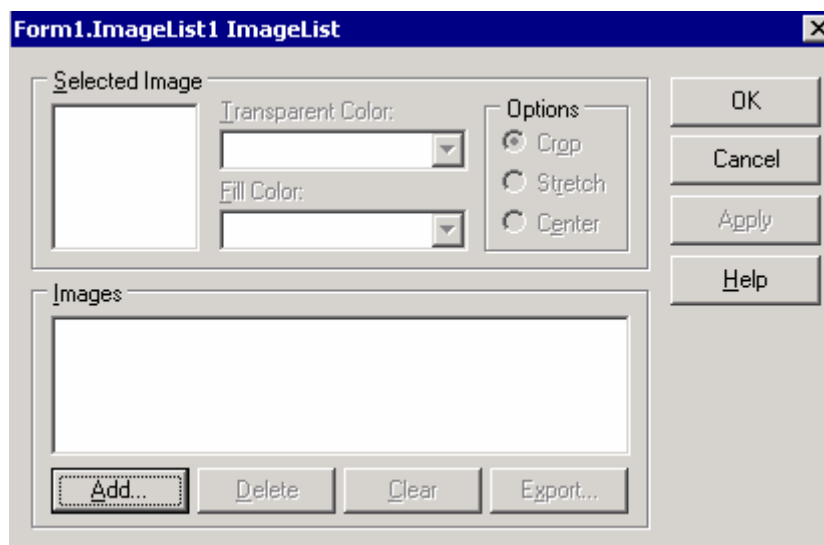


Рис 9.1 Добавление картинок

Здесь нажми кнопку *Add* чтобы добавить картинку. Откроется стандартное окно открытия файла. Открой какую-нибудь картинку, и она добавится в список. Желательно, чтобы она была размером 16x16. Именно такие габариты используются по умолчанию.

 На компакт диске, в директории **Images** ты можешь найти большое количество картинок для кнопок. А в директории **Примеры\Глава 8\Меню** находятся исходники примера и картинки, которые я использовал в нём.

Можешь таким образом добавить несколько картинок. Какие добавил я, смотри на рисунке:

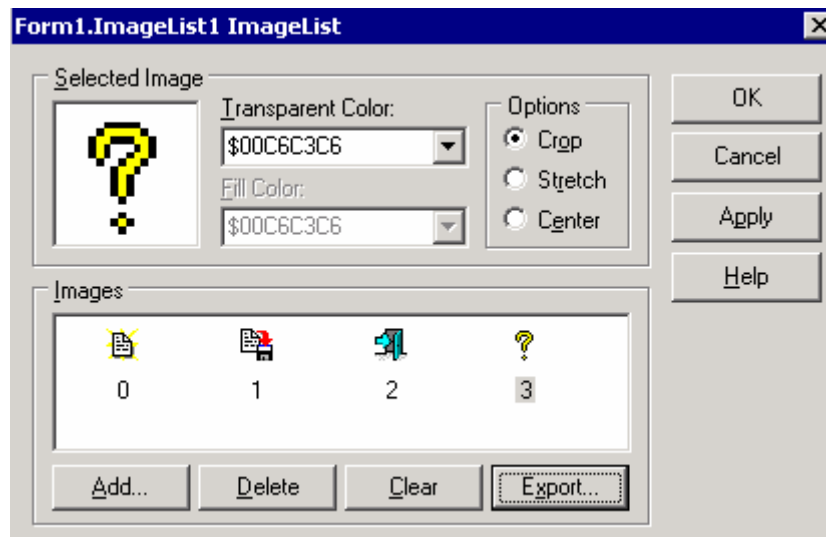


Рис 9.2 Заполненный список картинок.

Теперь подключим наш список картинок к менюшке. Выдели компонент *MainMenu1* и у свойства *Images*, в выпадающем списке выбери пункт *ImageList1*.

Теперь создадим само меню. Для этого дважды щёлкни по свойству *Items* и перед тобой откроется редактор меню:

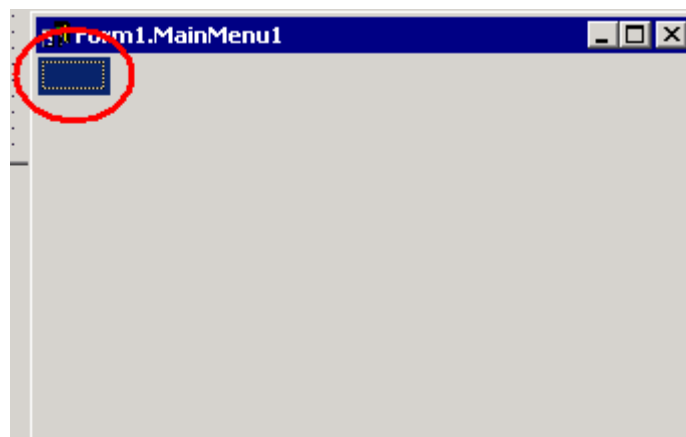


Рис 9.3 Редактор меню.

Этот же редактор можно вызвать, если дважды щёлкнуть по компоненту *MainMenu1*.

Красным кругом на рисунке я выделил уже созданный пункт. Перейди в объектный инспектор и набери в свойстве *Caption* слово «Файл». Как только ты нажмёшь кнопку Enter, будет создано меню «Файл»:

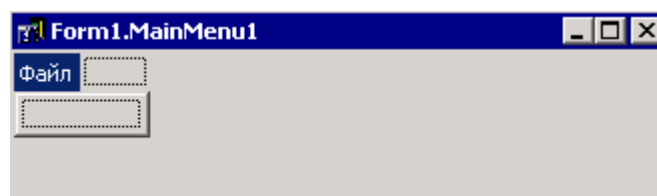
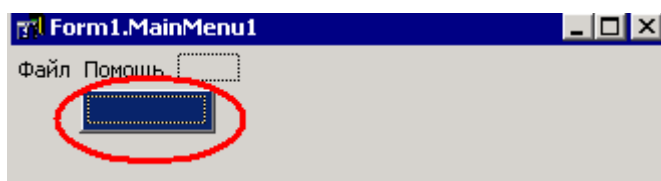


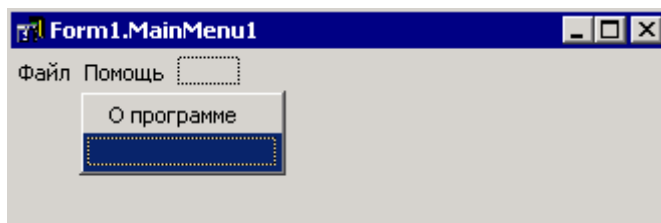
Рис 9.3 Меню «Файл».

