Проект "Вечное движение"

Классы TTimer(Таймер) **и** TShape **(Фигура)**

Во всех предыдущих проектах действия программы были ответом на действия пользователя. В ряде случаев программа должна работать сама, поэтому создадим простой проект, который иллюстрирует использование таймера для создания потока событий.

Поместим на Форму панель, добавим кнопки "Пуск", "Стоп" и таймер. Объект Timer находится на вкладке System (Системная) Палитры компонент. Основным свойством таймера является Interval — время срабатывания таймера в миллисекундах. По истечении этого промежутка времени возникает событие OnTimer. Таким образом, программа получает возможность периодически запускать процедуру обработки, в которой без вмешательства пользователя будут выполняться какие-то действия.

Панель является контейнером, внутри которого может перемещаться объект. В качестве такого выберем объект shape (Фигура), который размещается на вкладке Additional (Дополнительная).

Вид фигуры задается свойством Shape, которое может принимать следующие значения:

stCircle — круг;
stEllipse — эллипс;
stRectangle — прямоугольник;
stRoundRect — прямоугольник со скругленными углами;
stRoundSquare — квадрат со скругленными углами;
stSquare — квадрат.

Для задания цвета и способа заливки используется свойство Brush (Кисть). Выполним в Инспекторе объектов двойной щелчок на плюсе слева от слова Brush, тогда появятся составляющие этого свойства, которые можно задать обычным образом.

Как же мы сможем перемещать объект Shape? Очень просто: наша задача — изменить координаты фигуры на панели, то есть изменить свойства Left и top. Будем прибавлять к Left величину смещения по оси *Ox* — dx, а к Top — величину смещения по оси *Oy* — dy. Тогда фигура будет двигаться по прямой, по направлению вектора (dx, dy). Вторая задача — добиться, чтобы фигура отражалась от границ панели. Для этого в момент соприкосновения фигуры с вертикальными границами будем менять знак dx, а при соприкосновении фигуры с горизонтальными границами — знак dy на противоположный. При написании процедуры учтем, что правая и нижняя границы фигуры вычисляются по формулам Left+width и Top+Height соответственно.

Приведем полный текст процедуры.

```
procedure TForml.TimerlTimer(Sender:  TObject);
begin
with Shape1 do
 begin
   if  (Left <= 0)  or  (Left + Width >= Panell.Width) then dx  := -dx; Left  : = Left + dx;
   if  (Top <= 0)   or   (Top + Height >= Panell.Height) then dy  := -dy; Top  := Top + dy;
 end ;
end;
```

Попробуем запустить программу, но, увы, она даже не пройдет компиляцию! В нижней части модуля появится сообщение:
Undeclared indentifier: 'dx'.

Да, мы ведь забыли описать переменные dx и dy и задать их значения. Но это нельзя сделать внутри процедуры, так как значения этих переменных должны сохраняться между вызовами! (Попробуйте это сделать и понаблюдайте странное поведение шарика.)

Опишем поэтому эти переменные *вне* функции — в разделе констант секции **implementation:**

```
const
dx: integer = 5; dy: integer =5;
```

— и программа заработает правильно: шарик будет двигаться без остановки, отражаясь от границ Панели.

Займемся теперь кнопками "Пуск" и "Стоп". Кнопка "Пуск" включает таймер, устанавливая свойство Timer1.Enabled = **False.** Кнопка "Стоп" выключает таймер, устанавливая свойство Timerl .Enabled = **True.** Начальное значение свойства — **False** — установим в Инспекторе объектов. Тогда при запуске программы шарик двигаться не будет, так как нет сигналов от таймера. Движение начнется при нажатии кнопки "Пуск" и прекратится при нажатии кнопки "Стоп".

Добавьте в проект Таблицу цветов для изменения цвета фигуры во время работы программы.

Проект "Меню"

Класс TMainMenu. *Добавление меню к программе сложения чисел. Реакция программы на выбор пункта меню*

Меню как элемент интерфейса используется практически во всех серьезных приложениях. В Windows поддерживаются два типа меню - строчное, которому соответствует компонент MainMenu, и всплывающее (или локальное) — ему соответствует компонент PopupMenu. Обычно эти типы используются в комбинации.

Main Menu позволяет поместить главное меню в программу. При размещении на форме этот компонент выглядит как иконка. Создание меню включает три шага:

— помещение MainMenu на Форму:

— вызов Конструктора меню двойным щелчком по значку (или через свойство Items в Инспекторе объектов);

— определение пунктов меню.

Каждый пункт меню имеет имя (по умолчанию N1, N2 и т.д.) и название (свойство Caption). Но Конструктор меню так устроен, что сначала нужно ввести название пункта, только после этого пункт меню создается реально.

После создания первого пункта есть две возможности: двигаться вниз, создавая раскрывающийся список подпунктов, или двигаться вправо, создавая следующий пункт главного меню.

PopupMenu позволяет создавать всплывающее меню, которое появляется по щелчку правой кнопки мыши на объекте, к которому оно привязано. У всех видимых объектов имеется свойство Popup Menu, где и указывается нужное меню. Создастся PopupMenu аналогично главному меню.

