Библиотечка «Первого сентября»

Английский язык
Библиотека в школе
Биология
География
Дошкольное образование
Здоровье детей

Информатика

№3/2005

Искусство
История
Литература
Математика
Начальная школа
Немецкий язык
Русский язык
Спорт в школе
Управление школой
Физика
Французский язык
Химия
Школьный психолог

Г.Н. ГУТМАН

Учебные мини-проекты на DELPHI

БИБЛИОТЕЧКА «ПЕРВОГО СЕНТЯБРЯ»
Серия «Информатика»
Выпуск 3

Г.Н. Гутман

УЧЕБНЫЕ МИНИ-ПРОЕКТЫ НА DELPHI

Москва
Чистые пруды
2005

Введение

Предлагаемый сборник учебных мини-проектов предназначен для первоначального знакомства со средой Delphi. Книг “Освой Delphi за 5 минут” (или за час, два, три и т.д.) имеется достаточное количество. Среди них много весьма добротных, действительно позволяющих быстро приступить к самостоятельной работе. Однако, что, в общем, неудивительно, имеется крайне мало литературы начального уровня, предназначенной для освоения Delphi применительно к использованию в учебном процессе. Профессиональным программистам такие задачки, как мини-калькулятор или учебный графический редактор, неинтересны — им, скорее, подавай примеры работы с базами данных по технологии “клиент-сервер”. А учителю это как раз совсем не нужно.

Delphi можно изучать и использовать на нескольких уровнях:

1. Работа с визуальными объективами практически без программирования.
2. Использование готовых компонентов системы с написанием на их основе собственного программного кода.
3. Создание собственных компонентов на языке Паскаль и включение их в палитру компонентов Delphi в качестве стандартных.
4. Разработка законченных Windows-приложений.

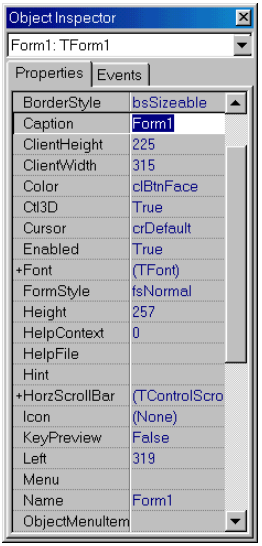
Для школьного курса алгоритмизации более чем достаточно первого уровня (задачи второго уровня можно решать в курсе профильной школы и на факультативах), поэтому данное пособие соответствует именно первому уровню освоения Delphi.

Исходные файлы всех описанных проектов находятся в свободном доступе на сайте газеты “Информатика” http://inf.1september.ru в разделе “Download”.

Знакомство со средой Delphi

Запуск приложения. Изменение свойств Формы. Сохранение приложения

При запуске Delphi на экране появляются несколько окон, которые в совокупности и образуют среду программирования. Сверху расположено главное окно, которое содержит меню, панель инструментов и Палитру компонент. Слева — окно Инспектора объектов, справа — заготовка будущего приложения — Форма.



Под Формой спрятано еще одно окно — Редактор кода программы. Если заглянуть в него, то окажется, что окно уже содержит некоторый текст. Это заготовка модуля будущей программы, написанная Delphi автоматически. Модуль неразрывно связан с Формой. Каждое наше действие, изменяющее внешний вид и содержание Формы, будет автоматически приводить к изменению программного кода.

Вернемся к Инспектору объектов. Окно Инспектора объектов состоит из двух вкладок (страничек), в одной из которых (`Properties`) перечислены свойства объекта, в другой (`Events`) — события, на которые данный объект может реагировать. Список всех имеющихся на Форме объектов расположен вверху окна в раскрывающемся списке.

Пока что там находится всего один объект `Form1` — сама Форма. Имена объектам присваивает сама система Delphi, их можно изменять, однако в первых проектах мы этого делать не будем. Посмотрим на список свойств — он достаточно длинный, и сначала кажется, что разбираться в нем долго и сложно. Однако значения всех свойств уже установлены, поэтому разбираться придется только с теми свойствами, которые мы захотим изменить. Можно даже ничего не менять — и сразу запустить программу (написанную за нас Delphi) на выполнение! Для этого можно нажать клавишу F9 или треугольник на панели инструментов или выбрать пункт **Run** в меню.

На экране появится окно нашего приложения, которое наделено базовыми свойствами любого приложения Windows: окно можно перемещать по экрану, изменять его размеры, сворачивать, разворачивать и т.д. Завершим наш эксперимент нажатием на кнопку закрытия окна приложения.