Давай создадим ещё и меню «Помощь». Щёлкни справа от созданного меню (в рамочке обведённой пунктиром) и снова перейди в объектный инспектор. Там введи в свойстве *Caption* слово «Помощь».

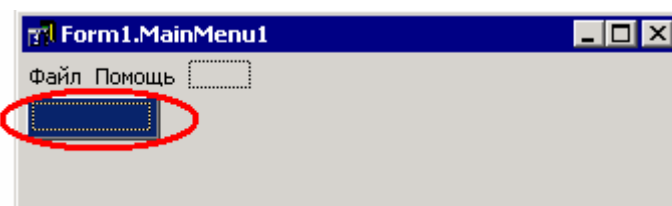
Теперь создадим подпункт для меню «Помощь». Щёлкни в рамке чуть ниже меню «Помощь», как показано на рисунке:



Здесь, в свойстве *Caption* мы введём слово «О программе». У тебя должно получиться что-то похожее на этот скрин:

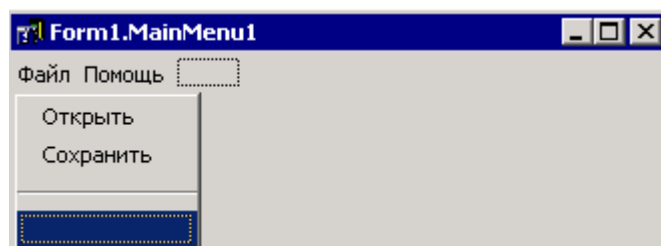


Теперь, таким же образом заполним меню «Файл». Выдели его. Теперь щёлкни в рамочке чуть ниже.



Здесь мы напишем в свойстве *Caption* слово «Открыть». Когда ты нажмёшь Enter или перейдёшь на другой пункт меню в редакторе, создастся пункт «Открыть» и тут же немного ниже создаётся пустой пункт. Щёлкни по нему и введи в свойстве *Caption* слово «Сохранить».

Теперь снова щёлкни на новом пункте меню и у него в свойстве *Caption* просто тире «-». Это заставит Delphi создать сепаратор:



И, наконец, создадим последний пункт – «Выход». Теперь назначим каждому пункту меню картинки. Я опишу, как это делать на примере пункта «Открыть», а остальные ты сделаешь сам.

Выдели пункт «Открыть». Теперь в объектном инспекторе щёлкни по выпадающему списку свойства «ImageIndex». Перед тобой откроется список всех картинок, которые ты подключил:

Hint	
ImageIndex	-1
Name	0
RadioItem	1
ShortCut	2
SubMenuItemImages	3
Tag	
Visible	True

Выбери тот, что тебе хочется, и картинка уже подключена. В редакторе меню картинка не будет видна, зато в редакторе форм ты сразу можешь увидеть её.

 На компакт диске, в директории \Примеры\Глава 9\меню ты можешь увидеть пример этой программы.

Теперь создадим обработчик события нажатия по пункту меню. Для этого выбери в дизайнера меню пункт «Выход» и щёлкни по нему дважды или перейди на закладку Events и дважды щёлкни по событию *OnClick*. Эти действия заставят Delphi создать обработчик события по нажатию меню. В этом обработчике напишем следующее:

```
procedure TForm1.N7Click(Sender: TObject);
begin
  Close;
end;
```

Здесь мы используем метод формы *Close*. Если я не ошибаюсь, то я уже говорил о нём. Если нет, то напомним, что этот метод закрывает форму. Если мы закрываем главную форму, то закроется всё приложение.

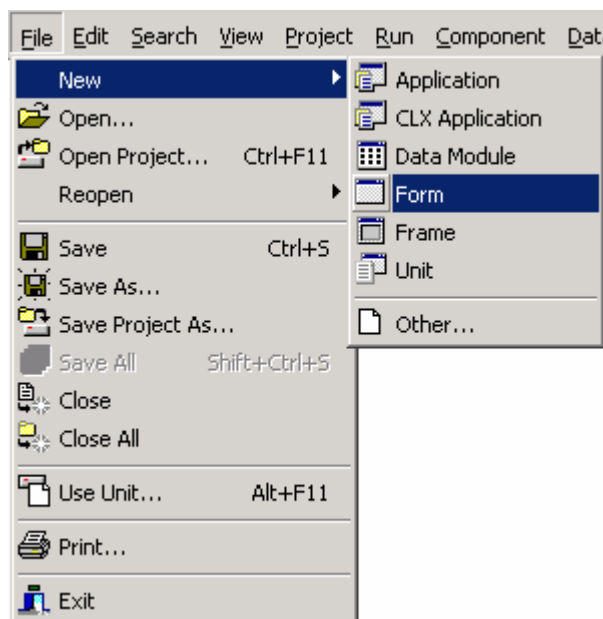
 На компакт диске, в директории \Примеры\Глава 9\меню1 ты можешь увидеть пример этой программы.

9.2 Создание дочерних окон.