Пункты главного располагаются под заголовком Формы. Внешний вид Формы с меню на этапе разработки приведен на рисунке. Маркерами (восемью черными квадратиками) выделен объект MainMenu1. Этот объект может размещаться в любом месте формы, так как он невидим при работе программы. Такие объекты называются «невизуальными компонентами».

Чтобы программа реагировала на выбор пункт меню, следует написать для каждого "конечного" пункта меню процедуру обработки данного события. Таким образом, нам придется написать четыре почти одинаковые процедуры, гак как каждая должна получить числовое значение полей ввода, произвести операцию над этими значениями и вывести результат. Отличаться эти процедуры будут только выполняемыми операциями.

Поэтому напишем отдельную процедуру, которую будем вызывать из всех четырех пунктов меню. В качестве параметра ей будем предавать символ операции: «+», «-», «*» или «/».

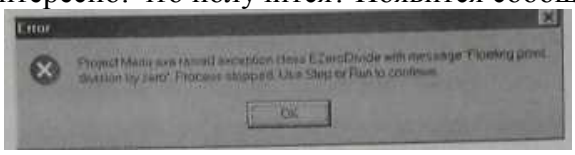
```
procedure Exec(ch: char) ;
var
 A, B, C: real;
 code: integer;
 S: string;
 begin
    Val(Form1.Edit1.Text, A, code);
    Val(Form1.Edit2.Text, B, code);
   case ch of
    '+':C:= A + B;
    '-':C:= A - B;
    '*':C:= A * B;
    '/' :C := A / B
   end;
   Str (C:8:2, S);
  Form1.Panel1.Caption:=S
end;
```

Тогда "обработчик" пункта меню "Сложить" будет выглядеть гак:

```
procedure TForml.NlClick(Sender: TObject);
begin
  Exec('+')
end;
```

Аналогично пишутся обработчики других пункте меню. Запустим программу и поработаем с ней:

Все идет хорошо, пока мы не решим поделить па ноль. Конечно, все знают, что этого нельзя делать, но интересно: что получится? Появится сообщение о делении на моль

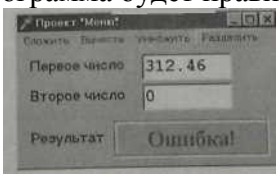


(Проект *Menu.exe* вызвал исключение "Деление на ноль". Процесс остановлен. Используйте Step или Run для продолжения.)

Чтобы избежать возникновения исключительной ситуации, добавим в оператор выбора проверку второго числа при делении. Если оно равно нулю, то на панель поместим слово "Ошибка". Окончательный текст процедуры станет таким:

```
procedure Exec(ch : char);
var
  A, B, C : real;
  code : integer;
  S : string;
begin
  Val(Form1.Edit1.Text, A, code);
  Val(Form1.Edit2.Text, B, code);
  case ch of
    '+': C:= A+B;
    '-': C:= A-B;
    '*': C:= A*B;
    '/': if B<>0 then C:= A/B
         else begin
              S:= 'Ошибка!';
              Form1.Panel1.Caption:=S;
              exit;
         end;
  end;
  Str(C:8:2, S);
  Form1.Panel1.Caption:=S;
end;
```

Теперь программа будет правильно работать даже при попытке деления на ноль.



Проект "Параметры шрифта"

Классы TLabel и TButton. Понятие события. Реакция программы на событие. Изменение свойств объекта программным путем

Познакомимся теперь с некоторыми готовыми компонентами — классами Delphi. Палитра содержит около сотни компонент (в последних версиях **Delphi** — еще больше). Знакомство с ними начнем с классов TLabel и **TButton,** размещенных на вкладке Стандартная (standard).

TLabel — класс "Надпись".

Используется для нанесения надписей, поясняющего текста непосредственно на форму.

TButton — класс "Командная кнопка".

Обычно при нажатии такой кнопки выполняется некоторое действие. Но как наше приложение узнает, что нужно сделать? При каждом действии пользователя (нажатии на клавишу клавиатуры, кнопку мыши, перемещении мыши и т.д.) возникает *событие,* которое через Windows передается приложению. Например, при нажатии на кнопку мыши возникает событие OnMouseDown, при отпускании — OnMouseUp. Можно рассматривать два этих действия как один "щелчок" — тогда это событие называется Onclick.Чтобы отреагировать на какое-либо событие, приложение должно вызвать соответствующую процедуру. Именно эти процедуры и пишет программист!

Поместим на Форму надпись Label1 и две кнопки — Buttonl и Button2. (Еще раз напомним, что имена "по умолчанию" объектам дает Delphi.)

Многие свойства этих компонент нам уже известны. Это прежде всего свойства Height, width, Left и Top, характеризующие размер и положение компонента. Свойство TLabel .Caption содержит текст метки, TButton.Caption — надпись на кнопке. Свойство Color есть только у метки, и задает оно цвет фона, на котором расположен текст.

Найдем в списке свойств метки ParentColor. По умолчанию это свойство имеет значение **True,** т.е. цвет метки будет таким же, как и цвет объекта, которому принадлежит метка. Если изменить цвет метки, то свойство ParentColor изменит свое значение на **False.**

Но главным свойством, которое мы рассмотрим на этом занятии, будет свойство Font. Слева от этого слова в Инспекторе объектов стоит знак "плюс". Это значит, что свойство Font представляет собой целый набор других свойств. Щелкнем дважды по плюсу — ниже появится перечень составляющих (при этом плюс изменится на минус).