Рассмотрим основные свойства Формы.

Положение Формы на экране.

`Left` — координаты левой границы Формы на экране;

`Top` — координаты правой границы Формы на экране.

Эти свойства определяют, в каком месте экрана будет появляться окно приложения при запуске программы.

Поставим эксперимент: возьмем Форму за строку заголовка и перетащим ее в другое место, следя за значением свойства `Left` (или `Top`). Мы увидим, как это значение меняется на наших глазах! Можно сделать и наоборот: щелкнем левой кнопкой мышки по строке со свойством `Left` и изменим с клавиатуры его значение. Как только мы нажмем Enter, мы увидим, что Форма "перепрыгнула" в другое место экрана!

Размеры Формы.

`Height` — высота Формы;

`Width` — ширина Формы.

Эти свойства задают размеры Формы. Следует, однако, учесть, что в эти размеры включены заголовок и граница Формы. Если же нам, как это обычно и бывает, нужна только рабочая область Формы, то ее размеры задаются в свойствах

`ClientHeight` — высота рабочей области Формы;

`ClientWidth` — ширина рабочей области Формы.

Заголовок Формы — свойство `Caption`.

Чтобы изменить заголовок Формы, необходимо набрать желаемый текст в поле свойства. В процессе набора мы видим, как изменяется текст в строке заголовка.

Цвет Формы — свойство `Color`.

В отличие от рассмотренных выше свойств, при выделении строки свойства в Инспекторе объектов в поле значения появляется стрелочка вниз. Нажав на нее, мы получим возможность выбрать цвет из списка. Все элементы этого списка являются символическими константами и начинаются с символов "cl" , входящих в слово "color". Названия некоторых цветов указывают на область их применения.

Измените свойства Формы и запустите программу.

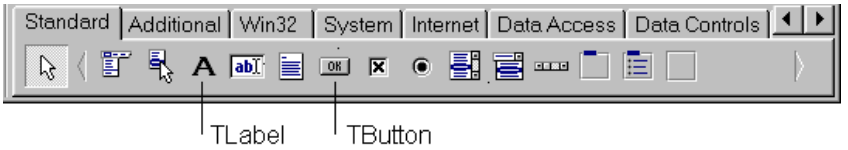
Осталось научиться записывать созданную программу, точнее, проект, так как Delphi создает еще несколько вспомогательных файлов.

Для этого необходимо выполнить команду **File || Save Project As**. В появившемся диалоговом окне выбрать (или создать) папку для записи файлов проекта *Unit1.pas* и *Project1.dpr*. Рекомендуется при этом сменить имя "Project1" на другое, отражающее суть программы, так как создаваемый Delphi exe-файл будет иметь имя проекта.

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!

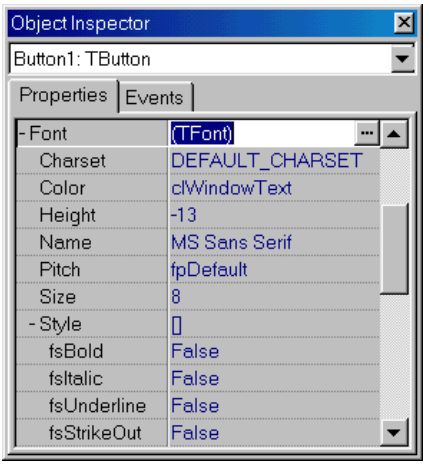
Поместим на Форму надпись `Label1` и две кнопки — `Button1` и `Button2`. (Еще раз напомним, что имена “по умолчанию” объектам дает Delphi.)

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 TForm1.Button1Click(Sender: TObject);
begin

end;
```

`TForm1` — это имя класса, в который включена процедура `Button1Click`. Имя процедуры состоит из имени объекта `Button1` 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:

```
Label1.Font.Size := Label1.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 = 96;
      MinSize = 6;
```

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

В процедуре `TForm1.Button1Click` перед присваиванием добавим проверку:

```
if Label1.Font.Size*2 <= MaxSize then
   Label1.Font.Size := Label1.Font.Size*2;
```

Для того чтобы не писать каждый раз префикс `Label1.Font`, можно воспользоваться оператором `with`:

```
with Label1.Font do begin
  if Size*2 <= MaxSize then
     Size := Size*2;
end;
```

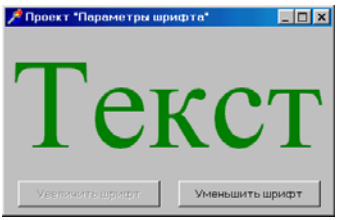
Аналогичную процедуру легко написать и для второй кнопки.

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 Label1.Font do begin
 if Size*2 <= MaxSize then Size := Size * 2
 if Size*2 > MaxSize then Button1.Enabled := False;
end;
```