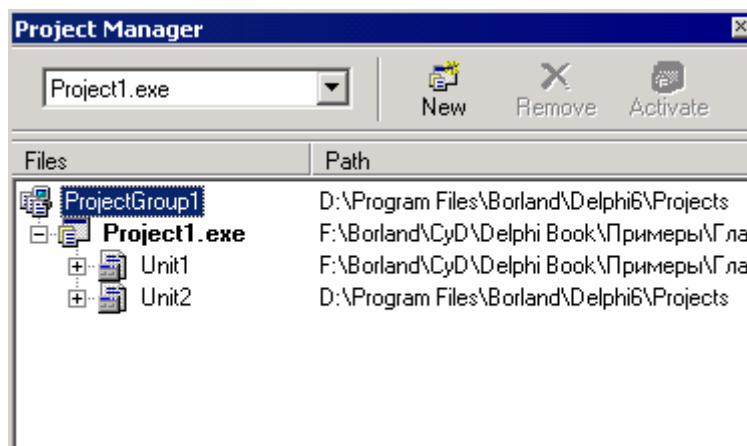
До сих пор мы создавали простые приложения, состоящие из одного только главного окна. В этой части главы я расскажу, как создать приложение, состоящее из одного главного окна, которое может вызывать дочернее окно.

Для примера я буду использовать исходник из предыдущей части главы, где мы создали меню. Открой тот проект.

Для начала создадим новую форму. Для этого, из меню File выбери пункт New, а затем выбери пункт Form, как показано на рисунке ниже. Это касается только Delphi 6. В более старых версиях нужно просто выбрать File -> New Form (я постараюсь всегда показывать отличия старых версий от Delphi 6).



Delphi должен создать новую чистую форму. Открой менеджер проектов (View -> Project Manager). Посмотри на его содержимое и убедись, что в твоём проекте Project1.exe теперь есть две формы: *Unit1* и *Unit2*:



Двойной щелчок по любой из форм в менеджере проектов вызовет форму в дизайнер для редактирования. В принципе, новая форма уже открыта, и нам не надо дважды щёлкнуть. Хотя можешь попробовать дважды кликнуть по *Unit1*, и Delphi моментально откроет эту форму для редактирования.

Давай сразу сохраним новую форму. Для этого при выделенной новой форме нажми Ctrl+S. Перед тобой появится окно для ввода имени формы. Я тебе говорил, что имена нужно задавать осмысленные, хотя сам это правило пока что постоянно нарушал. Теперь я этого делать не буду. Это окно у нас будет показывать информацию о программе, поэтому я назвал его AboutUnit.pas. Модуль главной формы я переименовал в MainUnit.pas.



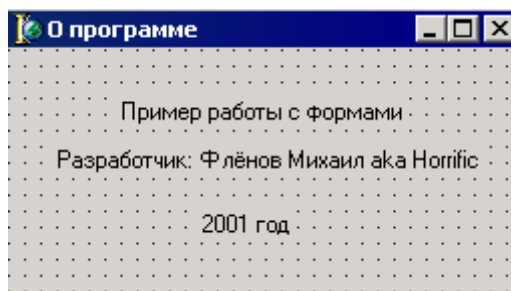
*Нельзя просто так переименовывать имена модулей. Для этого желательно использовать меню **File->Save As**.*

Сразу измени и имя формы с *Form2* на *AboutForm*.

Сохранили. Теперь изменим заголовок формы на «*О программе*».



Можешь ещё приукрасить как-нибудь эту форму. Лично я брошу несколько компонентов TLabel, чтобы сделать надписи. Но это уже не важно. Для нас главное научиться работать с этими формами.



Теперь нужно показать это окно. Давай создадим обработчик события *OnClick* для пункта меню «*О программе*» у нашей главной формы. Когда ты создашь обработчик, то Delphi даст процедуре тупое название типа *N4Click*. Число у тебя может отличаться. Как видишь, имя процедуры ничего нам не говорит. Но мы же договорились, что всё будем называть понятными именами, поэтому переименуй её в объектном инспекторе в *AboutClick*.

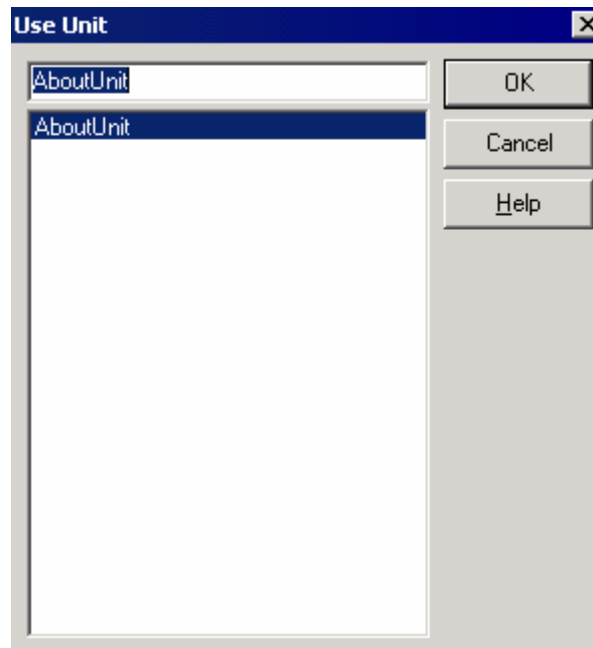


Теперь в получившемся обработчике напишем следующее:

```
procedure TForm1.AboutClick(Sender: TObject);  
begin  
  AboutForm.ShowModal;  
end;
```

В этом коде я вызываю метод *ShowModal* окна *AboutForm*. Этот метод показывает форму в режиме Modal. В этом режиме окно получает полное управление, и пока оно не закроется, главная форма не будет работать.