Из них для выполнения проекта нам понадобится только свойство Size, определяющее размер шрифта.

Начальные значения параметров шрифта можно установить и другим путем. При выделении свойства Font в Инспекторе объектов в поле ввода появляется кнопка с тремя точками. При нажатии на эту кнопку появляется диалоговое окно для выбора шрифта, в котором легко установить все параметры, даже не зная их названий.

Напишем на одной из кнопок "Увеличить шрифт". Перейдем в Инспекторе объектов на вкладку Events и дважды щелкнем по пустому полю справа от слова Onclick. В Редакторе кода появится заготовка процедуры, реагирующей на нажатие кнопки:

```
procedure TForml.ButtonlClick(Sender: TObject);
 begin

end;
```

TForml — это имя класса, в который включена процедура ButtonlClick. Имя процедуры состоит из имени объекта Buttonl и части имени события Onclick. Параметром процедуры является отправитель сообщения. Для написания процедур обработки событий этот параметр используется редко, так что мы пока не будем обращать на него внимания.

Для того чтобы увеличить размер шрифта, в тело процедуры вставим одну строчку кода:

```
Labell.Font.Size := Labell.Font.Size*2;
```

При каждом нажатии кнопки размеры метки будут увеличиваться в два раза. Но положение левого верхнего угла метки останется неизменным. Из-за этого текст будет уходить за правую границу Формы. (Правда, при этом автоматически появятся полосы прокрутки, позволяющие заглянуть за "горизонт".)

Ограничим поэтому максимальный (а заодно и минимальный) размер шрифта. В раздел **implementation** добавим две константы:

```
const MaxSize = 200;
      MinSize = 6;
```

Начальное значение размера шрифта (например, 24) можно установить в Инспекторе объектов.

В процедуре TForml .ButtonlClick перед присваиванием добавим проверку:

```
if Labell.Font.Size*2 <= MaxSize then
   Labell.Font.Size := Labell.Font.Size*2;
```

Для того чтобы не писать каждый раз префикс Labell. Font, можно воспользоваться оператором with:

```
with Labell.Font do begin
  if Size*2 <= MaxSize then
     Size := Size*2; end;
```

Аналогичную процедуру легко написать и для второй кнопки.

Запустим проект и убедимся, что кнопки увеличения и уменьшения размера шрифта работают правильно, то есть, когда размер шрифта достигает предельного значения, дальнейшего изменения размера не происходит, — наши нажатия на кнопку не приводят ни к каким действиям. Но тогда не надо и нажимать на кнопку! Попробуем в этой ситуации ее "выключить".

Для этого необходимо установить значение свойства кнопки Enabled равным **False.** Но сделать это нужно по результатам прогноза размера надписи "на шаг вперед":

```
with Labell.Font do begin
 if Size*2 <= MaxSize then Size := Size * 2
 if Size*2 > MaxSize then Buttonl.Enabled := False;
end;
```

Но как после этого восстановить работоспособность кнопки Buttonl? Включить кнопку так же просто, как и выключить, достаточно выполнить присваивание Buttonl. Enabled := **True;** нужно только понять, куда его вставить. Понятно, что если кнопка Buttonl выключена, то включить ее можно только с помощью кнопки Button2, и наоборот. Добавим поэтому в процедуру TForml. Button2Click строку:

```
Buttonl.Enabled := True;
```

и все получится!

Напишите процедуру TForml. Button2Click для уменьшения размера шрифта.

Внешний вид работающего проекта при максимальном размере шрифта представлен на рисунке.

Проект "Подбор цвета"

Формирование цвета из отдельных компонент.

Класс TColor, константы цвета, **функция RGB**

Цвета объектов образуются смешением грех компонент — красной (*red*), зеленой (*green*) и синей *(blue).* Интенсивность каждой составляющей цвета может изменяться or 0 до 255. Комбинация (0, 0, 0) соответствует черному, а (255, 255, 255) белому цвету.

Почти у каждого визуального компонента есть свойство Color. До сих пор мы выбирали его значение из списка стандартных цветов, но ничто не мешает создать цвет из отдельных компонент. Для этого воспользуемся функцией RGB:

Color := RGB (*red, green, blue) ;*

Можно создать также свою цветовую гамму, заранее заготовив цвета для различных визуальных объектов. Но использовать эти цвета можно будет только при создании соответствующего объекта на этапе выполнения программы (об этом поговорим немного позже).

Для подбора цвета разработаем проект, позволяющий легко изменять цвет панели с помощью полос прокрутки — объектов класса TScrollBar. Поместим на форму панель и три полосы прокрутки (они также находятся на вкладке standard). Каждая полоса прокрутки будет отвечать за интенсивность одной из трех цветовых компонент. Крайнее левое положение ползунка должно соответствовать минимальному, а крайнее правое --- максимальному значению интенсивности.