Но как после этого восстановить работоспособность кнопки `Button1`? Включить кнопку так же просто, как и выключить, достаточно выполнить присваивание `Button1.Enabled := True;` нужно только понять, куда его вставить. Понятно, что если кнопка `Button1` выключена, то включить ее можно только с помощью кнопки `Button2`, и наоборот. Добавим поэтому в процедуру `TForm1.Button2Click` строку:




```
Button1.Enabled := True;
```

и все получится!

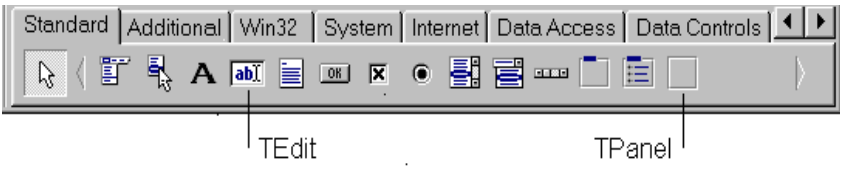
Напишите процедуру `TForm1.Button2Click` для уменьшения размера шрифта.

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

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 `Label1` 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” свой-

ство 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;

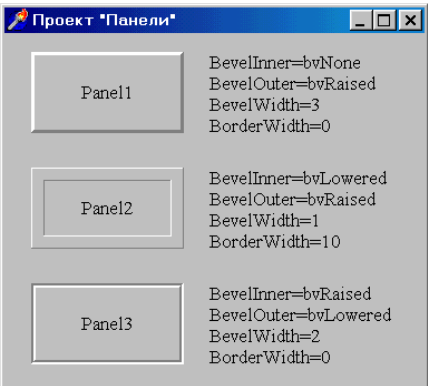
taRightJustify — выравнивание по правому краю.

Именно последнее мы и применим к Label1.

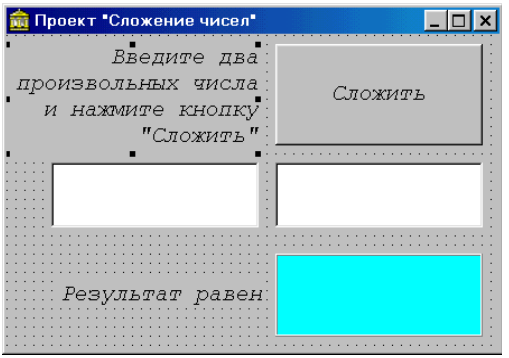
Поместим теперь на форму кнопку Button1 и два поля ввода — Edit1 и Edit2. Изменим шрифт в полях ввода и уберем текст в свойстве Text. Добавим вторую метку с текстом “Результат равен” и панель для размещения результата.

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.

Ниже показаны варианты панелей, полученных при разных сочетаниях ее свойств.



Теперь все объекты присутствуют на Форме.



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 `Button1`.

Объявим в процедуре `TForm1.Button1Click` 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(Edit1.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); Panel1.Caption := S;
```

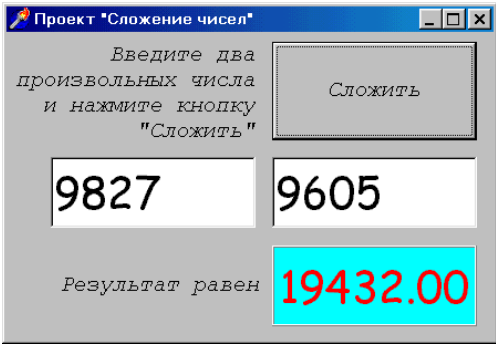
Процедура перевода числа в строку имеет вид:

`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 TForm1.Button1Click(Sender: TObject);
var
 A, B, C: real; code: integer; S: string;
begin
 Val(Edit1.Text, A, code); Val(Edit2.Text, B, code);
 C := A + B; Str(C:8:2, S);
 Panel1.Caption := S
end;
```



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.

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

— помещение `MainMenu` на Форму;

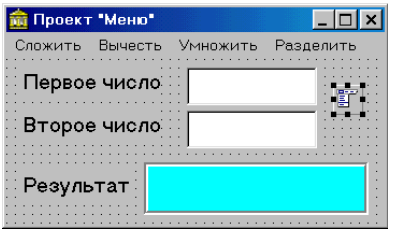
— вызов Конструктора меню двойным щелчком по значку (или через свойство `Items` в Инспекторе объектов);

— 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 `PopupMenu`, где и указывается нужное меню. Создается `PopupMenu` аналогично главному меню.