Если ты попробуешь сейчас откомпилировать код, то получишь ошибку. В Delphi 5 это будет просто ошибка, что *AboutForm* не найдена. Это потому, что эта форма описана в нашем модуле *AboutUnit*, а мы используем её в *MainUnit*. Чтобы *MainUnit* смог увидеть форму, описанную в *AboutUnit*, нужно её подключить. Для этого перейди в модуль *MainUnit* и из меню *File* выбери пункт *Use Unit*. Перед тобой откроется окно:



В этом окне нужно выбрать модуль, который ты хочешь подключить и нажать кнопку «OK». Что после этого изменится? Посмотрим на следующий отрывок кода:

```
var
    Form1: TForm1;

implementation

uses AboutUnit;

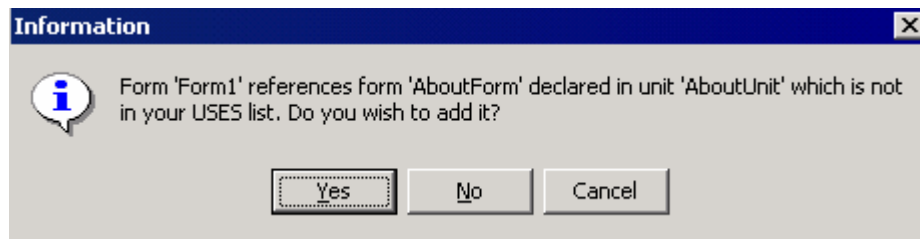
{$R *.dfm}
```

Как видишь, здесь появилась новая строчка *uses*. Точно такая же есть в начале модуля, но там мы подключаем стандартные заголовочные файлы. Здесь же мы подключаем модули написанные нами. В принципе, мы могли подключить модуль *AboutUnit* и в самом начале, но это делать не желательно.

В принципе, эту строку *uses* мы можем написать и вручную на этом месте и не выполнять никаких описанных мною выше действий. Так что выбирай, какой способ тебе удобнее – прописывать вручную или делать это автоматически.

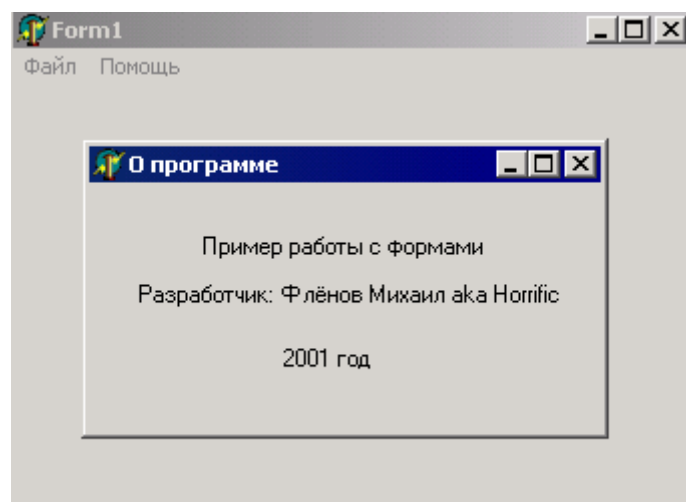
Теперь к форме *MainUnit* мы подключили наш модуль *AboutUnit* и можем смело использовать её содержимое.

Обладатели Delphi6 находятся в более удобном положении. Если ты забыл подключить модуль и откомпилировал код, то помимо ошибок ты увидишь следующее окно:



Здесь написано, что ты из главного модуля ссылаешься на форму *AboutForm*, которая объявлена в модуле *AboutUnit*. Тебе также предлагается подключить этот модуль. Если ты нажмёшь «Yes», то Delphi моментально сделает все действия для подключения.

Вот теперь можно компилировать код. Запусти программу и попробуй выбрать пункт меню «О программе». Если ты всё сделал правильно, то ты увидишь вторую созданную нами форму.



 На компакт диске, в директории \Примеры\Глава 9\Forms ты можешь увидеть пример этой программы.

9.3 Модальные и не модальные окна.

В предыдущем примере мы создали главное окно, которое вызывает дочернее в виде модального окна. Что значит модальное? Это значит, что управление полностью передаётся ему. Как только программа наткнется на код *AboutForm.ShowModal*, работа главной формы останавливается, и управление полностью передаётся дочерней форме. Пока модальное окно не закроется, главная форма работать не будет!!!

Рассмотрим простой пример:

```
procedure TForm1.AboutClick(Sender: TObject);
var
  Index: Integer;
begin
  AboutForm.ShowModal;

  Index:=10;
end;
```

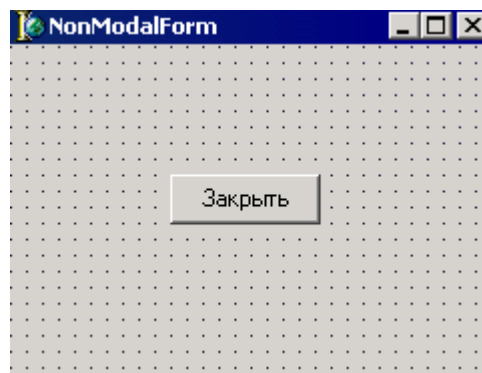
В этом примере я показываю модальное окно и после этого присваиваю переменной **Index** значение 10. Так вот, переменная **index** получит значение 10, но только после того, как модальное окно *AboutForm* закроется.

Для того чтобы создать не модальное окно, нужно вызвать метод **Show**. В этом случае главная форма создаст дочернее, показав его на экране, и смело продолжит выполняться дальше. Это позволит тебе работать с обеими формами одновременно, переключаться между ними и код обеих форм будет выполняться как бы параллельно. Рассмотрим пример:

```
procedure TForm1.AboutClick(Sender: TObject);
var
  Index: Integer;
begin
  AboutForm.Show;
  Index:=10;
end;
```

В этом случае я создаю немодальное окно, и выполнение кода не останавливается на точке *AboutForm.Show*, в ожидании закрытия окна, а спокойно продолжается дальше. То есть будет показано дочернее окно и моментально переменной *index* будет присвоено значение 10.

Давай создадим ещё одну форму. Сразу переименуем её свойство *Name* в *NonModalForm*. Можешь что-нибудь написать на ней, а я брошу только одну кнопку, с помощью которой можно будет закрыть это окно:



Сохрани новую форму под именем *NonModalUnit.pas*.