Зададим для всех полос свойство Min=0, а свойство Max=255. Настроим другие свойства:

Kind — определяет размещение полосы — горизонтальное (sbHorizontal) или вертикальное (sbVertical);

LargeChange — шаг перемещения ползунка при щелчке на самой полосе;

Smal IChange — шаг перемещения ползунка при щелчке на стрелке;

Position — числовой эквивалент положения ползунка на полосе скроллинга.

Основное событие для полосы скроллинга — перемещение ползунка (событие OnChange), при этом способ перемещения значения не имеет.

Напишем отдельную процедуру изменения цвета панели

```
procedure SetPanelColor;
var red, green, blue : byte;
begin
  red := Form1.ScrollBarl.Position;
  green := Forml.ScrollBar2.Position;
  blue := Forml.ScrollBar3.Position;
  Forml.Panel1.Color := RGB(red, green, blue);
  end;
```

и будем ее вызывать в ответ на перемещение ползунка на любой полосе скроллинга:

```
procedure TForm1.ScroilBarlChange (Sender : TGbjecU);
begin
   SetPanelColor;
end;

procedure TForm1.ScrollBar2Change(Sender: TObject);
begin
   SetPanelColor;
end;
procedure TForm1.ScrollBar3Change(Sender: TObject);
begin
   SetPanelColor;
end;
```

Проект готов, можем запустить и поработать с ним. Вариант оформления проекта приведен на рисунке.

Добавьте справа от каждого ScrollBar по одному Label для отображения численных значений переменных *red, green, blue* и добавьте соответствующие операторы в процедуру для изменения цвета панели.

Проект "Редактор текста"

Класс TMemo, стандартные диалоги для работы с текстовыми файлами: TOpenDialog, TSaveDialog. Установка параметров шрифта с использованием диалога TfontDialog

Компонент Memo, в отличие от строки ввода Edit, может содержать множество строк, то есть предназначен для работы с большими текстами. В Memo можно переносить слова, сохранять в буфере обмена фрагменты текста и восстанавливать их, выполнять другие базовые функции текстового редактора. При этом объем текста не должен превышать 32 Кб. Свойства BorderStyle, Readonly, HideSelection, MaxLenght —те же, что и у строки ввода.
Специфические свойства:
Lines — содержимое Memo как набор строк; Align — заполнение пространства формы по краям; AlignMent — выравнивание текста внутри формы; ScrollBars — наличие полос прокрутки содержимого поля; Wordwrap — автоперенос текста от правого края.
Для удобства работы с полем примечаний как с текстовым редактором имеются свойства WantTabs и WantReturns, которые "восстанавливают в
правах" функции клавиш Enter и Tab I применительно к работе с текстом.
Строки текста (свойство Lines) вводятся в редакторе строк, который вызывается при нажатии на кнопку с тремя точками. Но их можно загружать из файла и выводить в файл. Для этого Lines использует методы LoadFromFile ***{имя файла)*** -- загрузить из файла и SaveToFile ***{имя*** *файла)* — записать в файл.
Пришло время познакомиться с очень удобным средством — стандартными диалогами. Это диалоги открытия и сохранения файла для **текстовых** и графических файлов, установки цвета, параметров шрифта и т.д. Все они размещены на вкладке Dialogs Палитры компонент.
Все диалоги относятся к невизуальным компонентам, поэтому могут быть размещены в любом **месте** формы, в том числе и на других объектах.
Начнем создание проекта "Редактор текста".
Разместим на форме объект Memol. Установим свойства ScrollBars = = ssVertical, Align = alClient (полное заполнение формы). Добавим диалоги OpenDialogl, SaveDialogl и FontDialogl. Настроим диалоги открытия и сохранения файла. Настройка сводится к заданию свойства Filter. Это свойство определяет, какие файлы "увидит" диалог при запуске. Для настройки свойства выделим его поле в Инспекторе объектов и щелчком на трех точках развернем Редактор фильтра.

В левый столбец заносится поясняющая строка, а в правый — шаблон имени файла. В нашем примере таких строк две. В первой строке (она будет видна в поле Тип файлов при открытии диалога) заданы только текстовые файлы, во второй — все файлы.

Аналогично настраивается диалог SaveDialogl. В нем только дополнительно зададим свойство DefaultExt » txt, чтобы расширение *txt* автоматически добавлялось к имени при записи файла.

Для удобства управления **проектом** создадим меню, при выборе пунктов которого будут вызываться диалоги.

Разделитель вставляется в меню, если в поле Caption в Инспекторе
объектов ввести дефис (знак минус).

Осталось только написать реакцию программы на выбор пунктов меню.

Основное назначение диалога открытия (закрытия) файла — выбрать файл. Для этого нужно запустить выполнение диалога. Это делает метод Execute, принадлежащий каждому диалогу. Метод возвращает **True,** если пользователь завершил диалог нажатием кнопки "Ok", **и False,** если пользователь нажал кнопку "Cancel".