Пункты главного меню 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 пункта меню, следует написать для каждого “конечного” пункта меню процедуру обработки данного события. Таким образом, нам придется написать четыре почти одинаковые процедуры, так 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 будем передавать ей символ операции: “+”, “–”, “*” или “/”.

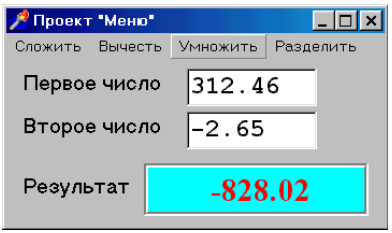
```
procedure Exec(ch: char);
var
 A, B, C: real;
 code: integer;
 S: string;
begin
 Val(Form1.Edit1.Text, A, code);
 Val(Form1.Edit2.Text, B, code);
 case ch of
 '+': C := A + B;
 '–': C := A - B;
 '*': C := A * B;
 '/': C := A / B
 end;
 Str(C:8:2, S);
 Form1.Panel1.Caption := S
end;
```

Тогда “обработчик” пункта меню “Сложить” будет выглядеть так:

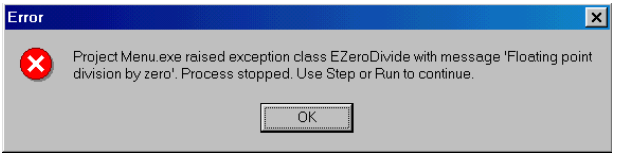
```
procedure TForm1.N1Click(Sender: TObject);
begin
 Exec('+')
end;
```

Аналогично пишутся обработчики других пунктов меню.

Запустим программу и поработаем с ней:



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

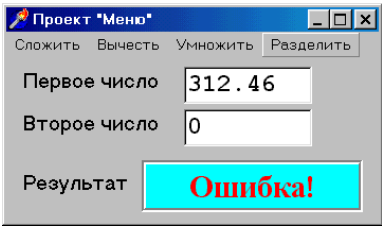


(Проект *Menu.exe* вызвал исключение “Деление на ноль”. Процесс остановлен. Используйте `Step` или `Run` для продолжения.)

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);
label m99;
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!';
           goto m99
      end;
 end; {case}
 Str(C:8:2, S);
m99: Form1.Panel1.Caption := S
end;
```

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



Проект “Подбор цвета”

Формирование цвета из отдельных компонент. Класс TColor, константы цвета, функция RGB

Цвета объектов образуются смешением трех компонент — красной (*red*), зеленой (*green*) и синей (*blue*). Интенсивность каждой составляющей цвета может изменяться от 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;

`SmallChange` — 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.

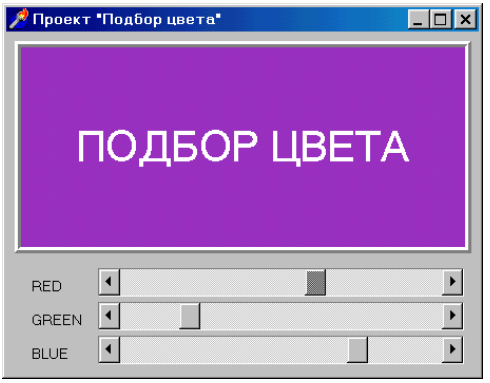
Напишем отдельную процедуру изменения цвета панели