Теперь вернёмся в главную форму и допишем в раздел **uses** имя модуля *NonModalUnit*:

```
uses AboutUnit, NonModalUnit;
```

Если не хочешь подключать эту форму вручную, то выбери из меню *File* пункт *Use Unit*, и выбери там имя модуля для подключения.

Всё, модуль подключён, теперь можно его использовать. Создадим обработчик события для пункта меню «Сохранить» и напомним в нём следующее:

```
procedure TForm1.SaveClick(Sender: TObject);  
begin  
  NonModalForm.Show;  
end;
```

Здесь я показываю форму *NonModalForm* как немодальное окно. Это значит, что если ты запустишь программу и выберешь из меню пункт «Сохранить», то увидишь окно новой формы и сможешь спокойно переключаться между главной формой и *NonModalForm* без каких-либо проблем.

 На компакт диске, в директории \Примеры\Глава 9\Forms1 ты можешь увидеть пример этой программы.

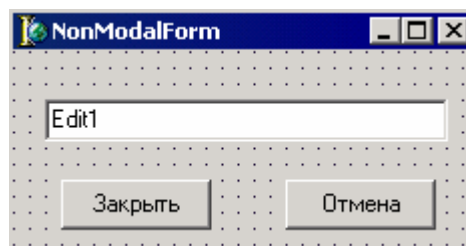
В последствии мы ещё очень много будем использовать оба типа окон (в основном модальные) и ты познакомишься с их работой более подробно.

9.4 Обмен данными между формами.

Зачем нужны все эти формы, если нельзя передавать параметры между ними и взаимно использовать процедуры. Когда мы вызываем какой-то диалог, мы хотим, чтобы пользователь ввёл в него какие-то данные. После этого мы должны получить введённые данные в главном окне и как-то обработать. Помимо этого, на диалоге очень часто располагаются кнопки «Да» и «Нет». Мы просто обязаны знать, какую кнопку нажал пользователь и в зависимости от этого обрабатывать ввод или нет.

В этой части главы мы закрепим всё то, о чём говорили в предыдущих частях про формы и научимся с ними работать. Пока мы знаем только, как их создавать и выводить на экран, а работа с ними осталась за кадром. Пора это исправить.

В прошлой части мы создали немодальное окно для пункта меню «Сохранить». Я немного изменил то окна, добавив на него строку ввода:



Теперь посмотрим на очень интересное свойство кнопки «Закреть» - *ModalResult*. В этом свойстве мы можем задавать значение, возвращаемое при закрытии окна. Давай выберем здесь «*mrOk*». Теперь если мы покажем окно как модальное и потом закроем его кнопкой «Закреть», то функция *ShowModal* вернёт нам значение *mrOk*.

Я ещё добавил на форму кнопку «Отмена», у которой свойство *ModalResult* я установил в *mrCancel*. Кстати, теперь ты должен очистить обработчики событий *OnClick*, для кнопок. Когда ты указал в свойстве *ModalResult* возвращаемое значение, кнопка уже автоматически умеет закрывать окно и не нужно создавать для неё обработчик *OnClick* и в нём писать метод *Close*.

В связи с этим, предлагаю изменить обработчик события *OnClick* для пункта меню «Сохранить»:

```
procedure TForm1.SaveClick(Sender: TObject);
begin
  if NonModalForm.ShowModal=mrOK then
    Application.MessageBox(PChar(NonModalForm.Edit1.Text),
      'Ты ввёл:', MB_OKCANCEL)
end;
```

Теперь построчно рассмотрим код. В первой строке я вызываю модальное окно и сразу проверяю возвращаемое значение. Если оно равно *mrOK* то выполняю следующее действие (if NonModalForm.ShowModal=mrOK then).

Вторая строка показывает стандартное окно диалога. Я это делаю с помощью метода MessageBox объекта Application. У этого метода три параметра:

- 1) Строка, которая будет показана внутри окна.
- 2) Строка заголовка окна.
- 3) Кнопки, которые будут на окне.
 - MB_OK – кнопка «ОК».
 - MB_OKCANCEL – кнопки «ОК» и «Отмена».
 - MB_RETRYCANCEL – кнопки «Повторить» и «Отмена».
 - MB_YESNO – кнопки «Да» и «Нет».
 - MB_YESNOCANCEL – кнопки «Да», «Нет» и «Отмена».

В качестве текста сообщения в окне я вывожу текст, введённый в строку ввода нашего модального окна (NonModalForm.Edit1.Text).

Теперь если пользователь нажмёт кнопку «Заккрыть» в модальном окне, то появится окно с введённым текстом. Иначе ничего не произойдёт.

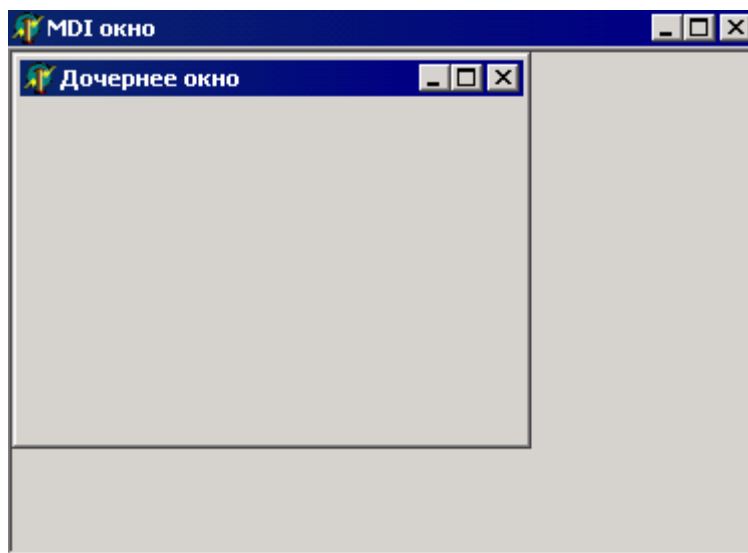
Вот и всё. В этом примере мы смогли показать окно, получить результат его выполнения и вывели введённый текст.