Полное имя выбранного файла (то есть вместе с путем) помещается в свойство FileName **диалога и** может быть использовано при **чтении** (записи) текста. Для удобства работы имя файла можно поместить и в заголовок формы. Все описанное реализует следующий код:

```
with OpenDialog1 do begin
  if Execute then begin
    Memo1.Lines.LoadFromFile(FileName);
    Form1.Caption := 'Проект "Редактор текстов"' +FileName;
```

```
  end;
end;
```
Напишите самостоятельно код процедуры сохранения файла. Диалог для выбора параметров шрифта программируется аналогично, только вместо имени файла необходимо использовать свойство Font:

```
with FontDialog1 do begin
  if Execute then Memo1.Font := Font;
end;
```

При изменении размеров окна поле Memo1 также изменяет свои размеры благодаря тому, что свойство Align имеет значение alClient. При этом текст также переформатируется.

Проект "Рисование картинки"

Класс TImage, стандартные диалоги для работы с картинками.
Класс Canvas (холст), инструменты рисования (Brush и TPen,
графические примитивы

Проект "Рисование картинки" предназначен для простейших манипуляций с изображениями: рисования простых изображений, сохранения их в файле и загрузки из файла. Основой проекта является класс TImage — набор данных и методов для работы с изображениями в формате *bmp.*

Основными являются свойства Picture и Canvas. Свойство Picture и есть картинка. Ее можно задать в Инспекторе объектов с помощью загрузчика картинок, который, хотя и называется Picture Editor (Редактор картинок), никаких операций редактирования не выполняет.

Для загрузки и сохранения картинок в процессе работы приложения у свойства Picture есть методы LoadFromFile и SaveToFile. Для выбора имени файла на вкладке Диалоги имеются специализированные диалоги OpenPictureDialog И SavePictureDialog. Все это нам знакомо по предыдущему проекту.

Итак, разместим на форме Панель. Она будет играть роль рамки для картины, поэтому установим Bevellnner = bvNone, BevelOuter = bvLovered. Поместим внутрь объект Image (он находится на вкладке Дополнительная). Добавим диалоги для загрузки и сохранения графических файлов. Добавим меню, чтобы обеспечить выполнение диалогов.

Настроим диалоги.

Свойство Filter уже настроено. Оно содержит несколько вариантов:

Filter Name	Filter
All (*.bmp; *.ico; *.emf; *.wmf)	*.bmp; *.ico; *.emf; * .wmf
Bitmaps (*.bmp)	*.bmp
Icons (*.ico)	*.ico

Настроим еще два полезных свойства. Свойство InitialDir (Начальный каталог) указывает, в какой каталог мы попадаем при вызове диалога. Свойство Def aultExt позволяет не вводить расширение при записи файла — расширение, заданное в свойстве, будет добавлено автоматически.

Проект уже наполовину готов. Но у нас пока нет возможности нарисовать свою картинку. Для вывода графических примитивов объект Image содержит свойство Canvas — холст. Заметим тут же, что это свойство есть у многих визуальных объектов, в том числе и у Формы, следовательно, можно рисовать и непосредственно на форме.

Для рисования Canvas содержит два инструмента и множество графических примитивов. Инструмент Pen (Ручка) определяет цвет, толщину и стиль линий и границ областей, а инструмент Brush (Кисть) — цвет и стиль заливки области. Сами же графические примитивы рисуются методами, принадлежащими холсту. Перечислим некоторые из них:

Arc — дуга;
Ellipse — закрашенный эллипс или окружность;
FillRect — закрашенный прямоугольник;
LineTo — провести линию в заданную точку;
MoveTo — перейти в заданную точку;
Rectangle — прямоугольная рамка;
TextOut — вывод текста.

Для рисования картинки выберем
Ellipse(X-R, Y-R, X+R, Y+R) — окружность радиуса R с центром в точке x,Y.

Предварительно зададим параметры инструментов:
Pen.Color : = ...; — цвет линии контура;
Pen. Width : = ...; — толщина линии контура;
Brush . Color : = ...; — цвет заливки.

По умолчанию линия является сплошной, но можно задать и другой стиль линии Pen. style:
psSolid— сплошная линия;
psClear — линия не рисуется;
psDash — линия из тире;
psDot — линия из точек и т.д.

К сожалению, все пунктирные и штриховые линии могут быть толщиной только в один пиксель. По умолчанию заливка является сплошной, однако можно задать и другой стиль заливки Brush.style:
bsSolid — сплошная заливка;
bsClear — нет заливки;

bsHorizontal — горизонтальная штриховка;
bsVertical — вертикальная штриховка;
bsFDiagonal — диагональная штриховка;
bsBDiagonal — диагональная штриховка;
bsCross — клетки; bsDiagCross — диагональные клетки.

Осталось запрограммировать собственно рисование. Для этого используем событие OnMouseDown, которое возникает, когда пользователь нажимает кнопку мыши. Если в этот момент курсор мыши находится над рисунком, вызывается процедура TForml.ImagelMouseDown с параметрами:
Sender — отправитель сообщения о событии OnMouseDown;
Button — номер нажатой кнопки (mbLef t, mbRight, mbMiddle);
Shift — нажата ли вспомогательная клавиша Shift|. Alt , Ctrl-
X, Y — координаты базовой точки курсора мыши.

Теперь у вас достаточно информации, чтобы написать процедуру самостоятельно. Завершите проект и поработайте с ним. Нарисуйте картинку из окружностей и сохраните ее на диске. Откройте ее снова, дополните рисунок и запишите его под другим именем.