```
procedure SetPanelColor;
var red, green, blue : word;
begin
  red := Form1.ScrollBar1.Position;
  green := Form1.ScrollBar2.Position;
  blue := Form1.ScrollBar3.Position;
  Form1.Panel1.Color := RGB(red, green, blue);
end;
```

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

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
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.



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 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.

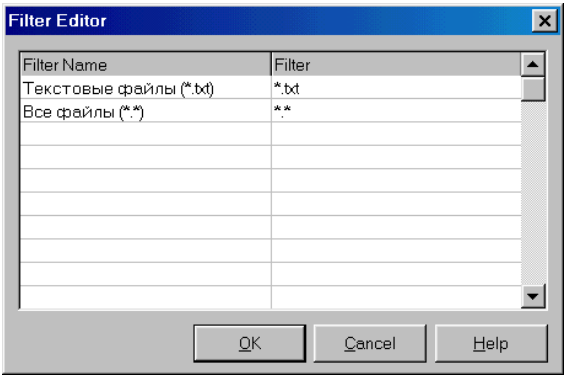
Все диалоги относятся к невизуальным компонентам, поэтому могут быть размещены в любом месте формы, в том числе и на других объектах.

Начнем создание проекта "Редактор текста".

Разместим на форме объект `Memo1`. Установим свойства `ScrollBars = = ssVertical`, `Align = alClient` (полное заполнение формы). Добавим диалоги `OpenDialog1`, `SaveDialog1` и `FontDialog1`. 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.

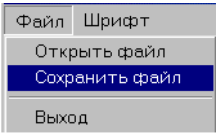
Его внешний вид показан на рисунке.



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.

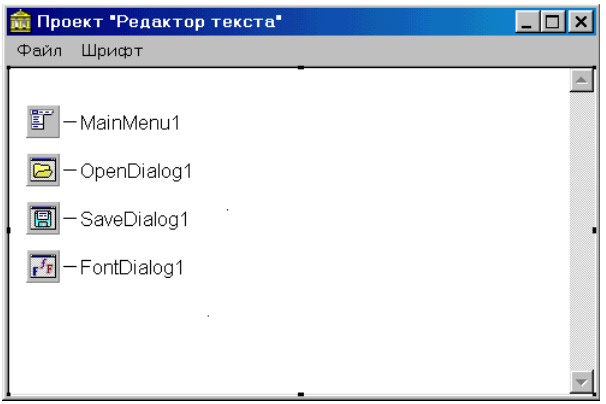
Аналогично настраивается диалог `SaveDialog1`. В нем только дополнительно зададим свойство `DefaultExt = txt`, чтобы расширение *txt* автоматически добавлялось к имени при записи файла.

Для удобства управления проектом создадим меню, при выборе пунктов которого будут вызываться диалоги. На рисунке показан момент выбора пункта “Сохранить файл”.



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

Общий вид проекта на этапе разработки показан на рисунке.



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:

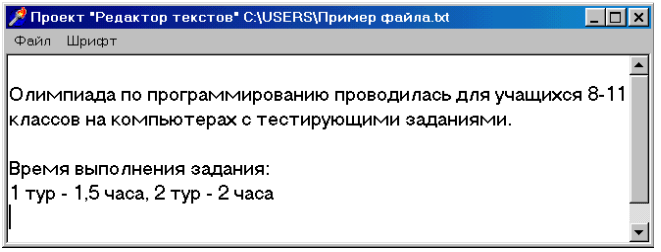
```
with OpenDialog1 do begin
  if Execute then begin
    Memo1.Lines.LoadFromFile(FileName);
    Form1.Caption := 'Проект "Редактор
                     текстов" '+FileName;
  end;
end;
```

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;
```

Окно работающего проекта показано на рисунке.



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 TBrush 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 BevelInner = 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 DefaultExt 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 `TForm1.Image1MouseDown` с параметрами:

`Sender` — отправитель сообщения о событии `OnMouseDown`;

`Button` — номер нажатой кнопки (`mbLeft`, `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 TForm1.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`, `go16x1`, `go4x4` и т.д.). В любом случае нам доступны 16 цветов.

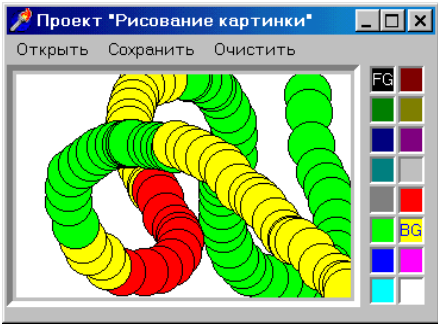
Начальные установки выбранных клеток таблицы:

`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.



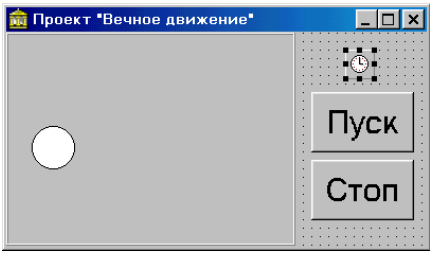
```
procedure TForm1.Image1MouseDown
          (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
                        X, Y: Integer);
begin
  with Image1.Canvas do begin
   Pen.Color := ColorGrid1.ForegroundColor;
   Brush.Color := ColorGrid1.BackgroundColor;
   Ellipse(X - 20, y - 20, X + 20, Y + 20);
  end;
end;
```

Проект “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 O*x* — dx, а к Top — величину смещения по оси O*y* — 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 TForm1.Timer1Timer(Sender: TObject);
begin
   with Shape1 do begin
    if (Left <= 0) or (Left + Width >= Panel1.Width)
    then dx := -dx;
    Left := Left + dx;
    if (Top <= 0) or (Top + Height >= Panel1.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 `Timer1.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 “Графический редактор”

Цель данного занятия — создать простейший графический редактор. Левой кнопкой мыши мы сможем рисовать непрерывную кривую, а правой — закрашивать выбранным цветом получающиеся области.