 На компакт диске, в директории \Примеры\Глава 9\Forms2 ты можешь увидеть пример этой программы.

9.5 Многодокументные MDI окна.

Что такое многодокументные MDI окна? Это когда главное окно содержит внутри себя несколько подчинённых окон. Дочерние окна чем то похожи на немодальные. Они так же не тормозят главное окно и работают независимо, только их область видимости ограничивается главным окном. Они находятся как бы внутри главного окна.

Хотя Microsoft уже не рекомендует использовать эту технологию, и убрала её из MS Word, сама её продолжает использовать. В Windows 2000 ярким примером MDI окон является консоль MMC.




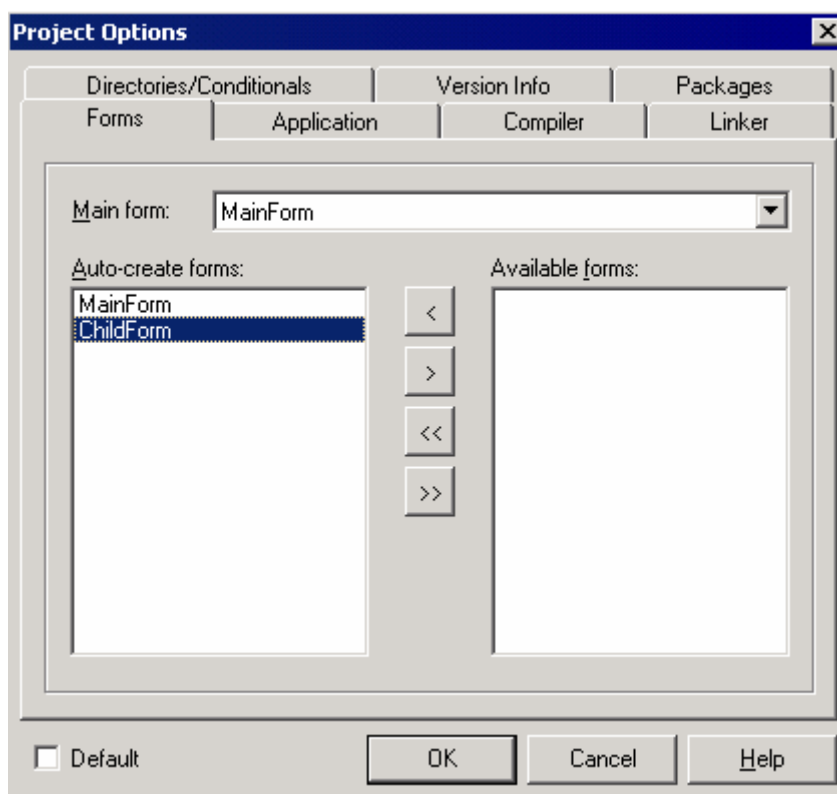
Пример MDI окна

Давай создадим простейшую MDI программу. Создай новое приложение. Сохрани главное окно под именем **MainModule**, а проект под именем **mdi**. Теперь измени свойство **FormStyle** у формы на **fsMDIForm**.


Теперь создай ещё одно окно (дай ему имя ChildForm) и измени у него свойство **FormStyle** у формы на **fsMDIChild**.

Вот и всё. Никакого геморроя, а MDI программа уже готова. Можешь запустить и посмотреть, как она работает.

 На компакт диске, в директории \Примеры\Глава 9\MDI ты можешь увидеть пример этой программы.

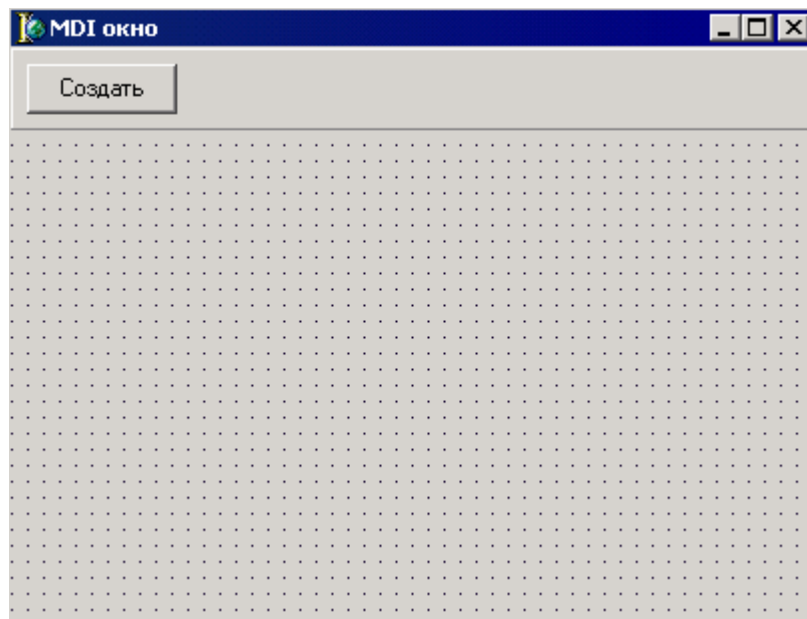


В нашем случае программа запускается и сразу создаётся дочернее окно. Как убрать его и создавать в рантайме (во время выполнения программы)? Очень просто выбери пункт меню «Options» из меню «Project» и ты увидишь следующее окно, показанное на рисунке выше.

В левой части окна перечислены те формы, которые будут создаваться автоматически (*Auto-create forms*). Выдели тут ChildForm (наше дочернее окно) и перемести его в список *Available forms*, нажав кнопку .

Теперь наша дочерняя форма не будет создаваться автоматически, и это придётся делать вручную. Ну, ничего, это не такая уж и проблема, как-нибудь справимся и победим эту проблему.

Брось на форму панельку и растяни её по верхнему краю окна (свойство Align надо установить в *alTop*). Теперь на панель бросим кнопку и дадим ей заголовок «Создать».