Как можно улучшить полученную программу? Во-первых, хотелось бы иметь возможность стирать неудачный рисунок. Вызовем Редактор меню и добавим пункт "Очистить". В процедуре обработки события закрасим всю рабочую область рисунка — ClientRect — белым цветом:

```
procedure TForml.N3Click(Sender:  TObject);
begin
  with Image1.Canvas do
  begin
   Brush.Color  := clWhite; FillRect(ClientRect);
  end;
end;
```

Кстати, эту же процедуру можно использовать для закраски фона рисунка при запуске программы. Чтобы не повторять тот же текст в процедуре обработки события OnCreate, возникающего при создании Формы, назначим для этого события уже написанный обработчик.

Для этого в Инспекторе объектов в списке событий Формы щелкнем мышкой на строке OnCreate. Справа появится раскрывающийся список с перечнем всех имеющихся в программе обработчиков. Выберем из списка N3Click — и получим требуемое.

Во-вторых, у нас нет возможности при работе программы изменять параметры окружности. Добавим в проект возможность изменения цвета с помощью компонента ColorGrid (Таблица цветов). Расположен он на вкладке Samples (Примеры).

Щелчок левой кнопки мыши по клетке с некоторым цветом выбирает его как цвет переднего плана (ForegroundColor). Щелчок правой кнопки выбирает цвет заднего плана (BackgroundColor). Для настройки объекта используем свойство GridOrdering — размеры таблицы (go2x8, gol6xl, до4х4 и т.д.). В любом случае нам доступны 16 цветов.

Начальные установки выбранных клеток таблицы:
ForegroundIndex— индекс цвета переднего плана (линии);
BackgroundIndex— индекс цвета заднего плана (заливка).
В выбранных клетках появляются обозначения "FG" и "BG" соответственно (при совпадении клеток — буквы "FB").

Приведем теперь иллюстрацию работы программы "Рисование картинки" и полнный текст процедуры обработки нажатия кнопки мыши.

```
procedure TForml.Image1MouseDown(Sender-: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
begin
with Image1.Canvas do
  begin
    Pen.Color := ColorGrid1.ForegroundColor;
    Brush.Color := ColorGrid1.BackgroundColor;
    Ellipse(X -20, y - 20, X + 20, Y + 20);
  end;
end;
```

Проект "Сложение чисел"

Поле ввода текстовой информации (класс TEdit).
Преобразование текста в число и обратно в текст.
Получение и вывод результата на форму

На этом занятии мы познакомимся еще с одним стандартным компонентом — полем (строкой) ввода TEdit. Заголовка (свойства Caption) у этого компонента нет, но есть свойство Text, определяющее содержимое строки. При необходимости можно ограничить длину вводимой строки с помощью свойства MaxLength. При вводе конфиденциальной информации указывают отображаемые символы (обычно "*"), при этом нужно переопределить свойство PasswordChar, задав его отличным от #0.

Составим проект для суммирования двух чисел, вводимых с клавиатуры. Если предыдущий проект уже сохранен, то для создания нового выполним команду **File || New Application.**

Проект будет содержать метки, поля ввода, командную кнопку и новый для нас объект — панель (класс TPanel). Панель является подобием контейнера, в который мы будем помещать другие объекты. Однако и сама по себе панель может служить для отображения текста (у нее есть свойство Caption), чем мы и воспользуемся в данном проекте. Однако все по очереди.

Создадим заголовок формы — Проект "Сложение чисел", затем выставим свойства шрифта формы, отличные от принятых по умолчанию. Эти свойства будут передаваться всем объектам, помещаемым на форму, но при необходимости для каждого такого объекта их можно изменить обычным способом.

Поместим метку Labell на форму и, в отличие от предыдущего проекта, установим свойство AutoSize в **False,** чтобы отменить минимизацию размера метки под текст надписи. Теперь зададим значение свойства WordWrap=**True**. Тем самым мы получим возможность расположения текста надписи в несколько строк. Наберем теперь текст надписи: «Введите два произвольных числа и нажмите кнопку "Сложить"» — и поэкспериментируем с размером надписи. Мы увидим, как меняется количество строк текста в зависимости от размера. Кроме того, к надписи можно применить различные варианты выравнивания текста. За это "отвечает" свойство Alignment — выравнивание текста внутри объекта. (Это свойство не надо путать со свойством Align.)

Свойство Alignment имеет следующие значения:

taCenter — выравнивание по центру;

taLeftJustify — выравнивание по левому краю;

taRightJustify — выравнивание по правому краю.

Именно последнее мы и применим к Label1.

Поместим теперь на форму кнопку Button1 и два поля ввода — Editl и Edit2. Изменим шрифт в полях ввода и уберем текст в свойстве Text. Добавим вторую метку с текстом "Результат равен" и панель для размещения результата.

Используя свойства BevelInner, BevelOuter И BevelWidth, а также BorderWidth, можно управлять внешним видом панели: создать иллюзию
приподнятости или "утопленности" панели, сделать рамку и т.д.