Поместим на форму объект типа `TImage`. Для отслеживания траектории движения мыши напишем обработчик события `OnMouseMove`:

```
procedure TForm1.Image1MouseMove
  (Sender: TObject;
  Shift: TShiftState; X, Y: integer);
begin
  if ssLeft in Shift
  then Image1.Canvas.LineTo(X,Y);
end;
```

Запустим проект и попробуем перемещать мышь с нажатой левой кнопкой. За мышкой останется след. Отпустим кнопку — и рисование прекратится. Но стоит нам снова нажать левую кнопку мыши, как нарисованная кривая соединяется прямой линией с курсором мыши. Понятно, как исправить ошибку: нужно в момент нажатия кнопки “перескакивать” в новое положение курсора без рисования. Сделаем это в обработчике события `OnMouseDown`:

```
procedure TForm1.Image1MouseDown
  (Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState;
  X, Y: integer);
begin
  Image1.Canvas.MoveTo(X,Y);
end;
```

Теперь перейдем к закрашиванию. Класс `Canvas` содержит процедуру `FloodFill`, которая может работать в двух вариантах: или закраши-

вать область текущим цветом до границы заданного цвета, или перекрашивать точки заданного цвета до границы любого другого цвета (в этом случае граница может состоять даже из частей разного цвета!). Различаются эти варианты значением последнего параметра процедуры — fsBorder или fsSurface. Например, закрасим красным область с границей черного цвета:

```
Canvas.Brush.Color := clRed;
Canvas.FloodFill(X,Y, clBlack, fsBorder);
```

А теперь перекрасим красную область в синий цвет:

```
Canvas.Brush.Color := clBlue;
FloodFill(X,Y, clRed, fsSurface);
```

Обычно мы не знаем заранее, какого цвета область, которую мы собираемся перекрашивать. Но это легко можно определить с помощью свойства Pixels, в котором хранятся цвета всех точек изображения. Pixels[X,Y] и есть цвет точки канвы с координатами X, Y.

Завершим создание редактора. Поместим на форму компонент TColorGrid. При изменении компонента будем изменять и цвет закрашивания:

```
procedure TForm1.ColorGrid1Change (Sender: TObject);
begin
    Image1.Canvas.Brush.Color := ColorGrid1.BackgroundColor;
end;
```

Перепишем теперь обработчик события OnMouseDown:

```
procedure TForm1.Image1MouseDown
  (Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState;
  X, Y: integer);
begin
  with Image1.Canvas do
  case Button of
  mbLeft:  MoveTo(X,Y);
  mbRight: FloodFill(X,Y, Points[X,Y], fsSurface);
  end;
end;
```

Внесем теперь некоторые усовершенствования. Во-первых, с помощью компонента TColorGrid мы можем изменять и цвет линии. Для этого достаточно в обработчик события OnChange добавить строку:

```
Image1.Canvas.Pen.Color := ColorGrid1.ForegroundColor;
```

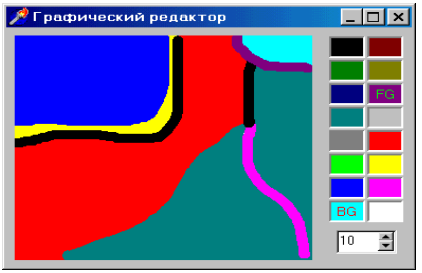
Во-вторых, можем изменять толщину линии. Используем для этого компонент TSpinEdit, расположенный на вкладке Samples. TSpinEdit представляет собой поле ввода, воспринимающее только числовые зна-

чения, и стрелки вверх и вниз, с помощью которых пользователь может “прокручивать” в заданных пределах (свойства `MinValue` и `MaxValue`) значения в поле ввода. Заметим, что границы, установленные для изменения значения `TSpinEdit`, можно нарушать при вводе данных с клавиатуры. Если же на этапе выполнения программы стереть содержимое поля ввода, то возникнет ошибка преобразования пустого значения в число. Для предотвращения этих неприятностей можно запретить редактирование с клавиатуры значения поля ввода (свойство `EditorEnabled` установить в `False`).