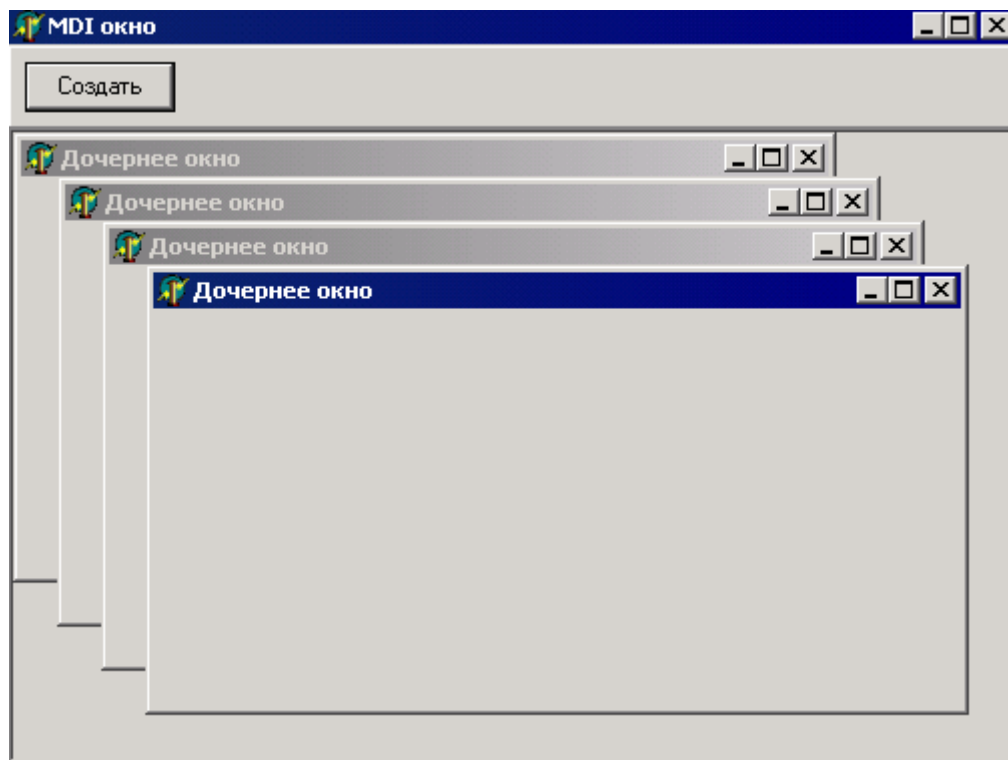
По нажатию этой кнопки мы будем вручную создавать окно

```
procedure TMainForm.CreateButtonClick(Sender: TObject);
begin
  ChildForm:= TChildForm.Create(Owner);
end;
```

Здесь я переменной ChildForm присваиваю указатель на новое созданное окно TChildForm.Create. Переменная ChildForm объявлена в модуле дочернего окна в разделе var:

```
var
  ChildForm: TChildForm;
```

Теперь запусти программу и попробуй несколько раз нажать на кнопку «Создать». У тебя должно создаться сразу несколько дочерних окон:



Но если ты попытаешься закрыть любое из них, оно просто свернётся. Чтобы окно закрывалось, нужно создать обработчик события **OnClose** для дочерней формы и в нём написать

```
procedure TChildForm.FormClose(Sender: TObject; var
    Action: TCloseAction);
begin
    Action:=caFree;
end;
```

Вот теперь наше приложение готово. В процессе книги мы ещё будем писать подобные приложения, поэтому успеем познакомиться с подобными приложениями.

 На компакт диске, в директории \Примеры\Глава 9\MDI ты можешь увидеть пример этой программы.

Напоследок дам несколько полезных свойств и методов формы, пример которых уже реализован в примере выше.

ActiveMDIChild – указывает на активное дочернее окно. Например, тебе надо изменить заголовок активной дочерней формы. Как тебе узнать, какая из них активная, когда их несколько? Очень просто. Создай в главной форме кнопку и по её нажатию напиши:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    ActiveMDIChild.Caption:='Активное дочернее окно';
end;
```

В этом коде я изменяю свойство *Caption* активной формы. Если нет активной дочерней формы (бывает, когда дочерних форм вообще нет), то свойство *ActiveMDIChild* равно **nil**.

MDIChildCount – целое число указывающее на количество дочерних окон.

MDIChildren – через это свойство можно получить доступ к любому дочернему окну. Например, второе окно можно получить с помощью *MDIChildren[2]*.

Давай попробуем изменить заголовки всех дочерних окон. Для этого запустим цикл от 0 до *MDIChildCount* и изменим все заголовки:

```
for i:=0 to MDIChildCount-1 do  
  MDIChildren[i].Caption:='Новый заголовок';
```

Есть ещё несколько интересных методов:

ArrangeIcons – выстроить иконки всех дочерних окон.

Cascade – выстроить каскадом все дочерние окна.

Next – Следующее дочернее окно.

Previous – Предыдущее дочернее окно

Tile – тоже выстроить дочерние окна.

9.6 Инициализация окон.

Вот теперь мы написали уже достаточно много примеров и готовы узнать, как инициализируются окна. В этой части я покажу: из чего состоит сердце нашей программы, где инициализируются окна и как управлять этим процессом. До этого момента я не хотел этого показывать, чтобы не забивать тебе голову, но теперь это необходимо, для продолжения разговора.

Создай новый проект. Сохрани модуль главной формы под именем *MainUnit.pas*, а проект под именем *SplashProject.dpr*. Теперь выбери из меню *Project* пункт *View Source*, чтобы увидеть исходный код проекта:

```
program SplashProject;  
  
uses  
  Forms,  
  MainUnit in 'MainUnit.pas' {Form1};  
  