Проект можно запустить, текст прекрасно вводится в поля ввода, кнопка нажимается, но больше ничего не происходит. Для реального сложения чисел необходимо еще написать процедуру обработки события Onclick для кнопки Buttonl.

Объявим в процедуре TForml.ButtoniCiick переменные A, B, C для хранения чисел типа real:

```
var A, B, C: real;
```

Поскольку строка ввода содержит текст, переведем его в числовое значение процедурой Val *{строка_текста, результат, код ошибки).* Код ошибки равен нулю, если в строке текста нет "посторонних" символов. Будем пока считать, что пользователь правильно вводит исходные данные, тогда код ошибки можно не анализировать:

```
Val(Editl.Text, A, code); Val(Edit2.Text, B, code);
```

Теперь вычислим сумму значений A и B, выполним обратный перевод числа в строку и выведем результат на Панель:

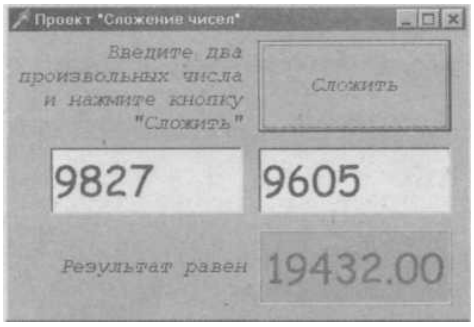
```
C := A + B; Str(C:8:2, S); Panell.Caption := S;
```

Процедура перевода числа в строку имеет вид:

Str *(выражение[формат_вывода], строка-результат).*

Формат вывода задавать необязательно (хотя без него труднее понять результат), поэтому формат указан в квадратных скобках. Итак, мы получили полный текст процедуры:

```
procedure TForml.Button1.Click(Sender: TObject);
var
  A, B, C: real; code: integer; S: string;
begin
  Val(Editl.Text, A, code); Val(Edit2.Text, B, code);
  C := A + B; Str(C:8:2, S);
  Panell.Caption := S
end;
```



Добавьте кнопки и напишите процедуры для вычитания, умножения и деления. Результат выводить на ту же Панель. При этом все переменные, чтобы их не объявлять еще три раза перенести в раздел i**mplementation.**

Проект "Список класса"

Разработаем проект, который позволит пользователю заполнять список своими данными. Пусть это будут данные (фамилия, имя) об учащихся класса. Поместим на форму два поля ввода и снабдим их надписями "Фамилия", "Имя". Ниже разместим кнопку "Добавить в список" и флажок "В алфавитном порядке". Оставшуюся часть формы отведем под список TListBox.
Напишем обработчик события OnClick для кнопки:

```
procedure TForml.ButtonlClick(Sender: TObject);
 begin
   ListBoxl.Iterns.Add(Editl.Text+' '+Edit2.Text);
 end;
```

Заполним поля ввода и нажмем кнопку "Добавить в список". Введенные данные появятся в первой строке списка. Теперь можно набрать данные для следующего ученика. Но придется предварительно щелкнуть мышкой в первом поле ввода, чтобы его активизировать (передать ему "фокус ввода"). Удобнее сделать это программным путем, добавив строку Edit I.SetFocus ; к телу процедуры TForml.ButtonlClick.
Заметим, что для перехода от первого поля ввода клавишу Tab. По второму и далее можно использовать клавишу рядок "посещения" управляющих элементов определяет
свойство TabOrder, содержащееся в каждом из них. По умолчанию он совпадает **с** порядком их создания при разработке проекта. Можно сделать элемент управления недоступным для клавиши Tab , установив свойство TabStop равным False. Изменить порядок посещения можно непосредственным изменением свойства TabOrder. Более удобным представляется использование редактора Edit Tab Order. Его можно вызвать из контекстного меню, появляющегося при щелчке правой кнопки мыши на форме.
В левой части перечислены имеющиеся на форме элементы управления. Стрелки в правой части окна позволяют передвинуть выделенный элемент управления на нужное место.
Итак, проект заработал, осталось только добавить обработчик события OnClick для флажка:

```
Procedure TForml.CheckBoxlClick (Sender:, TObject);
begin
   ListBoxl.Sorted:=CheckBoxl.Checked;
end;
```

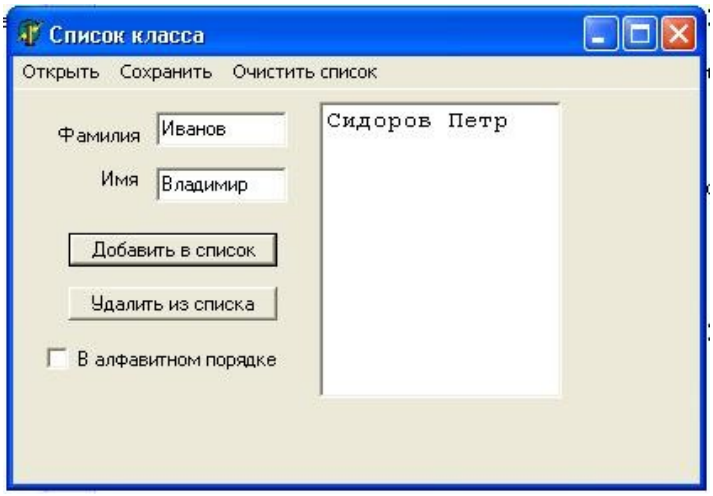
Подумаем теперь, каких возможностей не хватает в проекте? Во-первых, мы не можем исправить неверную информацию, уже внесенную в список. Во-вторых, и это даже важнее, чем "во-первых", в программе не предусмотрена возможность сохранения информации на диске. В результате после завершения программы все данные пропадают.
Для удаления неверных данных воспользуемся методом Delete. Для простоты будем предполагать, что множественный выбор запрещен. Добавим на форму кнопку "Удалить из списка" и напишем процедуру удаления выделенной строки из списка:

```
procedure TForml.Button2Click(Sender: TObject);
 begin
   ListBoxl.Items.Delete(ListBoxl.Itemlndex) ;
end;
```