Напишем обработчик события `OnChange` для объекта `SpinEdit1`:

```
procedure TForm1.SpinEdit1Change
  (Sender: TObject);
begin
  Image1.Canvas.Pen.Width := SpinEdit1.Value;
end;
```

На рисунке можно видеть, что получилось у автора при работе с программой “Графический редактор”.



Проект “Расчет оплаты”

Воспользуемся приобретенными знаниями, чтобы разработать программу для вычисления оплаты за работу в сети Интернет по смешанному тарифу. При таком способе оплачивается время работы в сети и, дополнительно, информация, полученная из Интернета. Обозначим:

`C_time` — тариф, по которому оплачивается время работы (условных единиц за минуту);

`C_traf` — тариф, по которому оплачиваются полученные из сети данные (условных единиц за 1 Мб информации);

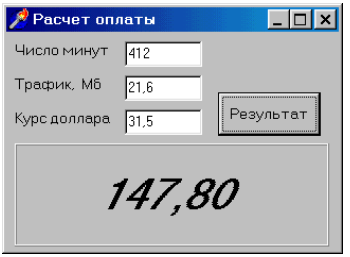
`C_kurs` — курс условной единицы (в рублях).

Тогда размер оплаты вычисляется по формуле: `Val:=(C_time* Time + C_traf*Traf)*C_kurs;` — где `Time` и `Traf` — время работы в сети (в минутах) и количество полученной информации (в мегабайтах).

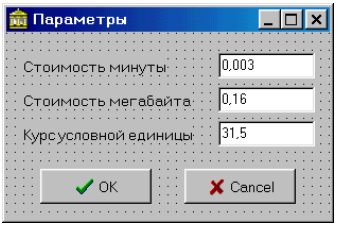
Поместим на форму поля для ввода исходных данных и курса условной единицы. Тарифы оплаты будем считать постоянными и задавать их в самой программе. Выводить результат будем на панель по нажатию командной кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var Time, Traf, Val, C_kurs : double;
begin
 Time := StrToFloat(Edit1.Text);
 Traf := StrToFloat(Edit2.Text);
 C_kurs := StrToFloat(Edit3.Text);
 Val := (C_time*Time + C_traf*Traf)*C_kurs;
  Panel1.Caption := FloatToStrF(Val, ffFixed, 10, 2);
end;
```

Примерный вид работающей программы приведен на рисунке.



Обдумывая проект, мы предполагали, что курс условной единицы будет изменяться со временем, а тарифы — нет. Однако это допущение может и не выполняться. Конечно, можно добавить на форму еще два поля — для ввода тарифов, но это сделает громоздким интерфейс программы. Выход из этой ситуации — создать отдельную форму для ввода параметров и вызывать ее в случае необходимости из главной формы.



Уберем надпись и окно для ввода курса доллара из формы “Расчет оплаты” и добавим кнопку “Параметры” для вызова второй формы. Создать заготовку второй формы можно командой **File || New Form**. При этом создается и новый модуль Unit2, описывающий новый класс TForm2, и экземпляр этого класса — переменная Form2.

Новый класс является производным от того же базового класса TForm и имеет те же возможности, что и первая форма. Следует только отметить, что нумерация объектов, вставляемых в каждую форму, начинается с единицы. Таким образом, три поля ввода во второй форме будут по умолчанию иметь имена Edit1, Edit2 и Edit3. Чтобы отличить объекты одной формы от объектов другой формы, в необходимых случаях мы будем указывать имя соответствующей формы.

Кроме полей ввода и поясняющих надписей, поместим на форму две кнопки: “OK” и “Cancel”. Кнопка “OK” служит для подтверждения сделанных пользователем изменений, нажатием кнопки “Cancel” можно даже после ввода новых значений параметров отказаться от их реального изменения. Эти (и некоторые другие) кнопки являются стандартными для организации пользовательского интерфейса. Эти кнопки являются объектами класса TBitBtn (вкладка Additional палитры компонентов). На поверхности такой кнопки можно поместить небольшую картинку — пиктограмму.

Основные свойства и события для кнопок типа TButton и TBitBtn совпадают. В частности, надпись на кнопке — это свойство Caption, параметры шрифта устанавливаются с помощью свойства Font и т.д. Дополнительно, кнопка TBitBtn содержит следующие свойства:

Kind — тип кнопки. В частности, bkOK — кнопка “OK”, bkCancel — кнопка “Cancel”, bkHelp — кнопка “Help” и т.д. Это стандартные кнопки с уже готовым набором рисунков. Можно создать и нестандартную кнопку, если установить значение свойства равным bkCustom;

Glyph (читается “глиф”) — набор пиктограмм на кнопку. Обычно в наборе две пиктограммы: первая соответствует “действующей” кнопке, вторая — отключенной кнопке;

NumGlyphs — число пиктограмм в наборе;

Layout — расположение пиктограммы относительно надписи на кнопке (по умолчанию — слева);

и некоторые другие, играющие вспомогательную роль.

Вернемся к нашему проекту. Он состоит из двух модулей, поэтому, чтобы каждый из них имел доступ к общим переменным C_time, C_traf, C_kurs, разместим их в разделе **interface** одного из модулей.

Напишем теперь процедуры обработки кнопок.

Для первой формы

Кнопка “Результат” вычисляет результат и выводит его на панель:

```
procedure TForm1.Button1Click(Sender: TObject);
var Time, Traf, Val: double;
begin
  Time := StrToFloat(Edit1.Text);
  Traf := StrToFloat(Edit2.Text);
  Val := (C_time*Time + C_traf*Traf)*C_kurs;
  Panel1.Caption := FloatToStrF(Val, ffFixed, 10, 2);
end;
```

Кнопка “Параметры” показывает окно второй формы:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form2.Show;
end;
```

Поскольку из первого модуля мы вызываем методы класса `TForm2`, в раздел **`implementation`** следует добавить указание **`uses`** `Unit2;` иначе система Delphi “спросит” у нас разрешения сделать это самой.

Для второй формы

Кнопка “OK” вычисляет новые значения параметров и закрывает окно второй формы. После чего посылает сообщение `OnClick` кнопке “Результат” первой формы, тем самым запуская обработчик этой кнопки.

```
procedure TForm2.BitBtn1Click(Sender: TObject);
begin
  C_time := StrToFloat(Edit1.Text);
  C_traf := StrToFloat(Edit2.Text);
  C_kurs := StrToFloat(Edit3.Text);
  Form2.Close;
  Form1.Button1.Click;
end;
```

Очевидно, что во второй модуль необходимо добавить указание **`uses`** `Unit1;` поскольку мы обращаемся к методам класса `TForm1`.

Кнопка “Cancel” закрывает окно второй формы:

```
procedure TForm2.BitBtn2Click(Sender: TObject);
begin
  Form2.Close;
end;
```

И последнее. Инициализацию общих переменных удобнее выполнить при создании второй формы, взяв в качестве начальных значения, установленные в Инспекторе объектов при конструировании формы.

```
procedure TForm2.Create(Sender: TObject);
begin
  C_time := StrToFloat(Edit1.Text);
  C_traf := StrToFloat(Edit2.Text);
  C_kurs := StrToFloat(Edit3.Text);
end;
```

УДК 372.800.2
ББК 74.263.2
Г97

Общая редакция серии «Информатика» *С.Л. Островский*

Гутман Г.Н.

Г97 Учебные мини-проекты на Delphi / Г.Н. Гутман. – М. : Чистые пруды, 2005. – 32 с. : ил. (Библиотечка «Первого сентября», серия «Информатика»).

ISBN 5-9667-0050-8

Сборник учебных мини-проектов предназначен для первоначального знакомства со средой Delphi. Может быть использован учителем при подготовке к урокам по разделу “Алгоритмизация” в основной школе и в рамках профильных курсов, а также для дополнительных занятий со школьниками.

УДК 372.800.2
ББК 74.263.2

Учебное издание

ГУТМАН Геннадий Натанович

УЧЕБНЫЕ МИНИ-ПРОЕКТЫ НА DELPHI

Редактор *С.Л. Островский*
Корректор *Е.Л. Володина*
Компьютерная верстка *Н.И. Пронская*

Свидетельство о регистрации СМИ ПИ № ФС77–19078 от 08.12.2004 г.

Подписано в печать 05.05.2005.
Формат 60x90$^{1}/_{16}$. Гарнитура «Таймс». Печать офсетная. Печ. л. 2,0.
Тираж экз. Заказ №

ООО «Чистые пруды», ул. Киевская, 24, Москва,121165.
http://www.1september.ru

Отпечатано с готовых диапозитивов в Раменской типографии
Сафоновский пр., д. 1, г. Раменское, МО, 140100.
Тел. 377-0783. E-mail: ramtip@mail.ru

ISBN 5-9667-0050-8