{$R *.res}  
  
begin  
  Application.Initialize;  
  Application.CreateForm(TForm1, Form1);  
  Application.Run;  
end.
```

Всё это ничто иное, как содержимое файла *SplashProject.dpr*. Первой строкой стоит имя программы **program SplashProject**. В этой строке ничего менять нельзя. После этого идёт уже знакомый раздел **uses** в котором можно подключать необходимые модули. У нас подключен модуль *Forms* (позволяет работать с формами) и *MainUnit* (модуль главного окна).

Между **begin** и **end** выполняется три строчки:

1. *Application.Initialize* – запускает инициализацию приложения. Убирать не рекомендуется.

2. *Application.CreateForm (TForm1, Form1)* – метод *CreateForm* инициализирует форму. У него два параметра – имя объекта и имя переменной, которая в последствии будет указывать на созданный объект. В нашем случае это имя формы *TForm1* и имя переменной *Form1*. Переменная *Form1* автоматически описывается в модуле объекта *TForm1* (в нашем случае это модуль *MainUnit.pas*) в разделе **var**:

```
var  
  Form1: TForm1;
```

3. *Application.Run* – после инициализации всех форм можно запускать выполнение программы с помощью метода *Run* объекта *Application*.

Здесь везде используется объект *Application*. Этот объект всегда существует в твоих программах в единственном экземпляре и создаётся здесь с помощью строки *Application.Initialize*. С этим объектом мы будем знакомиться постепенно на протяжении всей книги, а сейчас достаточно этих знаний.

Теперь создай новую форму *File->New->Form* и сохрани её под именем *SplashUnit.pas*. Снова посмотри на исходник проекта, он должен быть уже таким:

```
program SplashProject;  
  
uses  
  Forms,  
  MainUnit in 'MainUnit.pas' {Form1},  
  SplashUnit in 'SplashUnit.pas' {Form2};  
  
{$R *.res}  
  
begin  
  Application.Initialize;  
  Application.CreateForm(TForm1, Form1);  
  Application.CreateForm(TForm2, Form2);  
  Application.Run;  
end.
```

В раздел **uses** добавилось объявление нового модуля, а между **begin** и **end** появился код создания новой формы.

Теперь войди в свойства проекта (из меню *Project* нужно выбрать пункт *Option*). На закладке *Forms*, в списке *Auto-create forms* у нас описано две формы. Выдели *Form2* и перенеси её в список *Available forms*. Закрой окно свойств кнопкой *OK* и посмотри на исходник проекта. Как видишь, строка инициализации второй формы исчезла. Это потому

что мы перенесли её из списка автоматически создаваемых форм в список доступных форм. Теперь, чтобы использовать *Form2*, мы её должны сначала создать.

Чтобы дальше можно было удобнее работать, переименуй главную форму *Form1* в *MainForm*, а вторую форму *Form2* в *SplashForm*. Теперь брось на главную форму кнопку и по её нажатию напиши следующий код:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  Application.CreateForm(TSplashForm, SplashForm);
  SplashForm.ShowModal;
  SplashForm.Free;
end;
```

Здесь, в первой строке кода я инициализирую форму *SplashForm*. Во второй, созданное окно выводится на экран. И в последней строке я уничтожаю окно.

Но есть ещё один способ создания окон, который мы уже использовали и я предпочитаю именно его:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  SplashForm:=TSplashForm.Create(Owner);
  SplashForm.ShowModal;
  SplashForm.Free;
end;
```

Здесь я присваиваю в переменную *SplashForm* результат вызова метода *Create* объекта *TSplashForm*. Этому методу нужно передать только один параметр – владельца окна. Если владельца нет, то можно передавать **nil**, а в нашем случае я передаю *Owner* – свойство, в котором храниться указатель на текущее окно. Если главным окном должно быть не текущее окошко, то нужно указать имя объекта – *Form1.Owner*.

Давай сделаем так, чтобы наше окно *SplashForm*, появлялось на время загрузки программы. Подобные окна ты видишь при старте Word, Excel и других приложений. Для этого зайти в исходник проекта и подкорректируй его до следующего вида:

```
begin
  SplashForm:=TSplashForm.Create(nil);
  SplashForm.Show;
  SplashForm.Repaint;
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Sleep(1000);
  SplashForm.Hide;
  SplashForm.Free;
  Application.Run;
end;
```

Давай рассмотрим этот код построчно:

1. Я создаю окно *SplashForm*. У этого окна нет владельца, поэтому в качестве параметра методу *Create* я указываю значение **nil**.

2. Отображаю окно на экране не модально.
3. Перерисовать окно с помощью вызова метода *Repaint*.
4. Инициализация приложения.
5. Создаётся форма *TMainForm*.
6. Делаю задержку, чтобы окно *SplashForm* могло хоть немного повисеть на экране. Для этого я использую процедуру *Sleep*, а в качестве параметра я указываю время задержки в миллисекундах. Одна секунда равна 1000 миллисекунд. Для использования этой функции в раздел **uses** нужно добавить модуль *Windows*.
7. Прячу форму *SplashForm*, вызовом метода *Hide*.
8. Уничтожаю окно.
9. Запускаю приложение.

Запусти программу, и ты сначала увидишь окно *SplashForm* (я на него поместил текст *TLabel* с надписью «Идёт загрузка»), а потом уже появится главное окно.

Когда создаются окна (вызывается *CreateForm*), то программа выполняет обработчики события *OnCreate* всех создаваемых форм. Если у тебя приложение слишком большое и происходят долгие операции в этих обработчиках, то я тебе советую показывать подобное *SplashForm* окно. В этом случае, первым делом создаётся именно оно и отображается на экране. Пользователь видит, что идёт загрузка и спокойно ждёт. Если ты не будешь информировать о ходе загрузки, и на экране ничего появляться не будет, то пользователь может подумать, что программа зависла. Поэтому лучше лишний раз создать такое окно, чем потом пользователи будут ругаться на твою программу.

 На компакт диске, в директории \Примеры\Глава 9\Splash ты можешь увидеть пример этой программы.