Сохранение данных на диске и их последующее использование обеспечивается стандартными диалогами TOpenDialog и TSaveDialog для работы с текстовыми файлами. Для вызова этих диалогов удобно использовать меню, как это было сделано в проекте "Редактор текста" Приведем здесь только программный код пунктов меню, иллюстрирующий использование методов LoadFromFile и SaveToFile для списка строк.
Пункт "Открыть":

```
procedure TForml.NIClick(Sender: TObject);
```

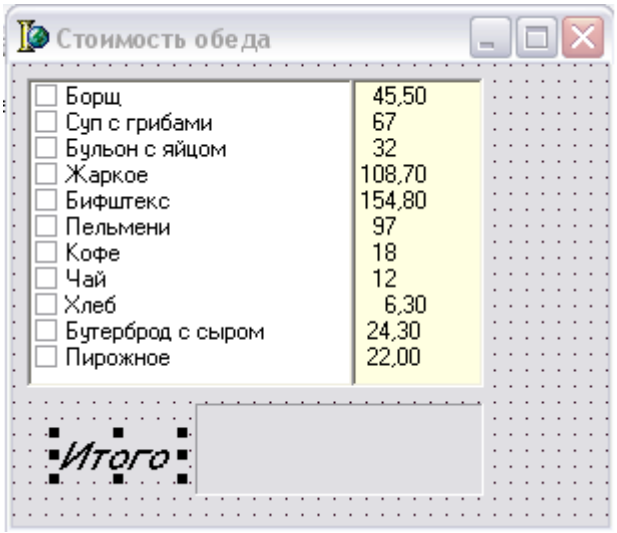
```
begin
with OpenDialog1 do
  begin
    if Execute then
      ListBox1.Items.LoadFromFile(FileName);
  end;
Пункт "Сохранить":
procedure TForm1.N2Click(Sender: TObject);
begin
with SaveDialog1 do
    begin
     if Execute then
     begin
    ListBox1.Items.SaveToFile(FiieName);
     end;
   end;
 end;
```



Проект «Стоимость обеда»

Имеется еще один компонент, очень похожий на **ListBox** — это список с индикаторами **CheckListBox**. Выглядит он так же, как **ListBox**, но около каждой строки имеется индикатор, который пользователь может переключать. Индикаторы можно переключать и программно, если список используется для вывода данных и необходимо в нем отметить какую-то характеристику каждого объекта, например, наличие товара данного наименования на складе. Все свойства, характеризующие компонент **CheckListBox** как список, аналогичны **ListBox**, за исключением свойств, определяющих множественный выбор. Эти свойства компоненту **CheckListBox** не нужны, поскольку в нем множественный выбор можно осуществлять установкой индикаторов. Состояния индикаторов определяют два свойства: **State** и **Checked**. Оба эти свойства можно рассматривать как индексированные массивы, каждый элемент которого соответствует индексу строки. Эти свойства можно устанавливать программно или читать, определяя установки пользователя.

Установим для формы свойство BorderStyle bsDialog**,** для того, чтобы нельзя было изменить размер формы.
На форме разместим объект класса TCheckListBox для отображения списка меню и рядом TListBox для отображения списка стоимостей этих блюд.
Для первого занесем список в Items названия блюд
Для второго их стоимости. Данные берем из рисунка.
TLabel для надписи «Итго:» и TPanel для вывода суммы.



Остается написать процедуру, которая будет реагировать на выбор в **CheckListBox** и подсчитывать сумму обеда, используя переменную S. StrToFloat преобразует текстовую переменную считанную из ListBox1 в число. FloatToStr преобразует число в текст, который выводится в Panel1.

```
procedure TForm1.CheckListBox1Click(Sender: TObject);
var i : word;
  S : double;
begin
  S:=0;
  with CheckListBox1 do
  for i:=0 to Items.Count-1 do
     if (Checked[i]) and (ListBox1.Items.Strings[i]<>'') then
          S:=S+StrToFloat(ListBox1.Items[i]);
  Panel1.Caption:=FloatToStr(S);
end;
```

Count – число элементов в списке, Count-1 потому, что первый элемент в списке нулевой.
Сложное условие (Checked[i]) and (ListBox1.Items.Strings[i]<>'') выбирает отмеченные блюда в CheckListBox1 и есть ли в списке ListBox1 стоимость для выбранных блюд.